Workload-Specific File System Benchmarks

A thesis presented

by

Keith Arnold Smith

to

The Division of Engineering and Applied Sciences

in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

in the subject of

Computer Science

Harvard University
Cambridge, Massachusetts

January, 2001

To Maddie, who didn't understand why Daddy had to work late

And to Jackie, who did

Abstract

A fundamental problem with the current generation of file system benchmarks is that they fail to take into account the fact that a file system's performance can vary depending on the workload running on it. Many benchmarks attempt to reduce file system performance to a single number, producing a simplistic one-dimensional ordering of the systems being tested. Although this may be useful for marketing literature, the performance of file systems in the real world is more complicated. Different workloads place different demands on the file system, and can result in different behavior from the underlying system. A file system that provides superior performance for a web server may have inferior performance when running a software development workload.

In this dissertation I demonstrate that the "one size fits all" approach of current file system benchmarks does not accurately predict the performance of different workloads on different file systems. I then present a new benchmarking methodology that not only predicts file system performance in the context of a specified workload, but also allows researchers and developers to isolate the areas of file system performance that present the greatest bottlenecks for particular workloads.

Acknowledgements

During my time at Harvard, I have been fortunate to enjoy the advice, support, and encouragement of many friends and colleagues. This dissertation could not have been completed without them.

First and foremost, I would like to thank my advisor, Margo Seltzer, who provided invaluable assistance and guidance at every stage of my graduate career. Margo was always available and willing to help with any problem I was facing—large, small, or completely unrelated to my graduate studies. She not only guided me through this project, she also showed me how to conduct computer systems research. I cannot imagine coming this far without Margo's guidance and counsel, and I cannot imagine working on any future project without referring back to the many lessons I learned from her.

My thesis committee—Margo, Brad Chen, Ugo Gagliardi, and Barbara Grosz—also provided essential guidance, especially in finishing my dissertation. They were patient with the (seemingly) endless delays in completing it, and their comments and suggestions improved my understanding of this research.

I am also grateful to all my graduate student colleagues, for brainstorming sessions, lunch dates, and endless conversation on topics ranging from compiler optimization to Chinese culture. Chris Small, a past and future office mate, has provided help at every step of the way, from configuring new machines to (re)learning how to drive a car. Catherine Zhang patiently listened to all of the details of my job search. Yaz Endo taught me the value of a power drill. Cliff Young was an ideal lunch companion. Ellie Baker provided tea, cookies, and friendship during first year courses on distributed

algorithms and computer graphics. David Holland's sharp eyes and quick mind helped me to find innumerable bugs in my software.

I have had few experiences as enlightening and exhilarating as the first lecture of Alan Perlis's introductory computer science course at Yale. Until that moment, I had no intention of majoring in computer science, which I viewed as nothing more than learning how to program. Afterwards, I could not imagine myself studying anything else. Without his infectuous enthusiasm for this subject, I would probably be an electrical engineer.

I also wish to thank my parents. Dad explained how rockets, televisions, and airplanes work. He taught me that if something is worth doing, it is worth doing right. Mom gave me my first lessons about computers, which she was programming before I was born. Together they encouraged me to pursue my interests and satisfy my curiosities. Their love, support, and encouragement set me on the trajectory that brought me to graduate school.

Finally, I extend my deepest love and gratitude to my wife, Jackie Horne. She encouraged me to return to school to pursue first my masters and then my doctorate, and she endured the consequences of her advice. She was patient and supportive during all of the paper crunches, including this final one. Having Jackie at the center of my life helped me keep the trials and tribulations of graduate study in perspective. I could not have completed this without her love.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

In his keynote address at the first Symposium on Operating Systems Design and Implementation, David Patterson observed that, “For better or for worse, benchmarks shape a field [Patterson94].” When a benchmark gains wide-spread acceptance, vendors and researchers use the benchmark as a yardstick for measuring improvements to their systems. Thus, an important focus for improving system performance is to improve the system’s benchmark results.

This narrow focus on benchmark results can be either good or bad, depending on the quality of the benchmark. A good benchmark accurately reflects the usage patterns of an important class of applications and provides results that predict the performance of those applications. With a good benchmark, targeting system improvements to increase benchmark performance results in better performance for end users. In contrast, a poor benchmark leads researchers and developers to tune their systems in ways that improve benchmark results but provide little or no benefit to the end user.

In the field of computer file systems, we often find ourselves at the “worse” end of this spectrum. There are few standard benchmarks, and those that exist have significant flaws. A recent survey of research papers published between 1991 and 1996 showed that file system researchers seldom used standard benchmarks [Small97]. The most

commonly used benchmark in this study, the Andrew Benchmark [Howard88] was frequently mis-used.

A fundamental problem with the current generation of file system benchmarks is that they fail to take into account the fact that a file system's performance can vary depending on the workload running on it. Many benchmarks attempt to reduce file system performance to a single number, producing a simplistic one-dimensional ordering of the systems being tested. Although this may be useful for marketing literature, the performance of file systems in the real world is more complicated. Different workloads place different demands on the file system, and can result in different behavior from the underlying system. A file system that provides superior performance for a web server may have inferior performance when running a software development workload.

In this dissertation I demonstrate that the "one size fits all" approach of current file system benchmarks does not accurately predict the performance of different workloads on different file systems. I then present a new benchmarking methodology that not only predicts file system performance in the context of a specified workload, but also allows researchers and developers to isolate the areas of file system performance that present the greatest bottlenecks for particular workloads.

1.1 Dissertation Overview

Because different types of file system workloads can have widely different characteristics, a benchmark that does not take into account the target workload for a file system may be a poor predictor of how a particular workload will perform on that file system. In Chapter 2, I survey a range of common workloads that a file system may encounter and examine some of the important differences between them. I also discuss existing benchmarks. This chapter also provides background information on the architecture and implementation of the file systems that I use throughout this dissertation.

There are two major problems with existing file system benchmarks. The first problem is that they are typically conducted on file systems that are empty—a state rarely seen by real users. The second problem is that most file system benchmarks do not answer the question that most users ask, "How will *my* workload perform on this file system?"

In Chapter 3, I address the first problem by presenting a technique called *file system aging*. Before benchmarking a file system, I use an artificial workload (generated from real file system usage patterns) to *age* a file system. An aging workload can reproduce the effect of months, or even years, of file system activity. Applying the aging workload to a file system prior to running benchmarks creates a benchmarking environment representative of the file system state typically seen in the real world. By using a fixed aging workload, a researcher can age file systems in a reproducible manner. Similarly, by applying the same workload to different file systems, we can see how differences in file system architecture affect long term file system behavior. Because the aging workload is generated from traces and snapshots of a real file system, aging workloads representative of different types of file system activity can be created using data collected from different file system.

To address the second problem, predicting the performance of a particular workload on a particular file system, I have developed a benchmarking framework called *HBench-FS*, which separates the evaluation of the test file system from the evaluation of the workload. In Chapter 4, I describe this scheme, in which a vendor would run a series of *microbenchmarks* on their file system. The resulting *microbenchmark vector* (or *system vector*) provides quantitative measurements for a wide range of simple file system operations, such as creating or reading a file. Potential customers would characterize their workload by tracing it. HBench-FS takes a file system trace and the system vector as inputs and combine them to predict the latency of each operation in the trace. By summing the predicted latencies for all of the operations in a trace, it can accurately predict the relative performance of the workload on different file systems.

Separating the workload from the file system evaluation provides several important benefits. It allows researchers and developers to explore "what if?" scenarios.

Before attempting to optimize a piece of the file system, a researcher can easily determine how much an application would benefit from the contemplated improvement by modifying the system vector and using the new vector to analyze the workload. Also, by separating the file system evaluation from the workload analysis, it should be possible for users to evaluate products from different vendors without having access to the hardware in question. Unlike the current situation, users would be able to perform this comparison in the context of their own workload(s), rather than by comparing a single benchmark number provided by the different vendors.

Having performance predictions on a per-operation basis allows several useful types of performance analysis. The user may, for example, know that certain operations are more critical than others (perhaps they occur when the end-user is waiting for a response) and can look at the performance of only those operations. Similarly, a researcher can analyze the performance of a workload in terms of where the time is going, determining what types of operations have the longest latencies or consume the most file system time.

Chapter 5 provides a survey of related work from several areas of computer science. In this chapter, I describe existing techniques for file system benchmarking, other tools for analyzing system behavior in the context of a particular application or workload, and application-specific benchmarking tools and techniques used in other areas of computer science.

Finally, I summarize my results and conclusions in Chapter 6.

1.2 Contributions

This dissertation makes the following contributions:

- I demonstrate that benchmarking empty file systems is, at best, inaccurate, and at worst misleading. I also show that a simple technique for artificially aging file systems provides more accurate and meaningful results than the traditional approach of benchmarking empty file systems.
- I show that different workloads place different demands on file systems, and that a simple, one-dimensional performance evaluation is inadequate

to predict how a particular application will perform on a specific file system architecture.

- I present a suite of file system benchmarking tools, *HBench-FS*, that analyze the performance of a file system in the context of a specific workload of interest. HBench-FS provides accurate predictions of the relative performance of different file systems executing the same workload. It also provides fine-grained performance information that can be used by either file system or application architects to locate and eliminate performance bottlenecks.

Chapter 2

Background

In this chapter, I provide background information about the architecture and implementation of the file systems that I use throughout this dissertation. I discuss the techniques for collecting data about file system workloads, and survey a range of common workloads a file system may encounter, examining some of the important differences between them. I also provide a brief overview of traditional file system benchmarks.

2.1 File System Architecture

This section provides an overview of file system design and implementation, focusing on the aspects of file system architecture that have the largest impact on performance. The following discussion focuses on the Berkeley Fast File System (FFS), which is the native file system used by the various BSD implementations of the UNIX operating system. This is a mature and well-optimized file system that has influenced the design of several other commonly used file system architectures, including Linux's *ext2fs* [Beck98]. It is also the file system that I use for most of the examples and demonstrations throughout this dissertation.

2.1.1 General File System Architecture

Computer systems typically provide storage in the form of files. The file system implements this storage abstraction on top of the storage peripherals attached to the host computer—typically one or more hard disks. The file system also exports an interface, allowing clients to read, write, and manipulate files. On a traditional desktop system, the file system's clients are user applications executing on the same machine. Network file systems, such as NFS [Sandberg85], allow clients on remote machines to share access to files on a server machine.

A file system consists of both the persistent on-disk data and the software that provides access to this data. The on-disk representation of a file system is essentially a large data structure. The file system software implements the various algorithms for manipulating the on-disk data. It is possible to have completely different software implementations for the same on-disk file system. This allows developers to add new features or performance enhancements to existing file systems without requiring that users re-initialize their on-disk file systems. It also allows completely different implementations for the same on-disk file systems. Many UNIX systems, for example, provide support for MS-DOS file systems that may be co-resident on the local disk [Forin94].

In addition to file data, a file system also stores a variety of internal data structures on the disk. This *meta-data* provides the information the file system needs to access and manipulate its file data. The meta-data includes global file system information, such as which disk blocks are allocated, as well as per-file information, such as the location of each file's data on the disk and each file's name. Modern file systems use *directories* to provide a hierarchical namespace. A directory is an object that contains the names of other files and directories, along with internal pointers that the file system can use to find the named objects. Thus, instead of providing a flat namespace, containing the names of all of the files on the system, directories allow file systems to organize files into a tree-like, hierarchy.[1] Directories are often implemented as a special

1. Most file systems provide support for *links* (also known as *shortcuts* or *aliases*), which allow multiple names to point to the same file. On such systems the namespace forms a directed graph, rather than a simple tree.

type of file that supports special operations and can only be manipulated by the operating system.

2.1.2 UNIX File System Architecture

UNIX-like systems implement files as an ordered stream of bytes. The basic unit of data transfer between an application and a file is a single byte. For efficiency, applications usually read or write larger units of data, but the file system imposes no such requirement. Applications may read data from any location within a file. The default behavior is for the file system to keep track of the *offset* of the last byte transferred to or from the file. Successive I/O requests start at the subsequent byte within the file. When writing data, an application may either overwrite existing data within the file or append it to the end of the file. The file system does not provide a mechanism for inserting or deleting data in the middle of a file. The only way to accomplish these actions is to rewrite the file contents after the point of insertion or deletion.[2]

UNIX systems allow support for multiple file system implementations on the same system. This allows a single system to access files on native file systems (e.g., FFS), on DOS or Windows partitions on the same machine, as well as on remote file systems on other machines. UNIX provides support for multiple file system implementations via a *Virtual File System* (VFS) interface [Kleiman86]. VFS provides a uniform interface to all file systems supported by the UNIX kernel. By conforming to this interface, developers can create new file systems and easily integrate them into the kernel. This is, in essence, an object-oriented approach to implementing files and file systems within the UNIX kernel. The kernel dispatches file system operations to the appropriate file system implementation based on pointers stored in its per-file (and per-file-system) data structures.

UNIX systems provide a number of services that are used by most file system implementations. Because the disk media on which file systems reside are typically

---

2. This is in contrast to *record-oriented* file systems, which require all file I/O to be performed on one or more records. Record-oriented file systems may support only a single record size per file, or may allow for variable sized records. Some record-oriented file systems also allow for insertion or deletion of records at arbitrary points within the file.

several orders of magnitude slower than main memory, the kernel attempts to cache frequently used data in memory. This allows the system to avoid repeated requests to the disk when it receives multiple requests for the same data.

The primary cache of file data, sometimes called the *buffer cache*, contains copies of recently accessed data blocks. The operating system typically replaces cache blocks in an LRU manner. To optimize performance, and to avoid making applications block on the disk unnecessarily, most file systems attempt to overlap I/O requests with application computation. By prefetching blocks from the disk before applications request them, the system tries to avoid forcing applications to stall while waiting for data from the disk. Similarly, the operating system may write dirty data from the buffer cache to disk asynchronously, so that write requests can return to the application without blocking on the disk.

On many systems, the size of the buffer cache is fixed at boot time. Other systems, such as Solaris, Sprite, and NetBSD, allow the amount of memory for caching file data to grow and shrink according to the relative demands of file system and virtual memory activity [Gingell87][Nelson88][Cranor99].

Although the raw data blocks containing file system meta-data are usually stored in the buffer cache along with regular file data blocks, the kernel also provides separate caches for frequently accessed types of meta-data. A name cache, sometimes called a *directory name lookaside cache* (DNLC), stores the results of recent name lookup operations, and an attribute cache (or *vnode cache*) stores the attributes and meta-data for recently accessed files. These caches avoid the overhead of repeatedly translating the raw meta-data and provide fast access to their contents even after the corresponding meta-data blocks have been evicted from the buffer cache.

When a file system can not satisfy requests using the data in the various caches. It reads and writes data to the underlying disk system. The operating system provides a simple block-oriented interface to disk devices. The file system performs all I/O operations in multiples of a fixed block size. The minimum block size is determined by the disk's sector size (the smallest possible I/O size), which is typically 512 bytes, but

many file systems use larger block sizes because of the performance benefits that come from issuing large sequential I/O requests to the disk.

Older file systems were often designed to exploit the geometry of the underlying disk when they allocated space to files. Current trends in disk and storage technology make this impossible for current file systems. In contrast to older disk drives, which divided all tracks into a constant number of sectors, modern disks maximize the usable storage space by increasing the number of sectors per track with the distance of the track from the disk's spindle [VanMeter97]. While the use of such *zoned constant angular velocity* disks makes it difficult for a file system to determine and exploit disk geometry, the real problem comes from large scale storage systems such as Hewlett-Packard's AutoRAID, which presents a standard SCSI disk interface to the host system, but actually stores data in two different formats on multiple internal disks [Wilkes95].

Without any reliable knowledge of the geometry and performance characteristics of the underlying storage media, the only behavior that contemporary file systems can rely on is that sequentially numbered disk blocks will be organized in such a way to optimize sequential I/O to these blocks. On a single disk, sequentially numbered blocks typically correspond to adjacent blocks in the same track. In general, this means that the optimal file layout is to allocate logically sequential file blocks to physically sequential disk blocks.

2.1.3 Berkeley Fast File System

In the early 1980s, Kirk McKusick and his colleagues at the Computer Systems Research Group at Berkeley developed the Berkeley Fast File System (FFS) [McKusick84]. They released it as part of the 4.2 Berkeley Software Distribution of UNIX (BSD). Since then, it has become the de facto standard file system for BSD-derived UNIX implementations, including FreeBSD, BSD/OS, Solaris, and Digital UNIX.

As FFS is the file system I use for most of the research in this dissertation, this section provides an overview of its design and implementation. This section also discusses the evolution of the FFS architecture since its initial introduction. A more

complete description of FFS is available in *The Design and Implementation of the 4.4BSD Operating System* [McKusick96].

2.1.3.1 Fast File System Architecture

FFS places a *superblock* at a fixed location at the beginning of its disk partition. This block summarizes the state of the file system and includes global file system information such as the file system block size, the number of blocks on the disk partition, various parameters governing the file layout policy, and pointers to the on-disk locations of other metadata structures.

Each file on FFS is described by an on-disk data structure called an *inode* (or *index-node*). This data structure contains information about the file's attributes and pointers to the blocks containing the file's data. The attribute information includes the file's owner and group, its access permissions, its size (in bytes), and the times the file was last accessed and modified. Because the inode is a fixed-size data structure (128 bytes), there is a limited amount of space to list the disk blocks allocated to a file. To allow arbitrarily large files, FFS uses an indirect addressing scheme (illustrated in Figure 2.1), similar to that used in the original UNIX file system [Ritchie74]. The inode itself contains pointers to fifteen disk blocks. The first twelve of these pointers contain the addresses of the first twelve blocks of a file. For larger files, FFS uses the remaining block pointers in the inode to point to *indirect blocks*. The first of these points to a *singly indirect block* containing pointers to additional file data blocks. The next points to a *doubly indirect block*, which contains pointers to additional singly indirect blocks, which in turn hold pointers to file data blocks. Finally the inode has a pointer to a *triply indirect block*, containing pointers to additional double indirect blocks. On a file system using the default block size of eight kilobytes, this scheme allows file sizes of over 64 terabytes. FFS only allocates indirect blocks on demand, as a file grows large enough to require them.

FFS stores directories in the same manner as files. FFS sets a flag bit in the inode to indicate that the file should be interpreted as a directory instead of as a regular file. For each file in the directory, the on-disk directory file contains the file name and the inode

number for that file. FFS always stores the inode for the root directory in a fixed location (inode number 2).

FFS divides the underlying disk partition into allocation pools called *cylinder groups*. In the original FFS implementation, a cylinder group corresponded to some number of consecutive cylinders on the physical disk. Since it is difficult to determine the cylinder size on modern disks, current FFS implementations simply allocate a fixed number of blocks to each cylinder group. Most of the blocks in each cylinder group are used as data blocks. FFS can allocate them to any file, either as data blocks or indirect blocks. A fixed set of blocks in each cylinder group hold inodes, with multiple inodes stored in each block. As both the number of cylinder groups and the number of inodes per cylinder group are fixed when the file system is initialized, the total number of inodes

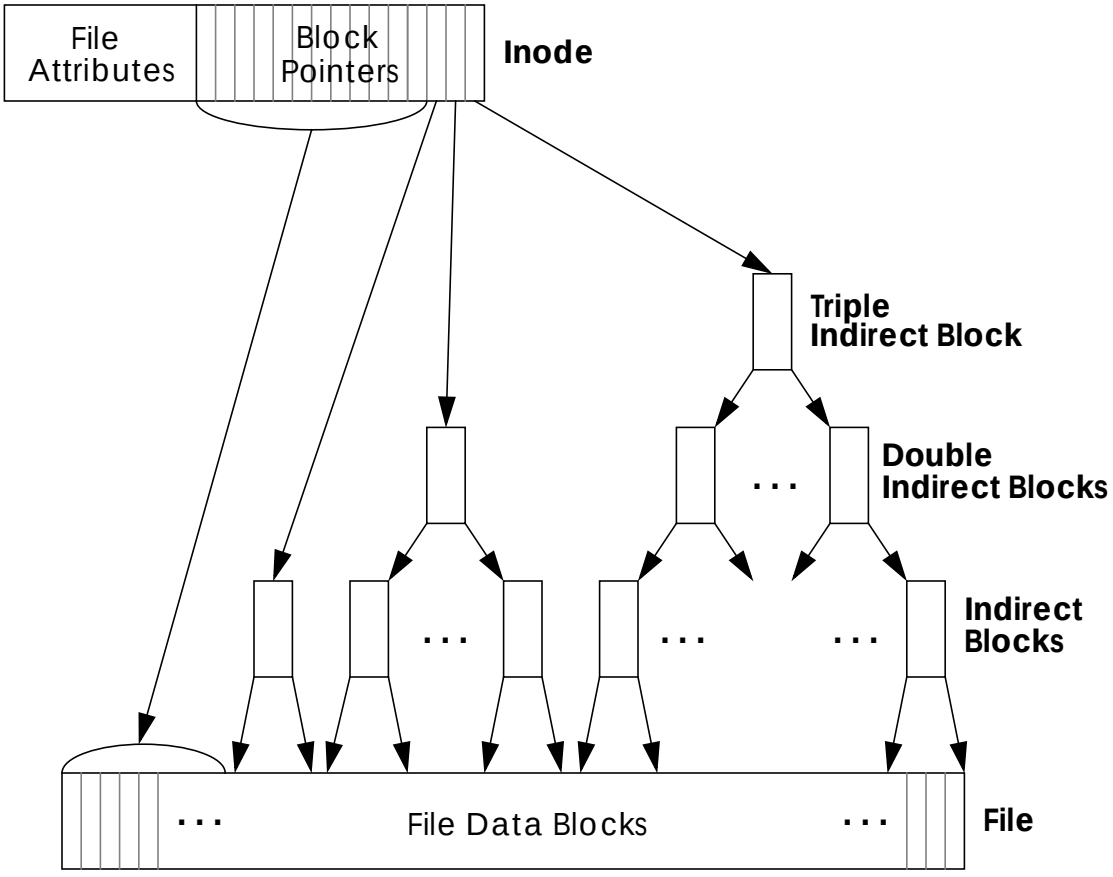

Figure 2.1. Sample FFS Block Layout. This figure depicts a file (at the bottom), its inode (at the top), and the indirect blocks the inode uses to list the file's data blocks. The inode contains twelve *direct block pointers*, which point to the first twelve data blocks of the file. The inode also contains three *indirect block pointers*, which point to a single, double, and triple indirect block. In addition to block pointers, the inode also contains the file's attributes.

represents the maximum number of files (or directories) that can be created on the file system. Each cylinder group also contains bookkeeping information, including bitmaps indicating which blocks and inodes are allocated, and a redundant copy of the superblock, which can be used for disaster recovery.

FFS uses cylinder groups as allocation pools, and attempts to exploit expected patterns of locality of reference by collocating related items in the same cylinder group. For example, FFS tries to allocate files in the same cylinder group as their directory (and hence in the same cylinder group as all other files in the same directory).

An important issue for all file systems is maintaining file system integrity in the face of a possible system failure. This is especially important for file system operations that modify more than one piece of data or meta-data, such as file creation and deletion. When FFS creates a new file, it writes two pieces of meta-data; it initializes the file's inode and writes the new file name in the appropriate directory, along with a pointer to the inode. In a naive implementation, a variety of states would be possible if the system crashed just after creating the file, depending on which (if any) of these modifications reached disk. In one scenario, the new directory entry might be written to disk before the system failure, but the newly initialized inode might not be. After rebooting, the file system would have a directory entry that referenced an invalid inode. This situation could result in a variety of unexpected behaviors.

Traditionally, FFS has taken two steps to avoid this sort of problem. FFS requires that when a file system operation modifies more than one piece of meta-data, the meta-data modifications must be written to disk in a predefined order.[3] This allows a consistency checking program (*fsck*) to scan the file system after a crash and detect and correct problems such as the one described above. FFS meets this requirement by synchronously writing each piece of meta-data. While this is sufficient to ensure file system integrity, it also introduces a large performance penalty. Any file system operation that requires synchronous meta-data updates will complete at disk speeds, rather than at

3. FFS only attempts to guarantee the integrity of file system meta-data after a system failure. User file data may be corrupted.

CPU or memory speeds. This problem has recently been addressed in FFS by the use of Soft Updates (see Section 2.1.3.2.2).

2.1.3.2 Fast File System Evolution

Over the years since its introduction, the Fast File System has benefited from a variety of optimizations and other improvements. In the following sections I summarize a few of the most important of these improvements, focusing on those that have been included in the main-stream FFS implementations provided with the various BSD UNIX kernels. Many other interesting and potentially useful optimizations have been proposed in the research literature.

2.1.3.2.1 Clustered I/O

The original implementation of the fast file system performed all I/O one block at a time. Thus, if an application requested a 64 kilobyte read, a file system with an eight kilobyte block size would generate eight separate disk requests, each individual block. In the likely event that these blocks were not contiguous on the disk, the system would also incur the overhead of seek latencies and rotational delays as the disk head moved from one block to the next.[4] McVoy and Kleiman addressed this problem by implementing *clustered I/O* in the SunOS version of FFS [McVoy91]. (Margo Seltzer and her colleagues added clustered I/O to the BSD UNIX version of FFS [Seltzer93].) They modified FFS to allow it to issue disk requests in multiples of the block size, up to the maximum *cluster size*. This value was typically configured to be 64 kilobytes, the maximum SCSI request size.

Although clustered I/O allowed FFS to issue disk requests spanning multiple blocks, FFS does not offer any guarantees about the size of the clusters it allocates on the disk. It attempts to allocate maximally sized clusters, but actual cluster sizes may be smaller, depending on the availability of contiguous free space.

---

4. Prior to the introduction of clustered I/O, FFS was typically tuned so that it would attempt to allocate every other block on the disk to a file (i.e., it used a disk *interleave* of one). Thus, in the best case, throughput to a single file was typically half of the raw throughput available from the underlying disk [McVoy91].

With the clustered I/O enhancement, FFS usually reads not only the block(s) requested by an application, but also the rest of the cluster containing the final block of the application's request. This eliminates the need for an additional disk request in the likely case that the application continues reading the file sequentially.[5] Similarly, FFS does not queue writes to the disk until the application has written enough data to fill a cluster.

A subsequent examination of the on-disk file layout showed that FFS frequently allocated small clusters to new files even when large extents of contiguous free space were available on the disk [Seltzer95]. In response to this problem, Kirk McKusick implemented a more sophisticated disk allocation algorithm, which dynamically reallocated data to new locations on disk in order to achieve larger cluster sizes. As this reallocation only occurs when the relevant data is dirty and in the buffer cache, the algorithm does not introduce any additional disk I/O. Evaluating the benefits of this new algorithm led me to develop the file system aging techniques discussed in Chapter 3 [Smith96].

The DEMOS operating system was an early example of the use of clustered I/O. DEMOS ran on the Cray-1 computer at Los Alamos Scientific Laboratory and needed to provide high file throughput [Powell77]. Kent Peacock introduced file clustering to UNIX, adding it to the System V file system [Peacock88].

2.1.3.2.2 Soft Updates

One of the most noteworthy enhancements to FFS was the recent introduction of Soft Updates as a technique to avoid the synchronous meta-data update problem described in Section 2.1.3.1. Soft Updates meets the FFS requirement that meta-data updates be written to disk in a fixed order by maintaining detailed information about the relationships between cached items of meta-data. Soft Updates uses this *dependency information* to guarantee that the individual pieces of meta-data are written in the correct order. This allows FFS to perform these meta-data updates asynchronously, relying on Soft Updates to

5. In some circumstances, FFS may prefetch an additional cluster of data beyond the end of the cluster from which the user is currently reading.

ensure that the ordering requirements are met. Using Soft Updates eliminates the disk as a bottleneck in meta-data intensive workloads, which can run up to twenty times faster than on FFS without Soft Updates [Ganger00]. Soft Updates is an alternative to the more common approach of using a *journaling* file system to eliminate synchronous meta-data updates [Chang90][Seltzer00a]

2.2 File System Workloads

One of the primary goals of file system design is to hide the slow speed of the disk using techniques such as caching, prefetching, and delayed write-back. Even when the file system must issue synchronous requests to the disk, file system designers attempt to minimize their latency by collocating related data on the same region of the disk. All of these optimizations incorporate assumptions about the behavior and file access patterns of user-level applications. Common prefetching policies, for example, assume that most files are read sequentially. In order to better understand how file systems are used, many researchers have studied the file access patterns of different workload and different computing environments.

In 1985 John Ousterhout and his colleagues at the University of California at Berkeley published a landmark study of file system usage patterns in the 4.2 BSD UNIX Fast File System [Ousterhout85]. As this study was conducted on a server in the Berkeley computer science department, the results document the usage patterns seen in that environment. The years since then have seen the publication of a variety of similar studies. Later studies have examined how file system usage has changed since initial BSD study. Other researchers have presented data on file system usage in different computing environments, such as VAX/VMS or Windows NT, and under different types of workloads, such as scientific computing or web servers.

In this section I provide an overview of these file system usage studies, with an emphasis on how file system usage varies under different workloads.

2.2.1 Methodology

There are two major techniques that researchers have used to collect data about file system usage patterns, *snapshots* and *traces*. A snapshot is a static description of file system state at a single instant in time. A snapshot is typically collected by a program that scans the file system (or the corresponding disk) and collects the desired information. In contrast, a file system trace describes file system usage rather than file system state. A trace is a dynamically collected record of the operations issued to the file system over a period of time.[6]

Each of these data collection techniques offers advantages and disadvantages. Snapshots suffer from the fact that they provide no information about how often or in what manner the file system is accessed. Looking at two snapshots collected from the same file system at different times, it is possible to infer some of the file system activity that must have occurred between the snapshots, but it is impossible to infer all of it. For example, if a file's last access time changed between the snapshots, we can infer that the file was accessed at the specified time. But it is impossible to know what other times (if any) the file was accessed. File system traces can provide complete information about the types of accesses made to the file system, but provide no information about the state of files that don't appear in the trace. For example, given a file system trace, we cannot tell what percentage of the files on the file system were accessed during trace.

A more complete technique for understanding file system usage is to combine a file system trace with a snapshot collected at the beginning of the trace period. The snapshot allows the trace to be interpreted in the context of the file system state at the time the trace was collected. For example, a trace might record only a numerical identifier for each file accessed, and the snapshot might provide the information needed to map these identifiers to the names of the corresponding files. With both a snapshot and a trace

---

6. Other researchers have emphasized the difference between static and dynamic data collection in snapshots and traces, respectively [Bennet91][Bozman91]. While this is an accurate distinction, I believe that it is more important to focus on the fact that snapshots and traces contain fundamentally different types of data. Several researchers, for example, have conducted studies based on a series of snapshots taken every few minutes on a file system [Bennet91][Chiang93]. This is dynamic data, but it conveys different information than a trace does.

it is possible to model the state changes each trace operation causes, providing an accurate view of the file system state at any point in the trace.

The choice of what data to include in a file system snapshot or trace is up to the researcher collecting the data. Many contemporary systems provide utilities that can be used to collect snapshots or traces. Tools for listing files or performing backups can be used to generate snapshots. Various system monitoring and profiling tools can be used to generate traces (e.g., *ktrace* in BSD UNIX systems, or the Linux Trace Toolkit [Yaghmour00]). In many studies, researchers have built their own tools, or instrumented their systems by hand, often because they needed to customize the type(s) of data collected.

In either case, the researcher is typically constrained both by the types of data available and by the amount of time and space available to collect and store the data. A common concern when collecting file system traces is that the operations performed by the tracing tools may interfere with the workload being traced. If the tracing system performs too many file system operations, the resulting trace might be dominated by the tracing system itself. Similarly, if the tracing infrastructure uses too many system resources, overall system performance may degrade to the point where the system is no longer useful. To minimize this interference, researchers often restrict the types (and therefore the amount) of data they collect.

Over time, faster computers and larger storage systems have allowed researchers to collect increasingly detailed file system data. The types of data included in snapshots and traces have increased, as have the durations of traces. For example, in their 1985 study of the BSD Fast File System, Ousterhout and his colleagues examined three traces, each of which recorded seven types of file system operation over approximately three days [Ousterhout85]. In contrast, for their 1998 study, Roselli and Anderson collected traces of 55 different types of file system operation over time periods of one to four months [Roselli98].

A complete discussion of the numerous studies of file system usage patterns is beyond the scope of this thesis. Mummert and Satyanarayanan provide a thorough but

brief survey of file system usage studies up to 1996 [Mummert96]. The important recent studies include snapshot and trace-based studies of Windows NT file system usage by Douceur and Vogels [Douceur99][Vogels99], and Drew Roselli's examination of long-term file access patterns in both HP-UX and Windows NT [Roselli98][Roselli00].

2.2.2 Workload Differences

Most studies of file system usage patterns have tried to make generalizations about access patterns, rather than examining the differences between different types of workloads. In fact, many of these studies examined only a single type of workload. A few studies, however, have collected data from different types of workloads. In addition, some studies have examined differences in file system workloads across different operating system environments (i.e., Windows NT and UNIX). Other studies have reported on changes in workload patterns over time.

All of these differences in file system access patterns motivate the need for application-specific benchmarking techniques. If benchmarking technology is to allow users to evaluate system performance in the context of their own workload, then it must also adapt to the differences in access patterns that come with changes of operating system or changes of time. In other words, an application-specific benchmarking framework must make accurate predictions about the performance of application workloads today, and must continue to be able to make accurate predictions in the future. Likewise it should allow researchers and users to evaluate the performance impact of switching from one operating system platform to another.

The wide range of differences in file access patterns seen in these studies indicates the need for workload-specific benchmarking techniques. Seemingly small differences in access patterns can have a substantial impact of file system performance and on file system design. Two similar workloads, for example, with different working set sizes might see considerably different performance on a system if one workload's working set fits in the buffer cache and the other does not.

In this section I survey the studies of file system usage patterns that have shed some light on these differences.

2.2.2.1 Differences Between Workloads

Although a variety of file system usage studies have collected data from different workload environments, only a few of these studies have provided substantive information about how file system access patterns have differed between the environments studied.

K.K. Ramakrishnan and his colleagues collected file system usage data at eight different production environments using VAX/VMS computer systems [Ramakrishnan92]. These sites included traditional software development, office management, transaction processing, and batch processing workloads. The authors were primarily interested in finding the common features across the different workloads they examined. They found many common features between the VAX/VMS environment and the UNIX-like environments examined by Ousterhout and Baker—most files are only open a short time, most files are small, most files are read/written sequentially. The authors also observed several significant differences in the file access patterns of the different workloads.

- Although the median file sizes were similar for all workloads, the average file sizes for the transaction processing and batch processing workloads were 5 – 10 times larger than for the other workloads, indicating that the large files in these workloads were larger than in the other workloads. The transaction processing and batch processing workloads also had fewer total files on their file systems than the other workloads in the study.
- Most of the workloads only accessed 10-30% of their total data (in bytes) during a day of activity. The transaction processing workloads accessed a much larger portion of their file system data, 59 – 86%.
- The batch processing workload had the highest ratio of data bytes read to data bytes written, followed by the transaction processing workloads.
- While most of the workloads accessed only a small percentage of the total files in their file systems (less than 25%), the workload from an order-entry database accessed 82.2% of its files in a twelve hour period.

In general, Ramakrishnan and his colleagues found that although their traces displayed similar probability density functions for many access characteristics (e.g., number of files accessed per day), the effect of the outliers was most significant in the transaction processing workloads. For example, in their program development trace, the most active 1.1% of the files account for 39% of the file opens, compared with the airline reservation database, where the most active 2.6% of files account for 84% of the file opens.

In a different study, Smith and Seltzer reported on a snapshot-based study of nearly fifty file systems on servers at the Harvard Division of Applied Science. The servers were all running SunOS 4.1.3. The authors collected snapshots of these file systems on a nightly basis for approximately ten months. The snapshots included extensive information about the physical layout of the files on disk, allowing the authors to examine the variations in file fragmentation across the file systems, and how file fragmentation changed over time.

Smith and Seltzer found wide variation in the amount of file fragmentation on different file systems. This reflected the usage patterns of the file systems. A news spool file system, which stored usenet news articles and experienced the frequent creation and deletion of small files, was highly fragmented. In contrast, a file system containing system binaries, had almost no fragmentation, reflecting the fact that there was very little turn over on the file system. Smith and Seltzer also found that the location of free space varied across the file systems in their study. Since new files must be allocated from the existing pool of free space, the location and fragmentation of free space on a file system restricts the layout (and hence the performance) of newly created files.

In 1994, Geoffrey Kuenning and his colleagues at UCLA published a trace-based study of file access patterns in commercial environment running DOS [Kuenning94]. Their goal was to evaluate algorithms for predicting what files a mobile computer might need while disconnected from the network so those files could be prefetched to the mobile computer’s disk. The authors divided their traces into three broad categories, personal productivity (i.e., e-mail, project planning, and calendar applications), programming, and commercial (i.e., a commercial accounting package). Kuenning and

his associates found that the commercial environment accessed more data per day. The total sizes of the files accessed in a single day was 18.2 megabytes in the commercial environment, compared with 0.3 – 1.0 megabyte in the other environments. The authors also examined the different sets of traces for *conflicts*—events where two or more users write to the same file within a short time span, a potential problem when one of the users is disconnected from the file server. They found that the commercial environment had a higher conflict rate (4.3 per user per day) than the other environments (0 – 1.2 per user per day).

Drew Roselli and her colleagues collected long term file system traces from several computing environments during the late 1990s [Roselli98][Roselli00]. These environments included UNIX clusters supporting instructional and research activities at the University of California, Berkeley, as well as a web server serving images from a large database library. They also traced eight desktop workstations running office productivity applications under Windows NT.

Roselli and her colleagues found that the research and instructional workloads were similar in many respects. This is not surprising since the users in both environments were performing similar tasks—programming and text editing. Despite these similarities, the authors found enough differences between these workloads to demonstrate the need for workload-specific benchmarking tools. In particular, the locality of reference and the working set sizes were significantly different in the two environments. Cache simulations showed that for comparable cache sizes, the instructional workload generated 5 – 10 times as much read traffic and three times as much write traffic to the disk. This would be important in evaluating the performance of a system for these workloads. A hardware configuration that provided sufficient disk bandwidth for the less demanding research workload might collapse under the heavier demands of the instructional workload.

Roselli and her colleagues also observed several significant differences between the web server workload and the more conventional research and instructional workloads.

- Unlike the instructional and research workloads, which showed a steady improvement in cache hit rate as the cache size was increased, the web server workload showed little improvement in cache hit rate until the cache size exceeded 16 megabytes. At this point, the hit rate improved dramatically. This suggests that although the web server workload had a well defined working set, there was little locality of reference within subsets of this working set.
- All of the workloads had more read traffic than write traffic. In the instructional and research workloads, however, large caches (256 megabytes) could absorb enough of the read traffic so writes would dominate the disk requests. In contrast, even with a 256 megabyte cache, the web workload would still generate twice as much read traffic as write traffic. This suggests that the web workload would benefit from a file system that was optimized for reads, whereas the instructional and research workloads would gain more benefit from a write-optimized file system.
- In the instructional and research workloads, the files that were accessed most frequently were written often and read rarely. This phenomenon did not occur in the web workload, where there was little write traffic.

Several studies have examined the file access patterns of scientific computing applications in both parallel and super computing environments. These studies have found a variety of significant differences between these scientific workloads and traditional time-sharing workloads.

Miller and Katz traced application file system accesses on a vector-processing supercomputer (a Cray Y-MP 8/8128 at NASA Ames) [Miller91]. The super computing applications that they studied performed repeated sequential accesses to large files. Since these files were typically too large to be cached in memory, a key characteristic in determining file system behavior was the ability of the system to stream large sequential files into and out of memory.

Purakayastha and his colleagues collected file access traces from a parallel supercomputer (a 512 node CM-5 at the National Center for Super computing Applications) [Purakayastha95]. Like Miller and Katz, they found that file sizes were larger than in traditional time-sharing workloads; more than a third of the files accessed were larger than ten megabytes, and some accesses were to files larger than ten gigabytes. Unlike the vector-computer workloads, however, Purakayastha and his colleagues found that most I/O requests were small (90% of them were less than one kilobyte). Furthermore, many requests were for sequential, but not consecutive blocks with the files. This reflects the partitioning of data sets across the multiple processors in the CM-5. Kotz and Nieuwejaar performed a similar study of a 128 node iPSC/860 at NASA Ames, and saw similar results [Kotz94b].

2.2.2.2 Differences Over Time

Ousterhout's 1985 study of file access patterns in the 4.2BSD UNIX Fast File System has become a touchstone to which many subsequent file system studies have referred. In particular, several subsequent studies explicitly compared their results with Ousterhout's (and with each other) to examine the changes in file access patterns over time. Since all of these studies examined traditional academic workloads in similar operating system environments, the authors attributed most of the differences between them to the changes brought by faster computers, larger disks, and new applications.

In 1991, Mary Baker and her colleagues performed a trace-based study of file access patterns in the Sprite distributed operating system [Baker91]. They explicitly compared their results to the earlier BSD UNIX study, and found that although most aspects of file system usage had remained the same, file throughput had increased by a factor of twenty, and that it had become much burstier. A primary cause for this change was that the sizes of the largest files accessed in the traces had increased by an order of magnitude. Baker and her colleagues also found that file lifetimes had decreased.

More recently, Werner Vogels and Drew Roselli performed trace-based studies of file access patterns in Windows NT and HP-UX [Vogels99][Roselli00]. Both authors explicitly compared their results to the earlier Sprite and BSD studies. These studies

showed a continued increase in file throughput with burstiness commensurate with increases in the large file sizes. Both of these studies also found that although sequential access was still the dominant access pattern, there was a significant increase in random accesses.

2.2.2.3 Differences Across Operating Systems

After the publication of the trace-based study of file accesses in the 4.2BSD UNIX Fast File System [Ousterhout85], researchers performed similar studies in other operating environments. Studies of VAX/VMS and VM/CMS showed that file access patterns in those environments were similar to those Ousterhout and his colleagues observed in BSD UNIX [Biswas90][Bozman91]. More recent studies have compared Windows NT workloads to UNIX workloads and have found more significant differences.

In her study of HP-UX and Windows NT workloads, discussed above, Roselli also examined block lifetimes for the different workloads [Roselli00]. This is a measure of the time between when a data block is written, and when it is deleted, whether because the block's file was truncated or deleted, or because the block was overwritten with new data. On many systems, a significant fraction of newly written data has a short lifetime. (Approximately 35% of files in Mary Baker's Sprite-based study were deleted less than thirty seconds after they were created [Baker91].[7]). Some systems, such as Sprite [Nelson88], exploit this characteristic by holding dirty data in the buffer cache for a short interval before writing it to disk. This allows short-lived data to die in the cache and reduces disk traffic.

Unlike the HP-UX workloads she studied (and the earlier Sprite workload), Roselli found that block lifetimes in Windows NT had a bimodal distribution. While 20% of newly written blocks died within one second, few died after that; only 30% of newly written blocks died within a day. The HP-UX workloads, in contrast, showed a sharper increase in blocks deleted over time. In the instructional workload, for example, 20% of

7. Baker's original study actually showed a much higher fraction of files—up to 80%—lived for less than thirty seconds. The more conservative 35% figure comes from Roselli's study, where she uses a different method to compute file and block lifetimes [Roselli00].

new data blocks also lived less than a second, but over 80% of newly written blocks lived less than a day.

In 1999, Douceur and Bolosky published a study of files in a commercial environment running Windows [Douceur99]. They collected snapshot data from more than ten thousand file systems on machines throughout Microsoft Corporation's main campus. Although they did not study file systems from any other operating system environment, it is interesting to compare some of their results to those from other studies. Several earlier studies (and anecdotal evidence) showed that most file systems had little free space [Bennet91][Smith94]. In contrast, Douceur and Bolosky found that on average file systems were only half full. While this difference may be due to inherent differences between Windows and UNIX environments, including the fact that Douceur and Bolosky were studying desktop workstations, in contrast to earlier studies, which examined centralized servers, another likely explanation is the rapid increase in disk capacities over recent years.

2.2.2.4 Netnews Workloads

Usenet news servers are another source of a unique file system workload. This workload stresses conventional file system architectures so severely, that there have been proposals for file system optimizations [Zadok99] or completely new file system architectures [Christenson97][Fritchie97] specifically targeted to supporting Usenet news.

With the advent of the World Wide Web, it may seem that Netnews is an obsolete technology, but anecdotal evidence suggests that news bandwidth continues to increase, currently reaching volumes of up to twenty gigabytes per day of traffic. There is enough demand for news that most ISPs provide news service, and file server vendors offer news-specific optimizations [Manley00].

Scott Fritchie provides an excellent overview of the problems traditional file systems, such as FFS, face in supporting a Usenet news server [Fritchie97].

- News server software, such as INN [Spencer98] Henry Spencer, David Lawrence. Managing Usenet. O'Reilly & Associates, Inc. Cambridge, MA. 1998., often organizes the file system with a separate directory for each

newsgroup and individual files within those directories for each article. In high volume newsgroups, directory sizes can reach tens of thousands of files. File systems, such as FFS, that use a simple linear scan of a directory when looking for a name pay a large overhead for these lookups.

- Articles tend to arrive at servers in random order with respect to news groups. This diminishes the effectiveness of the buffer cache, as the cached contents of directories are likely to be replace before they are reused. The random newsgroup distribution also works at cross-purposes with the common file system assumption that files in the same directory are likely to be accessed together, and therefore should be collocated on the disk.
- Even though the behavior of news clients tends to display locality of reference, this is of little use to the news server. Most clients are humans. By the time they read another article in the same newsgroup, the disk heads on the server have long since been relocated to a different location.
- Because each news article is stored as a separate file, processing a newsfeed requires the creation of many small files. UNIX-style file systems, which have traditionally performed file creates synchronously (see Section 2.1.3.1), perform very poorly with this type of workload.

2.3 File System Benchmarks

Like all systems benchmarks, traditional file system benchmarks can be broadly classified into two different categories. *Microbenchmarks* measure one specific characteristic of file system behavior, such as the time to create or delete a file. *Macrobenchmarks,* in contrast, measure the performance of the file system under some pre-determined workload. A macrobenchmark may use either a real application to generate a workload (e.g., compiling and linking a large piece of software), or it may itself be a custom application that drives the file system with a synthetic workload. In isolation, neither type of benchmark fully illuminates the behavior of a file system. Macrobenchmarks can determine the system on which a particular workload will perform best, but they provide little information about the underlying causes for such performance differences. Microbenchmarks, in con-

trast, are useful for understanding the detailed performance differences between systems, (e.g., one system may provide higher I/O throughput than another). Sometimes, microbenchmarks may show that one file system out-performs another in all areas. More commonly, however, different file systems offer different advantages. In such cases, the microbenchmark results by themselves provide no insight into which systems are best suited to particular workloads.

2.3.1 File System Microbenchmarks

Four quantities are the most common targets of file system microbenchmarks:

- the time to create a file,
- the time to delete a file,
- the throughput for reading files, and
- the throughput for writing files.

Occasionally researchers use microbenchmarks to measure other quantities, such as the time to create a symbolic link or read a directory. These quantities are measured less often, because the corresponding file system operations are perceived to occur less often in real file system workloads.

Each of these quantities offers a range of parameters that researchers can vary in writing a microbenchmark. For example, file read throughput usually depends on two important parameters, whether or not the file data is in the buffer cache, and whether the test program reads the files sequentially or randomly. While researchers usually report on the behavior of their microbenchmark programs in this regard, there are a variety of other parameters that also affect benchmark results. File read throughput can also depend on the size and number of files being read, as well as on the location of the files within the directory hierarchy. Even a minor detail such as whether the files are read in the same order they were created can affect benchmark results. Researchers seldom provide full details of their microbenchmark programs. Without knowing whether two studies used similar parameters for their microbenchmark programs, it is impossible to compare microbenchmark results from different researchers.

Chris Small and his colleagues reported on this problem, not just for file system benchmarks, but for many areas of systems research, and argued for the need for standardized benchmark programs [Small97].

In addition to ad hoc microbenchmark programs explicitly designed to measure one aspect of file system performance, some researchers use standard utility programs as microbenchmarks. Such researchers choose a program that exercises a small part of file system functionality, such as file copies, or recursive directory listings.

2.3.2 File System Macrobenchmarks

The goal of macrobenchmark programs to understand how real workloads perform on a file system. Thus, many macrobenchmarks consist of executing some application with carefully specified parameters. While any program that exercises the file system may be suitable for use as a macrobenchmark, the most common such program is the compiler. Not only does the compiler read and write many files, it is also a tool that researchers frequently use. A typical compilation-based macrobenchmark consists of building and linking the operating system kernel for the system being benchmarked.

The Andrew benchmark [Howard88] is one of the most commonly used file system macrobenchmarks [Small97]. It too uses the time to build a piece of software as a metric of file system performance. Andrew, however includes several other phases, intended to mimic the behavior of software developers. These phases consist of creating a directory hierarchy for the source files, copying the source files into this hierarchy, examining each new copy (using the *stat* system call), and reading each new copy (using the *grep* utility). Other researchers have modified Andrew to run multiple concurrent Andrew workloads on the same system, attempting to mimic the effect of timesharing [Seltzer93]. Ironically, in light of its wide spread use, Andrew was originally intended for measuring the scalability of file servers in distributed file systems, not for evaluating the performance of the file system.

In addition to application-based macrobenchmarks, a variety of macrobenchmark programs use synthetic workloads. The most widely known and used of these is the SPEC SFS benchmark (based on the earlier LADDIS benchmark [Wittle93]) for measuring

NFS server performance. This benchmark creates an initial file hierarchy on an NFS server. Multiple client machines then issue random NFS requests to the server. The clients select these operations from a pre-defined distribution that was generated from traces of actual system behavior. SFS reports results in NFS operations per second.

Other less commonly used synthetic macrobenchmarks include PostMark [Katcher97] and IOStone [Park90]. Both benchmarks are similar to SFS in that they perform random operations selected from a pre-determined distribution. Diane Tang provides an overview and critique of several file system benchmarks (including IOStone and LADDIS) [Tang95].

Like SFS, most macrobenchmark programs report only a single result, either the elapsed time to execute the benchmark, or the file system throughput achieved by the benchmark.[8] By providing only a single number as the evaluation of a file system's performance, these benchmarks are essentially claiming to be able to provide a one-dimensional ordering of different file systems. As we will see in Chapter 4, this claim doesn't hold up when we examine how different file systems perform with different workloads. In fact, the most that a benchmark can do is predict the performance of a file system with a workload that is similar to the benchmark workload.

This highlights the central deficiency of existing benchmarking methodology. Most users turn to benchmarks, or benchmark results, in order to answer the question, "How will *my* workload perform on this system?" Microbenchmarks provide detailed information about file system performance, but in isolation they are seldom sufficient to answer this question. In the absence of a detailed understanding of the interactions between a workload and the file system, however, microbenchmarks provide little insight into the workload's performance on the target file system. Macrobenchmarks, in contrast, are designed to answer this very question, but the user is still left with the difficult task of determining whether publicly available macrobenchmarks are similar to his own workloads. In the following chapters, I will discuss a technique researchers can

8. The Andrew benchmark is unusual in this regard as it reports the time for each of its five phases to complete. Since the different phases exercise different parts of the file system, this provides additional information about how differences in file system architecture map to differences in workload performance.

use to make the environment in which they execute benchmarks more realistic, and new benchmarking strategies that make it possible for users to determine how well their own workload will perform on a file system of interest.

Chapter 3

File System Aging

The increasing prevalence of I/O-intensive applications, such as multi-media applications and large databases, has placed growing pressure on computer storage systems. In response to these pressures, researchers have investigated a variety of new technologies for improving file system performance and functionality. Rosenblum and Ousterhout proposed the log-structured file system (LFS) as a way to address the increasing fraction of disk traffic due to writes [Rosenblum92]. Ganger and Kaashoek proposed several techniques for reorganizing small files and meta-data objects to improve on-disk locality [Ganger97]. Researchers have also explored a variety of strategies for application-assisted prefetching and caching, exploiting application-specific knowledge of I/O patterns to better utilize I/O systems [Patterson95][Kimbrel96][Mowry96][Cao96].

In order to accurately assess the utility of any of these technologies, researchers need tools that allow them to understand the behavior of their file systems in realistic conditions. In laboratory settings, “realistic conditions” are usually simulated by the use of benchmark programs. In Section 2.3, I described the range of benchmarking techniques and some of the benchmarks currently used by file system researchers. Selecting the proper benchmark, however, is only half of the problem. To accurately characterize the performance of a file system, the benchmark itself must be executed in an environment similar to the conditions under which the system will be used in the real world.

Unfortunately, the latter requirement seems to have been widely ignored by file system researchers. Standard practice in file system research is to perform benchmarking on empty file systems, a state rarely seen in real world environments.

In this chapter, I propose a methodology for artificially *aging* a file system by simulating a long term workload on it. By aging a file system prior to running benchmarks, the resulting benchmark performance resembles that of the real file system from which the workload was generated. Just as researchers use different benchmarking programs to simulate different application workloads, they can use different aging workloads to simulate different execution environments.

In the next section, I motivate this work by describing some of the inaccuracies that arise when using the traditional approach of executing benchmarks on empty file systems. Section 3.2 describes my file system aging technology and demonstrates the it produces realistically aged file systems. In Section 3.3 I use this aging methodology to evaluate two new file layout policies for the UNIX Fast File System. Finally, in Section 3.4, I present my conclusions.

3.1 Motivation

Executing a benchmark on an empty file system fails to capture two important characteristics of file system behavior, both of which can have a substantial effect on file system performance. First, real file systems are almost never empty. This fact can have a profound effect on the performance of a file system. Most file systems optimize throughput by allocating physically contiguous disk blocks to logically sequential data, allowing the data to be read and written at near optimal speeds. On empty disks, this type of allocation is simple. On real file systems, which are typically highly utilized, contiguous allocation may be difficult (or impossible) to achieve due to fragmentation of the available free space. As a result, new files may be more fragmented on a highly utilized system, resulting in lower file system throughput.

The second problem with benchmarking an empty file system is that it is impossible to study the evolution of the file system over time. With the passage of time, the state of a file system may change. As files are created and deleted, patterns of file

fragmentation may change, as well as the relative locations of logically related objects on the disk (e.g., a file's data and meta-data). There are a variety of file system policies that may have no effect over the short term on an empty file system, but that can have a noticeable impact on file system performance over the long run. Decisions that a file system makes today (e.g., which blocks to allocate to a new file) may affect the file system for months or years into the future.

In this section, I present an example of each of these problems, demonstrating that benchmarks conducted on an empty file system can either provide misleading results, or fail to measure the effects of significant changes to the underlying file system.

3.1.1 Empty File Systems

The most common problem with benchmarking empty file systems stems from the fact that it is very difficult to measure the effects of file fragmentation on an empty disk. Because fragmentation is a fact of life in most file system designs, it is foolish to benchmark such file systems when they are empty, and have no file fragmentation. To demonstrate this effect, I ran a simple file system benchmark on both empty and full UNIX file systems. To measure the performance of a full file system, I copied an active file system from one of the departmental file servers onto a test machine[1]. After benchmarking this file system, I built an empty file system, with the same parameters, on the same disk, and measured its performance.

The benchmark program that I use throughout this chapter measures file system throughput reading and writing files of a variety of different sizes. Figure 3.1 shows the read throughput for files from 16 KB to 16 MB. Throughput on the real file system was as much as 77% lower than throughput on a comparable empty file system.

1. Rather than copying the entire file system, I only copied the file system's meta-data. The result was that the test file system had exactly the same free blocks and allocated blocks as the original file system that I copied, but not the actual data.

3.1.2 Life Time Evolution

Most file systems attempt to optimize performance by clustering logically related data on the underlying disk(s). The effectiveness of different clustering strategies may not be apparent when observing the short term behavior of the file system. Over time, however, both free and allocated space on the disk may become fragmented, affecting the ability of the file system to perform clustering. Note that this fragmentation affects not only the sequential layout of each file's data, but also the proximity of related files on the disk, and the relative locations of a file and the meta-data that describes it. In such cases, the only way to evaluate competing designs is by comparing file systems after a long period of activity.

In previous work [Smith96], I studied the effect of one such design parameter on file system performance. The 4.4BSD fast file system [McKusick84] optimizes sequential I/O performance by allocating physically contiguous *clusters* of blocks to logically sequential file data. Over the life of a file system, as free space becomes fragmented, it becomes increasingly difficult to find contiguous free space for new clusters. In

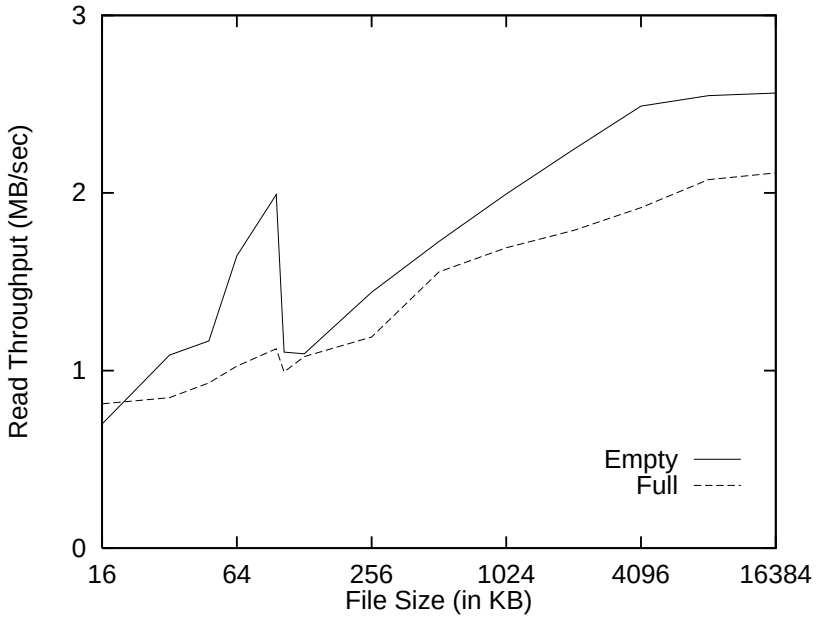

Figure 3.1. Effect of utilization on file system performance. This graph shows the read throughput for a range of file sizes on two UNIX file systems. The only difference between the file systems is the amount of free space available. One file system was empty when the benchmark was performed. The other file system was a duplicate of a seven month old file system that was 75% full. The sharp performance drop at 96 kilobytes occurs when the test files become large enough to require an indirect block. This is characteristic of all FFS file systems and is explained in detail elsewhere [Seltzer95].

comparing two different algorithms for finding and allocating free space to new files, we discovered that they provided nearly identical performance on an empty disk (see Figure 3.2A). After applying a simulated ten month workload to the two file systems, however, it became apparent that there was a substantial performance difference between file systems using the two different disk allocation policies (see Figure 3.2B).

3.2 File System Aging

As the previous section demonstrates, benchmarking empty file systems cannot provide an accurate assessment of the real-world behavior of a file system architecture. In order to get a realistic picture of file system behavior, a file system must be analyzed in realistic conditions. This means that the file system should not be empty, and should have the historical state that would be developed over many months, if not years, of operation. In order to analyze file system performance in this manner, we need to apply a methodology that allows researchers to fill a file system in a realistic manner, resulting in a file system that is similar to one that had been active in real-world conditions for an extended period of time. Analyzing file system performance in this manner presents a variety of problems that do not arise when benchmarking an empty file system:

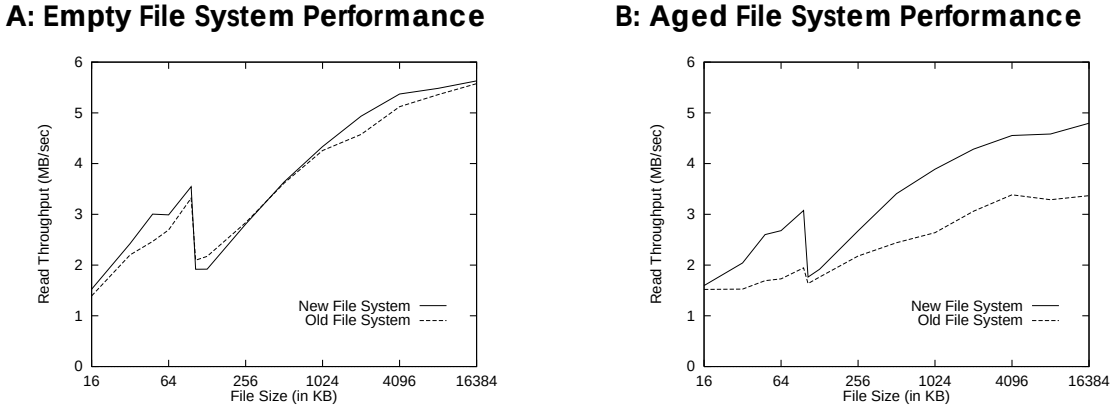

Figure 3.2. Effect of aging on file system behavior. Each of these graphs plots the read throughput for a range of file sizes on file systems using two different block allocation strategies. In the graph A, performance was measured on empty file systems. In the graph B, performance was measured after *aging* the two file systems with a simulated ten month workload. On the empty file system, the new allocation algorithm performed slightly better, but the performance of the two systems was nearly identical On the aged file systems, both file systems perform worse than in the empty case, and the new allocation algorithm provides a large improvement in read throughput. A complete discussion of this study is presented elsewhere [Smith96].

- Because different applications apply different workloads to the file system, it should be possible to simulate the effects of different file system workloads. A file system used in a traditional engineering environment for a year may behave very differently from one that has been used on a news server for a similar period of time, even if the underlying file system architectures are identical.
- The technique used to fill a file system should be reproducible, allowing scientific comparisons in a laboratory setting.
- The manner in which file systems are filled should be independent of the architecture of the underlying file system, allowing different file system implementations to be compared.

In order to study file system performance in a realistic manner, and to address the concerns listed above, I have developed a technique I call *file system aging*. I precompute an artificial workload intended to simulate the pattern of file operations that would be applied to a file system over an extended period of time. This *aging workload* consists of a sequence of records, each of which describes the creation, deletion, or modification of one file or directory[2]. By applying the same workload to different file systems, a researcher can see how differences in file system architecture affect the long term behavior of the file system. The aging workload is generated from snapshots and traces of a real file system. Aging workloads representative of different types of file system activity can be created using data collected from appropriate file systems.

Despite my desire for an architecture neutral file system aging technique, the tools I present in this chapter have several minor dependencies on the underlying file system (FFS in this case). These dependencies are discussed in Section 3.2.2.

In this section I present the technique I use to generate aging workloads, describe the program that actually applies a workload to a test file system, and then evaluate the

2. In a real file system workload, a file creation consists of multiple events, one to create the file and one or more to write data to the new file. In my aging workloads, I coalesce these separate events into a single record specifying the name and size of a new file. This is a reasonable simplification in light of the fact that most files are written in their entirety immediately after being created [Ousterhout85] [Baker91].

accuracy of an aging workload by comparing artificially aged file systems with the original file systems from which the aging workloads were generated.

3.2.1 Generating a Workload

The central problem in aging a file system is generating a realistic workload. Because a test system is likely to start with an empty disk, this workload should start with an empty file system and simulate the load on a new file system over many months or years, resulting in a file system that is mostly full. One possible method for generating this workload would be to collect extended file system traces and to age a test file system by replaying the exact set of file operations seen in the trace. Unfortunately, the time and storage space required to collect such a trace usually make this strategy impractical[3]. Instead, I have generated aging workloads from two sets of file system data that were already available. As a result, we sacrifice some realism in the workload, in exchange for greater flexibility in tuning the workload to our needs.

An aging workload is a sequence of records, each of which describes a single file system operation. At first glance, this is the same as a file system trace. The difference is that where a file system trace includes all operations performed on a file system, the aging workload only needs to include operations that can have an effect on the long term state of the file system. In particular, the operations we are interested in are those that allocate or deallocate file data or file system meta-data. The most common of these operations are file (and directory) creates and deletes. We are also interested in write operations, as they represent the primary mechanism by which file data space is allocated. As most files are written in their entirety immediately after they are created [Ousterhout85] [Baker91], we don't need to explicitly include individual write operations in the workload. Instead, it is sufficient to have the file create records include the size of the target file[4].

To generate an aging workload, I use a set of file system *snapshots* collected from a file system on a local file server. These snapshots, originally gathered for a different

3. Drew Roselli and her colleagues at Berkeley recently released a set of file system traces collected over a period of almost a year [Roselli00]. These are the first traces I know of that could potentially be used to age a file system.

research project [Seltzer95], were collected nightly from approximately fifty file systems on five different file servers over periods of time ranging from one to three years. Each snapshot describes all of the files on a file system at the time of the snapshot. For each file, the snapshot includes the file's inode number, inode change time, inode generation number, file type, file size, and a list of the disk blocks allocated to the file.

By using a sequence of snapshots of one file system, I generate an aging workload modeled on the actual activity on that file system during the period of time covered by the snapshots. Because I have snapshots from a variety of different file systems, I can generate aging workloads that are representative of different file system uses. The extended period of time covered by the file system snapshots makes it possible to build an aging workload that simulates many months of file system activity.

Generating a workload from a sequence of traces is a three step process. First, I generate operations that initialize the target file system to a state similar to the first snapshot of the original file system. Next, I create a skeleton of the workload by comparing successive pairs of snapshots and generating a workload to account for the changes on the original file system between each pair of snapshots. Finally, I flesh out the workload by adding the create and delete events for a variety of short-lived files.

The first step in creating an aging workload is to generate a sequence of file system operations that will bring the test file system into a state similar to the one represented by the first snapshot of the original file system. Because the only state I am trying to reproduce is the set of files that exist on the file system, this is a simple matter of creating each file in the initial snapshot. I sort the actual create operations based on the inode change times of the files in the snapshot in the expectation that this will be a reasonable approximation of the order in which the files were created on the original file system.

---

4. For conventional file system architectures, file and directory create and delete operations are the only records that we need to include in an aging workload in order to reproduce the long term evolution of file system state. For some experimental file system architectures, however, we would have to include other types of operations in the aging workload. Carl Staelin's Smart File System, for example, migrates file data blocks to the central cylinders of a disk based on how frequently they are accessed [Staelin91]. To properly age such a file system, the aging workload would have to include file read operations.

Next I generate the skeleton of the aging workload. By comparing the inodes listed in successive pairs of snapshots, I generate a list of the files that were created, deleted, modified, or replaced between the times of the two snapshots. The major difficulty at this stage is determining the sequence in which these actions occurred, as the snapshots do not provide sufficient information to determine the exact time at which these operations took place.

I use several heuristics to assign times to the create and delete operations generated by comparing successive snapshots. The inode change time, recorded for each file in a snapshot, indicates the last time that the file's meta-data was modified. Such modifications include the original creation of the file, and the allocation of new disk blocks to the file. As previous studies have shown that files are typically written in one burst, and are seldom modified after they are first written [Ousterhout85] [Baker91], I use the inode change time on a newly created file to approximate the time at which the file was created. When a file was deleted between two snapshots, there was no information providing hints about the time it was deleted. I randomly assign times to the file deletions that occurred between two snapshots. This was an ad hoc decision made to expedite the development of the file system aging workloads. A more careful analysis of file deletion times in real file system traces might provide a more accurate solution and improve the realism of the aging workloads.

When the same inode is listed in two successive snapshots, but with different file attributes, one of two things may have happened on the original file system; the file was either modified, or replaced. The inode generation number provides the information required to determine which of these actions actually occurred. If the generation number is the same in both snapshots then the file was modified. In this case I place a file modification operation in the aging workload, and assign it a time corresponding to the inode change time in the later snapshot. If the generation number is different between the two snapshots, then the original file must have been deleted, and a new file assigned the same inode number. In this case, I place two operations in the workload, a delete, and a

subsequent create. I determine the time of the create as described above, and place the delete immediately prior to it.

After processing the snapshots in this manner, the workload is missing an important component of real file system activity. Any file that was both created and deleted between successive snapshots will not appear in either snapshot. Trace-based file system studies have shown that most files live for less than the twenty-four hours between successive snapshots [Ousterhout85] [Baker91]. These files may have a significant effect on the state of the longer lived files on the file system.

To approximate the effect of these short-lived files, we must add additional file operations to the workload generated from the snapshots. In order to add this additional workload, we must answer two questions—what operations should we add, and where (both physically and temporally) should we add them?

To determine what file operations we should add to the aging workload, I examined the patterns of activity displayed by short-lived files in a seven day trace of NFS requests to a Network Appliance file server [Hitz94]. For each day in the trace, I made a list of the active directories, and then created a profile of the short-lived file activity in those directories. The result was 449 different profiles, each containing a list of create and delete operations on short-lived files that occurred on one day in one directory. For each day in the aging workload, I select 25 of these profiles at random and added them to the aging workload[5].

Given a day of activity from the aging workload, and a set of short-lived file profiles, I integrate the two by finding the most active directories[6] in that day of the aging workload, and randomly distributing the profiles among them. I time-shift each profile so that it coincides with the peak of activity in the directory to which it is added.

The NFS trace that I used to generate our profiles of short-lived file activity was originally collected during a study of cleaning algorithms for log-structured file systems

---

5. I actually scaled the number of short-lived file profiles that we used based on the size of the file system from which I generated the aging workload, adding one profile for every 40 MB on the original file system.
6. Since the file system snapshots do not preserve the names of the files in them, I actually used the most active cylinder groups instead of the most active directories. This is a reasonable approximation since FFS allocates all of the files in a directory to the same cylinder group.

[Blackwell95], and was generated from a server used for a typical academic workload, consisting of text editing, compilation, executing simulations, etc. I therefore only use this trace to generate aging workloads from file systems that were used in similar environments. In order to generate aging workloads for other types of file system activity, such as database or news servers, I would need to use different traces to approximate the activity to short-lived files.

3.2.2 Replaying the Workload

To age a file system, I apply an aging workload generated as described above to an empty file system. In all of my measurements, I use a target file system that is the same size as the file system from which the aging workload was generated, although an aging workload could also be used on larger file systems. The aging program reads records from the workload file, performing the specified file operations. Although the aging workload includes timestamps for each file operation, I simply execute the requests as rapidly as possible. Replaying the workload in real time was unnecessary for our purposes, because in FFS (and many other file systems) the order in which requests are received by the file system, not the relative times of the requests, determines the behavior of the file system.

The task of replaying an aging workload was complicated by the fact that the file system snapshots did not provide pathnames for the files. Because FFS exploits expected patterns of locality by allocating files in the same directory to the same cylinder group on the disk, the algorithm used by the aging program to assign files to directories can have a major impact on the accuracy of the aging simulation.

Due to the absence of the original pathnames in the file system snapshots, I decided that it would be sufficient to create the files in the correct cylinder groups. By creating files in the same cylinder group on the simulated file system as on the original file system, I ensured that each cylinder group on the simulated file system received the same set of allocation and deallocation requests that were presented to the corresponding cylinder group on the original file system from which the snapshots were generated. I used each files's inode number to compute the cylinder group to which it was allocated

on the original file system. To force the files into the same cylinder groups on the aged file system, I exploited several details of the FFS implementation.

I start the aging process with an empty file system. The first step is to create one directory for each cylinder group on the file system. The algorithm used by FFS to assign directories to cylinder groups ensures that each directory was placed in a different cylinder group. For each file in the aging workload, I use its inode number to compute the cylinder group to which it was allocated on the original file system, and place the file in the corresponding directory on the aged file system. Because FFS places all files in the same cylinder group as their directory, this guarantees that all of the files that are in the same cylinder group on the original file system are also in the same cylinder group on the aged file system.

There are two drawbacks to this approach. First, by creating an extra directory for each cylinder group, I am introducing one file per cylinder group that did not exist in any of the data sets used to generate the aging workload (i.e., the directory). The effect of these directories should be negligible, however, as the space that they occupy is much less than that of the files being manipulated during the aging simulation. The second drawback is that by exploiting these details of the FFS implementation, I am limiting the applicability of my file system aging tools to file systems that use the same physical partitioning to improve the clustering of logically related data.

3.2.3 Workload Verification

In order to evaluate the realism of my simulation, I compared a test file system aged using our techniques with the real file system from which I generated the aging workload. Because my test file system necessarily starts in an empty state, I generated an aging workload from a file system for which I had snapshots starting the day it was created. This file system, which contains the home directories of several graduate students studying parallel computing, was not one of the file systems that I used in deriving the aging methodology. The aging workload I generated from this file system simulates 215 days (approximately seven months) of activity on a one gigabyte file system. The workload

CPU Parameters		Disk Parameters		File System Parameters	
CPU	Intel Pentium Pro	Disk Controller	NCR 53c825	Size	1024 MB
Clock Speed	200 MHz	Disk Type	Fujitsu M2694ES	Fragment Size	1 KB
Memory	32 MB EDO RAM	Total Disk Space	1080 MB	Block Size	8 KB
Bus Type	PCI	Rotational Speed	5400 RPM	Max. Cluster Size	56 KB
		Cylinders	1818	Rotational Gap	0
		Heads	15	Cylinder Groups	63
		Avg. Sectors/ Track	94	*Heads*	*19*
		Track Buffer	512 KB	*Sectors/Track*	*111*
		Average Seek	9.5 ms		

Table 3.1: Benchmark configuration. This table describes the hardware configuration used for benchmarking and verifying the file system aging workload. The file system parameters shown in italics were set to match the file system from which I generated the aging workload, despite the fact that they do not match the underlying hardware.

contains approximately 1.3 million file operations that write 87.3 gigabytes of data to the disk and takes seven hours to replay on a generic FFS implementation.[7] At the end of the workload, the file system is 65% full.

I ran this aging workload on a test file system that was configured with the same file system parameters as the original system and compared the resulting state of the test file system with the state of the original file system at the end of the sequence of snapshots. In this discussion, I refer to the original file system from which the aging workload was generated as the *real file system*, and I refer to the test file system that was aged using my artificial workload as the *simulated file system*. Table 3.1 describes the hardware configuration that I used both to age the simulated file system, and for the benchmarks described in Section 3.3.

One of the primary changes observed in many file system architectures as a system ages is increased file fragmentation on the disk. Therefore I started by comparing

7. I measured the seven hour replay time for this aging workload on a file system mounted with BSD's *async* option. This option forces all file system writes to occur asynchronously, including the synchronous meta-data updates described in Section 2.1.3. It is impractical to use this option on a file system holding live data, but it is useful when running an aging workload. If there is a system failure while executing the aging workload, there is no danger of losing valuable data as we can simply re-run the workload. Running the same workload on a file system mounted without the *async* option takes 39 hours.

several aspects of fragmentation on the real and simulated file systems. I define a *layout score* to quantify the amount of file fragmentation in a file or file system. The layout score for an individual file is the fraction of that file's blocks that are optimally allocated. An optimally allocated block is one that is contiguous with the preceding block of the same file. The first block of a file is not included in this calculation, since it is impossible for it to have a "previous block." Similarly, layout score is not defined for one block files, since they cannot be fragmented. A file with a layout score of 1.00 is perfectly allocated; all of its blocks are contiguously allocated on the disk. A file with a layout score of 0.00 has no contiguously allocated blocks.

To evaluate the fragmentation of a set of files (or of an entire file system), I compute the *aggregate layout score* for the files. This metric is the fraction of the blocks in all of the files that are optimally allocated (again ignoring the first block of each file and one block files).

At the end of the simulation period, the aggregate layout score on the real file system was 0.815, compared to 0.876 on the simulated file system.[8] Thus, although the aging workload does cause fragmentation on the file system, it does not generate as much fragmentation as occurred on the real file system. Figure 3.3, which presents a time series of the aggregate layout scores for both file systems over the 215 days of the simulation, indicates that although the aggregate layout score of the simulated file system tracked the real file system very closely for the first half of the simulation, during the second half of the simulation, the aging workload failed to replicate several large changes in file fragmentation on the real file system.

To gain a better understanding of the fragmentation differences between the real and simulated file systems, I sorted the files on both file systems by size and computed the aggregate layout scores for files of a variety of sizes. Figure 3.4 shows the results. Although the two file systems have similar layout scores for small files (up to 64 KB), for larger files, the simulated file system has higher layout scores, indicating that it failed to

8. Note that these seemingly high layout scores—more than 80% of the blocks on both the real and simulated file systems were optimally allocated—are typical of FFS. On all of the file systems in the snapshot library, there is seldom an aggregate layout score of less than 0.7 except on news servers, which are subject to extreme file fragmentation.

capture all of the fragmentation that actually occurred on the real system. It is these large files that cause the aged file system to have a higher aggregate layout score then the real one. The difference in layout scores is most noteworthy for files of 2 – 4 MB. It is not clear what caused this discrepancy. Many files of these sizes on the original file system are unusually fragmented, and have layout scores of less than 0.5. On other file systems that I have examined, large files do not exhibit this degree of fragmentation. According to the the owners of these files, they are outputs from large simulation programs. I have speculated that file activity concurrent with the creation of these large files, and taking place in the same cylinder groups, may have caused this fragmentation, but I do not have the data necessary to confirm this hypothesis.

To summarize, the simulated aging workload mimics the real file system from which it was derived in the steady increase in fragmentation over time. However, the total amount of fragmentation on the simulated system is less than on the real file system, largely because the simulated file system failed to replicate several large changes in fragmentation seen on the real file system. The fundamental cause of this inaccuracy in our aging workload is that when I did not have sufficient information to perfectly

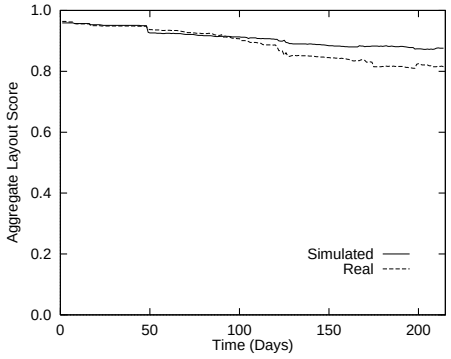

Figure 3.3. Real vs. simulated file system. This chart plots the aggregate layout score for each day in the seven month simulation period. The "Simulated" line shows the fragmentation on the artificially aged file system. The "Real" line shows the fragmentation on the original file system from which the aging workload was generated. Although the two file systems behave similarly for the first half of the simulation, the aging workload fails to capture several of the large changes in the original file system workload during the later half of the simulation period.

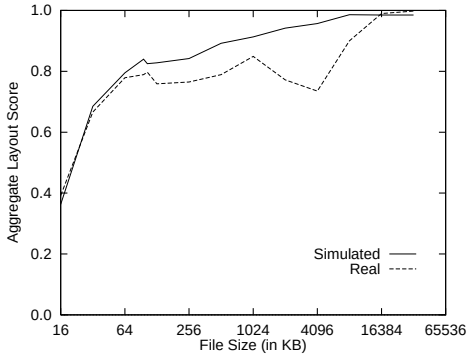

Figure 3.4. Fragmentation as a function of file size. File sizes were rounded up to an even number of file blocks and the aggregate layout score was computed for files of various sizes on the real and simulated file systems. The results are graphed here. Both file systems suffer from extreme fragmentation of small files (< 32 KB). On the real file system, file layout drops noticeably for large files (2 – 4 MB). A similar decline is not present on the simulated system.

reconstruct the workload on the real file system, I made randomized decisions. The two most important areas where this occurred were in assigning times to file delete operations and in simulating the activity of short-lived files on the file system. In real file system workloads, there are dependencies between these operations and the other activity occurring on the file system. An accurate model of these interdependencies would allow more realistic decisions regarding file delete times and short-lived file activity. The absence of such a model decreases the verisimilitude of the aging. Nevertheless, these tools are still superior to the traditional approach of benchmarking empty file systems, and they are effective for evaluating the impact of design decisions on the long term behavior of a file system.

3.3 Applications of Aging

In an earlier study [Smith96], I used file system aging to analyze the effectiveness of an improved block allocation scheme in FFS. On an empty file system, the original and improved schemes were virtually indistinguishable, but on an aged file system the improved scheme resulted in performance improvements of up to fifty percent. Aging enables a researcher to explore the long term effects of a number of policy decisions and file system features. In this section, I will use my aging methodology to answer the following questions about FFS layout.

- Indirect blocks (blocks that contain pointers to data blocks) are usually allocated in a separate cylinder group from the one containing the previous part of the file. This imposes a sharp performance penalty on midsize files (i.e., 104 KB to 256 KB). If we allocate the first indirect block of a file in the same cylinder group as the start of the file, how does this affect performance? Are there any undesirable side effects?
- Fragments (partial blocks) are rarely allocated adjacent to the preceding block of their file. Placing fragments adjacent to their preceding blocks may improve performance, but it may also lead to more internal fragmentation. Is changing fragment allocation beneficial?

The basic technique used in exploring these two issues was to propose and implement a modification to FFS. I then aged two file systems that differed only in this modification, and ran a variety of benchmarks on the aged file system to evaluate the effect of the proposed change on the long term behavior of the file system.

In order to compare the performance of two file systems, we used two simple benchmark programs. The first measures the file system throughput sequentially reading and writing files of a variety of sizes. Each run of the benchmark measures the read and write performance for one file size. The benchmark operates on 32 MB of data, which is decomposed into the appropriate number of files for the file size being measured. Because FFS allocates all of the files in a single directory to the same cylinder group, the data is divided into subdirectories, each containing no more than twenty-five files. This increases the number of cylinder groups exercised during the benchmark.

The benchmark executes in two phases:

1. **Create/Write:** All of the files are created. For file sizes of 4 MB or less, the entire file is created with one write operation. Large files are created using as many 4 MB writes as necessary. This phase measures write throughput, including the time required to create new files and allocate disk space to them.
2. **Read:** The test file system is unmounted and remounted to flush the file cache. Then the files are read in the same order in which they were created. As with the create phase, I/O is performed in 4 MB units.

For each file size in our tests, I executed this benchmark ten times, averaging the resulting throughput measurements. In all test cases, the standard deviation was less than 1% of the average throughput.

This benchmark is unrealistic in one important sense. Real file system workloads seldom create large batches of files of the same size. Actual usage patterns typically interleave the creation and deletion of files of a variety of sizes, possibly resulting in more file fragmentation than we would see in the sequential I/O benchmark described above. Our second benchmark attempts to address the problem, by exploiting the more

"realistically" created files that are left on the test file system at the end of the aging workload.

Previous research has shown that most older files are seldom accessed [Satyanarayanan81], and therefore that the most active files on a file system tend to be relatively young. I approximated the set of *hot* files on our simulated file system by using all of the files that were modified during the last thirty days of the aging workload. These files represent 9.5% of the files on the aged file system (3,207 out of 33,797 files), and use 92.3 megabytes of storage (14.5% of the allocated disk space).

The second benchmark measures file system throughput when reading and writing this complete set of hot files. To limit the amount of time spent seeking from one file to the next, I sorted the files by directory, so multiple files would be read from one cylinder group before moving to another. To preserve the file layouts, I overwrite the files during the write phase of this test.

I used FFS enhanced with the improved block-clustering algorithm [Smith96] mentioned in Section 3.1.2 as the baseline system.[9] The performance of this file system (after aging) is shown in Figure 3.5.

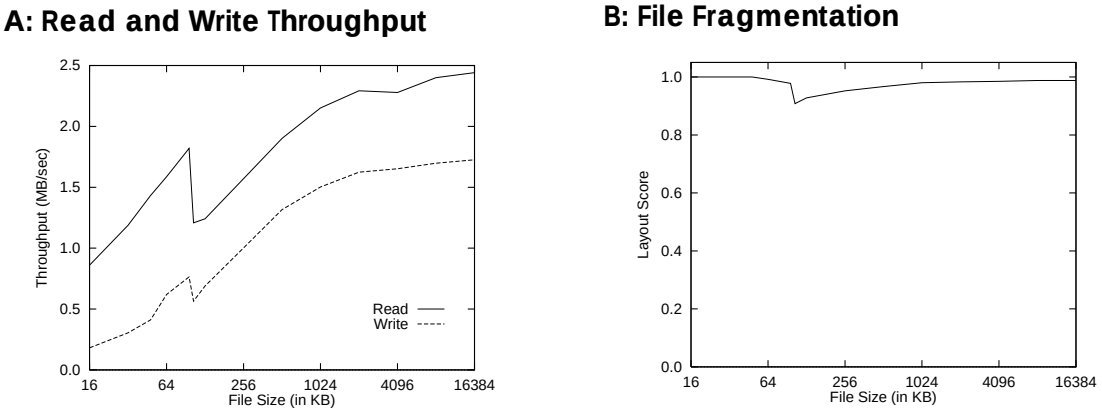

Figure 3.5. Performance baseline. These charts show the performance of our baseline file system in the file throughput benchmark. The benchmark was executed after the file system had been aged using the workload described in Section 3.2.1. Graph A plots read and write throughput as a function of file size. Graph B plots the layout scores of the test files created during the benchmark. The sharp drops in all of these graphs as the file size passed 96 KB corresponds to the point where FFS allocates the first indirect block to a file (see Section 3.3.1).

9. At the time I performed these experiments, this allocation algorithm was still considered experimental. It is now a standard part of FFS on many BSD-based systems.

3.3.1 Indirect Block Allocation

Each time an indirect block is allocated to a file in FFS, the file system assigns that block, and all of the data blocks it references, to a different cylinder group than the previous part of the file. The new cylinder group is chosen by selecting the next cylinder group on the disk that has at least an average number of free blocks (relative to the rest of the file system).

This scheme seems undesirable, as it forces long seeks at periodic locations in large files. For very large files, however, these extra seeks are typically amortized over the transfer of the entire file, and have a negligible effect on I/O throughput. Given a typical file system block size of 8 KB, this policy will force a change of cylinder groups every 16 MB of the file. However, switching cylinder groups may be useful in practice, as it prevents a single large file from consuming all of the free space in a cylinder group.

Unfortunately, there is one glaring problem with this policy of switching cylinder groups with the allocation of each indirect block of a file—in FFS the first indirect block is allocated after only the twelfth data block of a file. On an 8 KB file system, this means that FFS imposes an extra seek after the first 96 KB of a file. For medium size files of a few hundred kilobytes, this extra seek can have a noticeable impact on performance. The effect of this extra seek is apparent in the performance of our baseline file system in Figure 3.5A. The layout score of the test files drops from 0.98 to 0.91 when the first indirect block is allocated (between 96 KB and 104 KB) and both read and write performance decline precipitously at the same point. There is a larger drop in read performance (33%) than in write performance (25%) because the indirect block not only causes a seek during the read, but also interferes with file prefetching, as the blocks referenced from the indirect block cannot be prefetched until the indirect block itself has been read from the disk.

To address this problem, I modified FFS to not switch cylinder groups until it allocates the second indirect block in a file. (In the test file systems, this occurs when the file size reaches approximately 16 MB). I call the implementation of FFS that includes this enhancement *NoSwitch*. I expected this minor enhancement to have the effect of

improving file throughput for files of a few hundred kilobytes. Larger files should not see as much improvement because the savings from eliminating one seek are amortized over the time it takes to read or write the entire file. I used the throughput benchmark to compare the performance of the NoSwitch file system to our baseline file system on both empty and aged partitions. Figure 3.6 shows the results.

As expected, read and write throughput to files of a few hundred kilobytes improves on the NoSwitch file system. Note that there is still a slight performance drop as file size passes 96 KB and the first indirect block is used. This occurs because in addition to transferring the file data, the file system must also transfer the indirect block. Comparing the performance on empty and aged file systems in Figure 3.6 we see that the NoSwitch system outperforms our baseline in both cases. The magnitude of the performance improvement, as shown by the area between the pairs of curves for the two file systems, is smaller on the aged file system. In the best case (104 KB files) the NoSwitch file system improves performance by 87% on an empty file system, but only by 43% on an aged file system.

If the only concern were whether the NoSwitch file system would improve performance, we would not have needed to run our benchmarks on an aged file system. By using an aged file system, however, we can more accurately assess the magnitude of the performance improvement. Using an aged file system also allows us to assess an

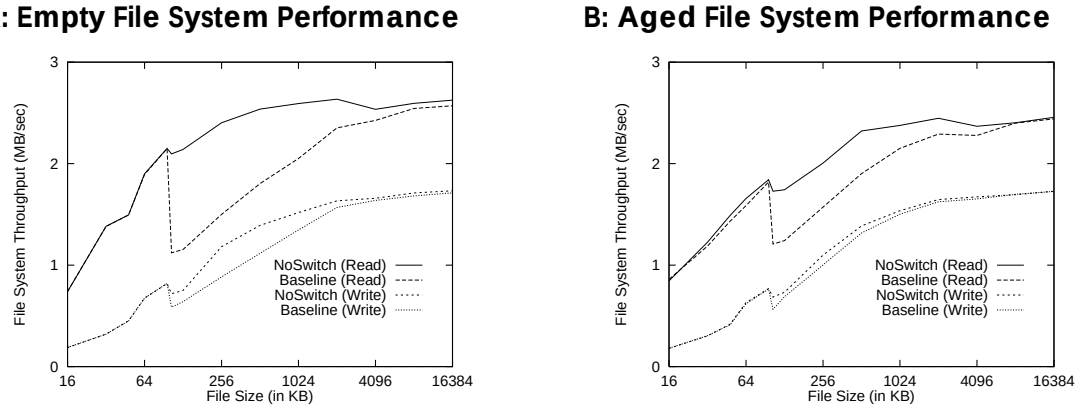

Figure 3.6. Performance with improved indirect block allocation. These charts compare the read and write throughput of the baseline file system to a file system that does not switch cylinder groups when allocating the first indirect block (*NoSwitch*). Graph A shows this comparison on an empty file system; Graph B shows this comparison on aged file systems. The NoSwitch file system offers higher throughput in both cases, but the magnitude of the improvement, indicated by the area between the NoSwitch and Baseline lines in the graphs, is significantly smaller on the aged file systems.

adverse side effect of this enhancement. As described earlier, FFS attempts to exploit locality of reference by co-locating all of the files in a directory in the same cylinder group as the directory itself. In a directory with many files, some of which are large, the original scheme of switching cylinder groups after only twelve blocks of a large file may have ensured that a single large file did not consume all of the free space in a cylinder group, forcing subsequently allocated files to be placed in other cylinder groups, thus destroying the desired locality.

To study this effect, I examined the state of the baseline and NoSwitch file systems after they had been aged. If the NoSwitch file system caused an increase in the number of files displaced from the cylinder group of their directory, we would expect to see a larger number of files where the first data block of the file is in a different cylinder group from the file's inode. (FFS also tries to locate a file's inode in the same cylinder group as its directory.) I counted the number of these *split files* on the two file systems, and for each such file, determined how many cylinder groups separated the file's inode and its first data block. The more intervening cylinder groups, the longer the seek required to read the file's data after reading its inode. The results are summarized in Table 3.2.

	Baseline	**NoSwitch**
Number of split files	4312	9155
% of all files that are split	13	27
% of one cyl. group splits	58	37
% of < 10 cyl. group splits	95	67

Table 3.2: Number of split files on NoSwitch file systems. This table compares the number of split files (files where the inode and the first data block are in different cylinder groups) on the baseline file system and on the aged file system with the NoSwitch enhancement. The four rows of the table present, respectively, the total number of split files on each file system, the percentage of all files on each file system that are split, the percentage of split files where the data block is only one cylinder group away from the inode, and the percentage of split files where the data block is no more than ten cylinder groups away from the inode. Both file systems had sixty-three cylinder groups.

The NoSwitch file system has more than twice as many of these split files as the baseline file system, indicating that not switching cylinder groups when the first indirect block is allocated does, in fact, cause highly utilized cylinder group to run out of free space. On the baseline file system, most of the split files require relatively short seeks; in

more than half the cases, the file's data is only one cylinder group away, and in almost all cases, the data is within ten cylinder groups of the file's inode. In contrast, a third of the split files on the NoSwitch file system involve seeks of more than ten cylinder groups.

In an attempt to balance the performance gain for large files, which are not allocated in one cylinder group, against the potential performance loss from the longer seeks required to read the extra split files that are generated on the NoSwitch file system, we turn to the results of the hot file benchmark. The results of this benchmark, which are summarized in Table 3.3, show that the NoSwitch file system offers a modest improvement in read throughput, with virtually no change in write throughput. This performance improvement suggests that the throughput gained from a better layout of larger files outweighs the throughput lost by increasing the number of split files. The improvement in performance is small enough, however, that it may be an artifact of this particular workload, and this file system modification may not be universally applicable. It is important to note, however, that we would have had no means to evaluate this trade-off if we had only benchmarked NoSwitch on an empty file system.

	Baseline	**NoSwitch**
Aggregate Layout Score	0.928	0.931
# of split files	327	594
Read bandwidth (MB/sec)	0.810	0.835
Write bandwidth (MB/sec)	0.494	0.495

Table 3.3: Performance of recently modified files on NoSwitch file system. This table presents the read and write throughput of the files modified during the last thirty days of the aging workload on the baseline and NoSwitch file systems. The aggregate layout scores of the files used during this test, and the number of these files where the first data block was located in a different cylinder group than the file's inode ("split files") are also presented. Throughput measurements are the averages of ten test runs. All standard deviations were less than 0.2% of the reported means.

3.3.2 Fragment Allocation in FFS

To limit the amount of internal fragmentation caused by small files, FFS allows a single file system block to be subdivided into *fragments*. The minimum fragment size is determined at the time that the file system is created, and blocks may only be divided into pieces that are integral multiples of the fragment size. For files with no more than twelve data blocks (i.e., files that do not use any indirect blocks), a partial block containing an

integral number of fragments may be used as the last data block instead of a full-sized file system block. On our test file system, for example, the block size was 8 KB and the fragment size was 1 KB. Thus, a 30 KB file would be allocated as three file blocks, followed by a partial block containing six fragments.

While this scheme is efficient in reducing the amount of disk space wasted by internal fragmentation, the algorithm FFS uses to allocate fragments to files results in suboptimal file layout. When allocating a fragment, FFS first attempts to find a free fragment of the appropriate size in the same cylinder group as the file. If such a fragment is not available, FFS will divide a larger free fragment. Finally, if no fragment of an appropriate size is available, FFS will allocate an entire file system block, and divide it into fragments. Thus the primary goal of the fragment allocation algorithm is to limit the amount of free space that exists in fragments. The downside of this approach is that the fragment at the end of a file is seldom allocated near the preceding block of the file. In Figure 3.4, for example, we see that the layout scores of small files are much lower than those for other files, indicating that small files are more fragmented. This fragmentation is almost entirely due to the fragment allocation policy. On the baseline file system, for example, only 36% of two block files are allocated with their two blocks contiguous on disk. Of the two block files where the second block is a full block rather then a fragment, however, 87% are allocated contiguously.

Ideally, we would like the fragment at the end of a file to be contiguous with the preceding block of the file. To this end, I modified the FFS fragment allocation algorithm. The new algorithm always attempts to allocate the block immediately adjacent to the previous file block. If that block is available, it is broken into fragments, and the unused portion is marked as free. If the desired block is not available, we fall back to FFS's original fragment allocation policy. For small files, where the only data block is a fragment, we always use the original FFS policy, hoping to fill in the free fragments created when full blocks are broken up to provide contiguous fragments for larger files. I refer to the version of FFS that uses this new fragment allocation policy as *SmartFrag*.

I used the sequential I/O benchmark to compare the performance of the SmartFrag file system to that of the baseline system. Since we are interested in the behavior of files that use fragments, I focused on small files in running this benchmark. Figure 3.7 shows the layout score of the small files created by running the benchmark on the two aged file systems. Figure 3.8 presents the measured performance of the two versions of FFS on both empty and aged file systems.

Figure 3.7 shows that the SmartFrag scheme dramatically decreases file fragmentation for files that use fragments. Both the SmartFrag and baseline file systems achieve nearly perfect layout for file sizes that are an integral multiple of the eight kilobyte disk block size. For intermediate sizes, however, SmartFrag eliminates almost all of the fragmentation seen on the baseline system.

This difference in file layout translates to the performance differences seen in Figure 3.8. The saw-tooth effect in the read performance on all of the tested systems is caused by changes in the performance characteristics of the file systems when fragments are used. All file sizes that are even multiples of the file system block size do not require fragments. Note that at these file sizes, the performance of the baseline file system is the same as on the SmartFrag file system, as they both use the same file layout algorithm. For file sizes that are not integral multiples of the file system block size, SmartFrag outperforms the baseline system due to the improved allocation of fragments for these

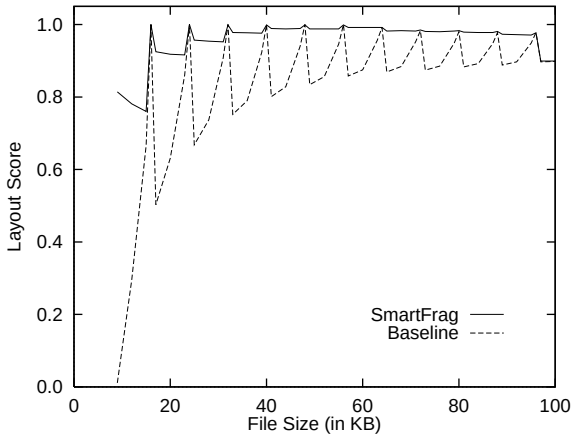

Figure 3.7. File layout with smart fragment allocation. This graph shows the amount of file fragmentation for small file sizes on the baseline and SmartFrag file systems. The layout score is plotted for the files created by the throughput test in Figure 3.8B.

files. Both the SmartFrag and baseline systems have decreased throughput for files that use fragments because FFS issues a separate I/O request to the disk driver for the fragment, regardless of whether the fragment is contiguous with the previous file block.

The differences in write throughput are much smaller because this test both creates and writes the test files, and the test time is dominated by the create operation, which requires synchronous disk writes [Seltzer95]. The cost of creating a file is the same for both the SmartFrag and baseline file systems.

The write performance on all of the tested systems also shows an unexpected jump when the file size reaches 64 KB. This is the result of a performance bug (since fixed) in the version of FFS used in these tests. Until a full cluster (64 KB) of data has been written to a file, FFS did not use clustered writes. The result is that for smaller file sizes, FFS issued one write request for each file block, regardless of the layout on disk. At these file sizes, the overhead of performing these individual I/O operations completely masked any performance differences caused by the fragment allocation policy. For files larger than 64 KB, we see that the SmartFrag file system provided improved throughput for file sizes that required the use of a fragment.

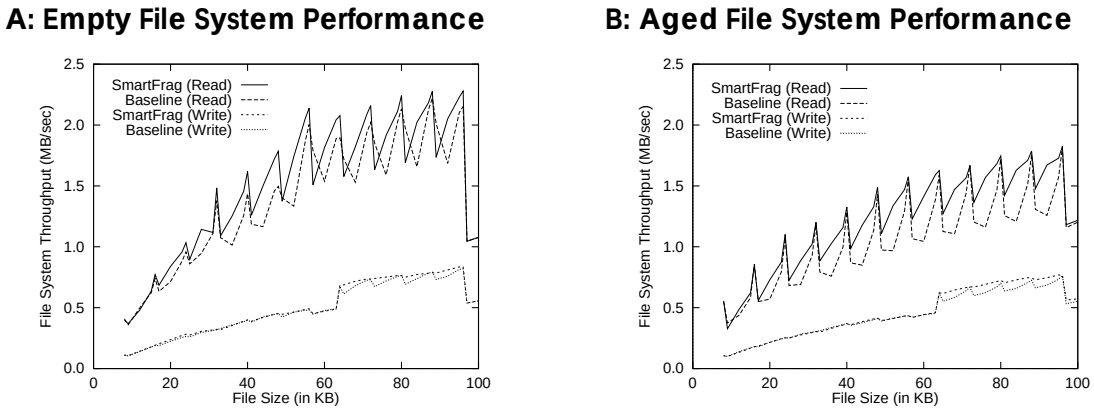

Figure 3.8. Performance with smart fragment allocation. These charts compare the read and write throughput of the baseline file system to a file system that uses our improved fragment allocation algorithm (*SmartFrag*). Graph A shows this comparison on empty file systems; Graph B shows this comparison on aged file systems. The saw-tooth effect shows the impact of changing fragment size on file system performance. The peaks represent file sizes that are an integral number of blocks. File sizes that require the use of a fragment do not perform as well because reading or writing the fragment requires an extra I/O operation. The step in writer performance at 64 KB files it the result of a performance bug in FFS, described in the body of this chapter. All of the performance curves drop precipitously after 96 KB because FFS switches cylinder groups at this point.

The potential downside to the SmartFrag strategy is the amount of fragmentation of free space that it causes. Most of the data on a file system is allocated in full-sized file system blocks. If too much of the file system's free space is in fragments instead of full-sized blocks, the file system may run out of free blocks while there is still a sizeable amount of free space in fragments. Seltzer and her colleagues have described a particularly spectacular instance of this problem [Seltzer95]; one of their news servers reported that it was out of free space despite the fact the file system had more than ninety megabytes (ten percent of the disk) free—all in fragments.

To evaluate how much the SmartFrag file system increases the fragmentation of free space, I compared the number of free blocks and free fragments on the baseline and SmartFrag file systems. As expected, both file systems had the same amount of free space. On the SmartFrag file system, however, twice as much of this space was in fragments (5% vs. 2.5% of free space). Because the total amount of fragmented free space is relatively small on both file systems, this side effect of the SmartFrag allocation scheme is tolerable for most applications; it is unlikely to cause problems until the file system is very close to maximum capacity.

3.4 Conclusions

The behavior of a file system can change dramatically with the passage of time. As a file system is filled, or as successive generations of files are created, modified, and deleted, the performance characteristics of the system also change. By ignoring these changes in file system behavior, researchers fail to accurately assess how file system designs will respond to real-world conditions. Not only do active file systems behave differently from empty ones, but there are also a variety of file system design decisions whose full effects are only apparent after a long period of use.

In order to accurately evaluate the long-term behavior of competing file system architectures, I have developed a process for artificially aging a file system by replaying a long-term workload on a test file system. As demonstrated by the evaluation of two new file layout policies for the UNIX fast file system, this technology allows for the scientific

evaluation of design decisions that may have no discernible effect on the short-term characteristics of file system behavior.

Chapter 4

Workload-Specific Performance Analysis

One of the fundamental questions that benchmarking tries to answer is, “How well will my workload perform on this system?” In the previous chapter, I demonstrated that file system aging helps to answer this question by producing benchmark results that are more accurate and realistic than the traditional approach of benchmarking empty file systems. But the results achieved by measuring the performance of an aged file system can only be as good as the benchmark used to generate those results. Regardless of the verisimilitude of the test file system, a read-intensive benchmark is probably a poor predictor for the performance of a write-intensive workload.

Unfortunately, current file system benchmarks often suffer from this type of mismatch with user workloads. Typical benchmarks, such as SFS, Andrew, and IOStone, assign a single score to each system that they measure. The tacit assumption made by these benchmarks is that there is a single one-dimensional ranking of the systems under test. In the real world, this is seldom the case. The file system that performs best for one workload, may not be the best for another workload. This has been shown repeatedly, in a variety of research papers comparing different file system architectures [Seltzer93] [Patterson95] [Tomkins97] [Seltzer00a].

The ideal benchmark would allow a user to evaluate a system in the context of her own workload. Workload-specific performance evaluation is becoming increasingly important in today's computing environment. Many tasks that were formerly handled by a single central server now run on dedicated hardware. For example, many large computing installations have separate systems acting as file servers, mail servers, web servers, news servers, name servers, etc. Ideally, administrators of such sites should have tools that they could use to determine which of several competing hardware and operating system configurations would provide the best performance for each of these services. These tools could also be used to evaluate the benefits of hardware upgrades, such as installing more memory or faster disks, and to determine optimal configuration parameters for applications and operating systems.

Traditionally, if a user has wanted to evaluate the performance of her workload on a variety of file systems, she has had to actually execute that workload on the various systems of interest. While this is possible for users with the clout (financial or otherwise) to get access to evaluation systems, it is less practical for most users. To facilitate the (common) situation of the latter class of users, I have designed a benchmarking methodology that allows a user to evaluate the performance of her workload on a file system without having access to an actual machine using that file system.

The key to this benchmarking methodology is separating the measurement of the file system from the analysis of the user's workload. In this scheme, a file system vendor provides a profile of its file system—the results of a suite of microbenchmarks that evaluate different aspects of the system's performance. The user analyzes her application by collecting a trace of the calls to the file system API. By combining these two profiles I predict the performance of the workload on the file system and identify the types of operations where the file system consumes the most time. The latter information is useful both to application developers, who wish to tune their software to avoid file system bottlenecks, and to file system architects who wish to eliminate those bottlenecks.

In the next section, I motivate this work by demonstrating that different file system architectures provide better performance for different workloads. In Section 4.2, I

discuss traditional approaches to analyzing file system performance. Section 4.3 describes the goals of my workload-specific performance analysis tools. In Section 4.4 I explain the architecture and implementation of these tools, and in Section 4.5 I validate them by comparing their performance predictions against the actual behavior of several workloads on a variety of file system architectures. Section 4.6 describes possible avenues for future work. Finally, in Section 4.7 I present my conclusions.

4.1 Motivation

Most existing macrobenchmarks attempt to reduce file system performance to a single number. Thus, these benchmarks assume that the file system with the best benchmark result is the best performing. Unfortunately, real world file systems seldom lend themselves to such simple analysis. The file system that is best for one workload may not be the best for a different workload. Even similar benchmark programs, intended to represent the same type of workload, may rank files systems differently.

The problem can be observed in Figure 1, which shows data collected in an earlier study comparing a log-structured file system ("LFS") to two versions of FFS [Seltzer93]. In this figure, "FFS" is an old version of the Fast File System, that does not include support for clustered I/O (see Section 2.1.3.2.1), and "EFS" is a version of FFS that includes clustered I/O. The graph shows the time to run one or more concurrent instances of the Andrew benchmark [Howard88]. This set of benchmark results clearly shows that there is no single file system that consistently out-performs the others. When only one instance of the Andrew benchmark was run, LFS outperformed the other two file systems. When six instances of the Andrew benchmark were run, EFS showed the best performance, and LFS the worst performance. Many other research studies comparing multiple file system architectures, have shown this same phenomenon—that ranking file systems by performance depends on the benchmark used [Howard88] [Patterson95] [Tomkins97] [Seltzer00a].

4.2 Existing Benchmarking Techniques

Traditional techniques for analyzing and understanding file system performance have relied on a mix of both microbenchmarks and macrobenchmarks. Researchers typically use microbenchmark results to explain (or predict) the performance of macrobenchmarks. At its best, this approach allows a researcher to understand which file system operations perform well (or poorly) on a particular system and to examine how these microbenchmark results influence the performance of large-scale workloads, highlighting the workloads that are well-suited to the architecture.

While this type of measurement can provide excellent insight into the behavior of the systems under study, it is also painstaking to carry out. As it is seldom practical for such studies to focus on more than a handful of workloads, the results may be of little benefit to users whose workloads differ from the macrobenchmarks in the study. The only recourse for such a user is to perform her own analysis of the system in question,

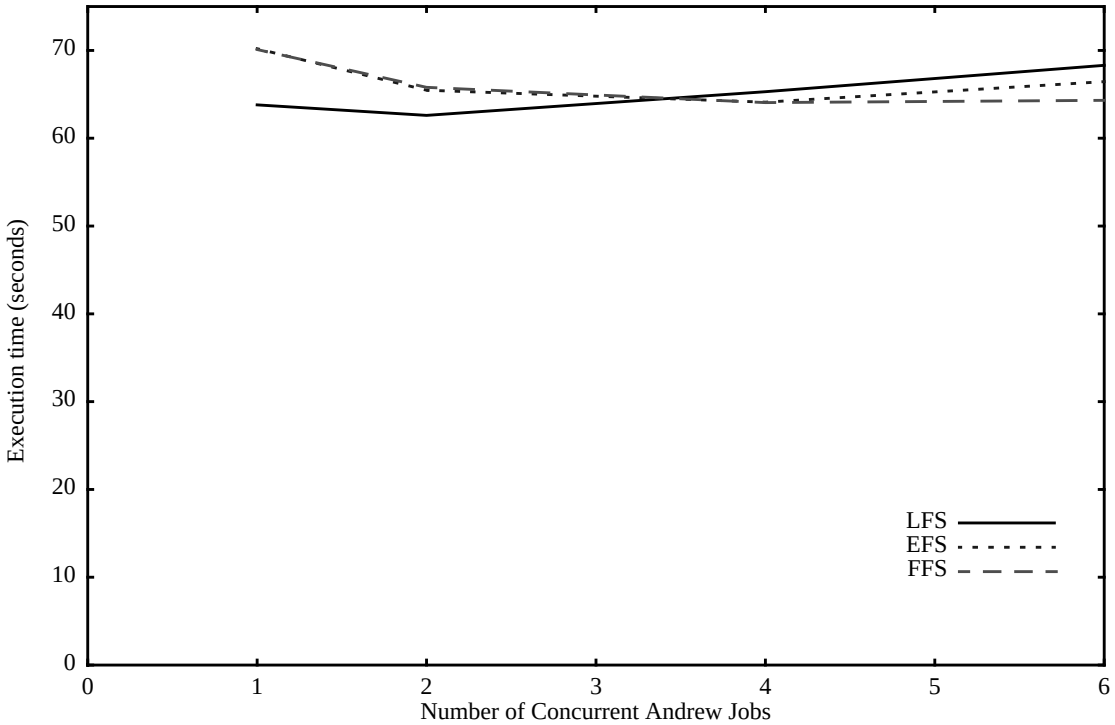

Figure 4.1. File System Comparison. This graph compares the performance of three file systems, a log-structured file system (LFS), an old version of the Berkeley Fast File System (FFS), and a version of the Fast File System that supports clustered I/O (EFS). The graph shows file system performance (in application run time) to execute one or more concurrent instances of the Andrew benchmark. The file system offering the best performance varies, depending on the degree of multiprograming. This data was copied from an earlier publication with permission of the author [Seltzer93].

either by performing her own measurements of the system, or by trying to infer the performance of her workload from previously published results. The former approach requires access to the file system of interest, a potential problem for users with limited financial resources. The latter approach requires a detailed understanding of the interaction between the workload and the file system. Inferring the performance of a workload based on microbenchmark results alone can also be problematic if the workload stresses parts of the system that were not microbenchmarked.

My benchmarking methodology is an attempt to automate this general approach of using microbenchmarks to understand workload performance. I provide an extensive suite of microbenchmarks that evaluate the performance of all aspects of standard file system functionality and a set of tools that analyze the performance of a workload using the results of these microbenchmarks. The output of this analysis provides the same types of information researchers have traditionally generated by hand when analyzing file system performance.

4.3 Goals

My goal is to develop a benchmarking methodology that explicitly acknowledges the workload-dependent nature of file system performance. This benchmark should allow users to understand how a given workload will behave on different file system architectures. It should also highlight the performance trade-offs the different file systems make, and the impact of those trade-offs on performance. In this chapter, I present a set of tools, called *HBench-FS*,[1] that allow a researcher to perform several important types of performance analysis.

- HBench-FS predicts the overall performance of a workload on a target file system. This allows easy comparisons between competing file system architectures in the context of a workload of interest. For these comparisons to be useful, HBench-FS must provide accurate performance

1. Several other researchers at Harvard have been investigating workload-specific benchmarks for different types of computer systems. The name “HBench-FS” is part of the overall naming scheme for these benchmark suites. Hence *HBench-OS* [Brown97a] measures operating system performance, *HBench-Web* [Manley98] measures web server performance, etc.

predictions. More importantly, HBench-FS must be able to correctly rank different file systems in terms of the performance they provide for a workload of interest.

- HBench-FS provides an analysis of the underlying causes of performance differences between different systems. In addition to predicting the overall performance of a workload on a target file system, HBench-FS breaks down the performance in terms of file system functionality, showing the request types that consume the most time. While the ultimate goal is to provide complete accuracy in this performance breakdown, the data that is most useful to researchers and developers is information about which types of file system operations consume the most time for a particular workload and file system combination.
- Developers and researchers can use the results from HBench-FS to optimize application or file system performance in response to the specific bottlenecks a workload encounters on a file system.
- Developers or researchers contemplating changes to a file system, or an entirely new file system architecture, can use HBench-FS to conduct "what if" experiments. A user can provide a hypothetical performance characterization of the proposed file system and use HBench-FS to predict the performance of various workloads on it.

The output from HBench-FS provides a variety of information that helps to perform the different types of performance analysis described above. HBench-FS also provides access to its raw prediction data, which can be used for customized performance analysis.

While the individual tasks described above can be performed using existing techniques, such an approach usually requires a tremendous amount of expertise, as well as access to both the applications and file systems in question. HBench-FS separates file system analysis from workload analysis, allowing a user to study the performance of her workload on a file system architecture she may not have access to, or which may not exist yet. HBench-FS also automates many tasks traditionally associated with performance

analysis, including the selection of microbenchmarks, and determining which aspects of file system performance have the greatest impact on a workload.

4.4 Application-Specific Benchmarking

Figure 2 shows the high-level architecture of HBench-FS. A set of analysis tools combine data from profiles of both a file system and a workload of interest to predict how the workload will perform on the file system. The file system profile consists of the results of a large suite of file system microbenchmarks, intended to fully characterize all aspects of traditional file system functionality. The workload profile includes a system call trace of the desired workload, collected from any operating system environment, and a snapshot of the file system at the beginning of the trace. Because the file system and workload profiles can be generated separately, a researcher need not have access to both the file system implementation and the application to analyze how the two will perform together. In an ideal world, system vendors would publish file system profiles for the systems they sell. Application vendors would also publish profiles of their applications. In the simplest case, an end user could select the application and file system profiles of interest, and analyze their combined performance. More sophisticated users could generate their own application profiles.

In the following sections, I provide more details about HBench-FS, including the file system profiles, the workload profiles, and the techniques it uses to make

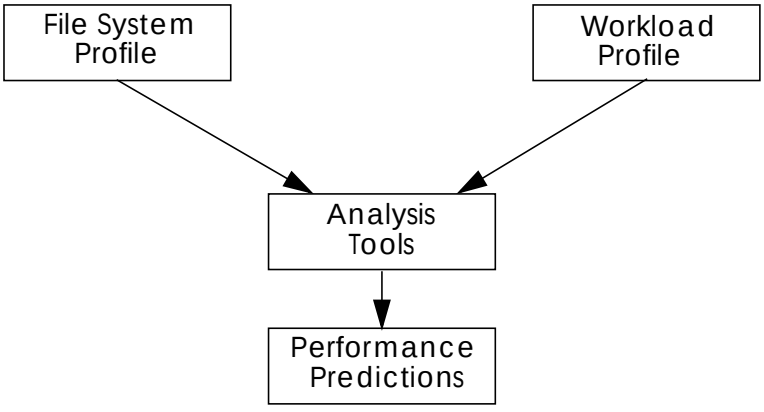

Figure 4.2. HBench-FS High-Level Architecture. HBench-FS consists of a set of tools that analyze the performance of a specific workload on a target file system by combining data about both the file system and the workload. In addition to these analysis tools, HBench-FS includes tools for generating a performance profile of a target file system.

performance predictions from these profiles. As HBench-FS is a research prototype intended to demonstrate the validity of this approach, rather than a commercial-quality product, it has several limitations, which I also describe in the following sections.

4.4.1 General Approach

Margo Seltzer and her colleagues have proposed three different methods for performing application-specific benchmarks, *vector-based*, *trace-driven*, and *hybrid* [Seltzer99]. HBench-FS falls into the latter category, combining aspects of both the vector-based and trace-driven approaches. A vector-based benchmark uses a *system vector* to characterize the performance of a system, and an *application vector* to characterize the behavior of an application. Each element of the system vector gives the performance of one aspect of the target system. The corresponding element of the application vector is a measure of how much the application uses that piece of the system's functionality. The dot-product of these vectors provides a prediction of the application's performance on the system that provided the system vector. Aaron Brown has used this technique to analyze the effect of operating systems on Apache web server performance [Brown97a]. Xiaolan Zhang has used vector-based techniques to perform application specific benchmarking of Java virtual machines [Zhang00].

Given a workload profile that consists of a stream of file system requests issued by an application of interest, the goal of HBench-FS is to predict the latency of each operation in the trace. HBench-FS makes these individual performance predictions using the vector-based methodology described above. In this case, the system vector consists of the microbenchmark results from a file system profile. HBench-FS generates an application vector (called a *request vector* in this context) for each file system request, and produces the performance prediction for each request by taking the dot-product of the two vectors.

The performance of a file system is heavily dependent on the order in which it receives requests. Read requests to sequential ranges of a file typically outperform random requests to the same file. Similarly, a series of requests that exhibit strong locality of reference will outperform a request stream with no locality of reference. HBench-FS

captures these aspects of file system performance in the system and request vectors by including separate performance elements for sequential and random access patterns and for cached and uncached file accesses. In order to generate these request vectors, HBench-FS needs know the access pattern and cache behavior of each request. As this information depends on the relationships between requests in a workload it cannot be obtained by looking at individual file system requests in isolation. This is where HBench-FS uses Seltzer's trace-driven methodology. Before generating the request vectors, HBench-FS preprocesses the workload trace using a cache simulator and other tools that annotate the trace to indicate the cache behavior and access patterns of each request.

To better understand how HBench-FS generates performance predictions consider as an example, a single request to create a file named /usr/home/keith/test/file. HBench-FS would subdivide this request into six discrete file system operations, five name lookup operations (one for each component of the pathname), and a single file create operation. HBench-FS also analyzes the cache behavior of each request it processes. Thus, it might conclude that the first three lookups will hit in the name and attribute caches, and that the final two lookups will miss in both caches. If the request vector had the form

<# lookup misses, # lookup hits, # file creates> (EQ 1)

then the application vector would be <2, 3, 1>. Table 4.1 provides sample microbenchmark measurements for these values (taken from FFS performance on the fast test configuration described in Section 4.5.1). Using these values, we get a system vector of <6.347, 0.012, 6.794>, and we predict the latency of the file creation request by taking the dot product of these vectors:

<2, 3, 1> • <6.347, 0.012, 6.794> = 19.524ms (EQ 2)

As we will see later, generating the request vector for a file system request is more complicated than I have described here. In addition to differentiating between lookups that hit or miss in the name and attribute caches, HBench-FS also considers the request pat-

tern of a stream of lookups, treating successive lookups to the same directory differently from lookups to different directories. The above analysis also assumes that the create request will succeed. In practice, it could fail for a variety of reasons—the target file might already exist, or a component of its pathname might be invalid. By analyzing the file system requests in the context of the snapshot included in the workload profile, HBench-FS can determine whether and how a request will fail, and generate the appropriate request vector.[2]

4.4.2 Limiting Assumptions

I designed HBench-FS to evaluate a limited class of file system and storage system architectures—locally-attached single-disk file systems. While this means that HBench-FS cannot be used to evaluate a variety of interesting architectures, this class of file systems provides a sufficient range of architectural variation to provide an interesting set of test cases for validating and experimenting with this benchmarking methodology.

In its current form, HBench-FS cannot model several important aspects of networked file systems. To make performance predictions for these systems, HBench-FS would need to account for the effects of network topology and congestion on file system performance. Similarly, large-scale direct-attached storage devices, which typically contain multiple disks and additional cache memory, would require extensions to HBench-FS's performance model to account for an additional level of caching and the ability to concurrently serve multiple I/O requests off of parallel disks. The general HBench-FS approach does not preclude analyzing these more complex file system architectures and in Section 4.6 I discuss some of the ways they might be supported.

Microbenchmark	Result (ms)
Lookup Miss	6.437
Lookup Hit	0.012
File Create	6.794

Table 4.1: Sample Microbenchmark Results. This table contains the microbenchmark results used for the example in Section 4.4.1. The results are taken from the *Fast FFS* configuration described in Section 4.5.1.

2. HBench-FS only checks for common error conditions, such as invalid pathnames. Other types of errors (such as invalid permissions), are much less common and typically represent exceptional case rather than the expected case behavior for applications.

The current version of HBench-FS is also limited to operating system environments that provide a POSIX, or POSIX-like interface to the file system and other basic operating system services.

4.4.3 File System Characterization

HBench-FS uses an extensive set of microbenchmarks to characterize the performance of a file system. These programs attempt to measure the performance of all aspects of a file system's functionality. They do not make any assumptions about the underlying file system architecture nor do they take advantage of any file system or operating system specific knowledge to obtain measurement data. HBench-FS uses the collective microbenchmark results to create the system vector, which it combines with the stream of request vectors to generate performance predictions for a workload.

4.4.3.1 Microbenchmark Goals

In designing the suite of microbenchmarks HBench-FS uses in generating the system vector, there were several goals and constraints. First, the benchmarks should be comprehensive. In other words, they should measure all aspects of file system functionality. This functionality is defined by the set of system calls that manipulate the file system or its files. Table 4.2 lists the important calls in the POSIX file system interface and provides a brief description of their functionality. The goal of the HBench-FS microbenchmarks then is to quantify the performance of the different aspects of file system functionality expressed in this system call interface.

Where possible, the benchmarks should be orthogonal to avoid redundancy. Ideally, each benchmark should measure exactly one piece of file system functionality, and each piece of functionality should be measured by exactly one microbenchmark. As a practical reality, however, there are areas where it is impossible to divide the file system functionality in as fine a granularity as might be desired. Writing a file, for example, typically requires allocating space for the data, as well as transferring the data to the file. In most file system interfaces, there is no way to separate and measure these two actions.

System Call	Arguments	Functionality
open	Pathname of target file. Access mode.	Open a file, returning a file descriptor. Certain file operations (e.g., read and write) can only be performed using a file descriptor.
creat	Pathname of new file. File permissions. Access mode.	Create a new file, returning a file descriptor for the new file.
unlink	Pathname of target file.	Remove the specified file.
read	File descriptor. Transfer size. Buffer.	Read data from the current file offset into an application buffer.
write	File descriptor. Transfer size. Buffer.	Write data from an application buffer to the file, beginning at the current offset.
close	File descriptor.	Close a file. The calling process can no longer use the file descriptor
lseek	File descriptor. New offset.	Set the file offset pointer
truncate	Pathname of target file.	Truncate the file, freeing the storage space allocated to the file without removing its inode or directory entry
ftruncate	File descriptor.	
mkdir	Pathname of new directory. Access mode.	Create a new directory
rmdir	Pathname of target directory.	Remove a directory
getdirentries	File descriptor. Transfer size. Buffer.	Read directory entries from a directory.
stat	Pathname of target file.	Return the attributes for a file or directory
fstat	File descriptor	
access	Pathname of target file. Desired access mode.	Determine whether specified type of access is allowed on the target file.
rename	Pathname of existing file. New pathname for file.	Change the name of a file.
chmod, chown, chgrp, utimes	Pathname of target file. New file attributes	Change the access mode, owner, or group, or access times of target file.
fchmod, fchown	File descriptor New file attributes	

Table 4.2: File System Interface. This table lists the important calls in the POSIX file system interface, lists their primary arguments, and provides a brief description of their functionality. Several of these calls (truncate, stat, chmod, chown, chgrp) have two versions, one that takes a file pathname as an argument and another (e.g., ftruncate) that takes a descriptor for an already open file as an argument. Although it is listed as a separate call, creat is usually implemented by passing a special file creation flag to open. The descriptions of some calls ignore complex aspects of their behavior.

Portability is another important goal for the microbenchmark suite. To achieve this, the microbenchmarks should access the file system only through standard public interfaces. As mentioned in Section 4.4.2, the benchmarks assume that they run on a file system that supports the POSIX interface. The benchmarks cannot exploit knowledge the designer may have about file system internals, nor can they use system-specific interfaces. For example, some systems provide a system call that provides access to the values of various system parameters such as the size of the buffer cache (e.g., the *sysctl* call in BSD UNIX). Since this call is not standard, however, we cannot rely on it to determine the size of the target system's buffer cache. Instead, HBench-FS includes a microbenchmark that measures the buffer cache size empirically.

4.4.3.2 Microbenchmark Selection

The simplest approach to building a microbenchmark suite might be to provide one microbenchmark for measuring each call in the file system interface. This approach presents a number of problems, however. First, many system calls perform differently depending on their arguments and the state of the system. The read system call is a prime example. The amount of time it takes to complete a read will vary considerably depending on how much of the target data (if any) is in the buffer cache. Even if all of the data is in the buffer cache, the latency of a read call will vary according to the size of the read request; it takes longer to transfer one megabyte of data from the buffer cache than to transfer one kilobyte. Requests for data that is not in the buffer cache will also see performance that varies. In this case, performance depends not only on request size, but also on the access pattern. Many file systems optimize file layout for sequential access. Thus a series of reads that request sequential ranges of data from a file will often perform better than a series of reads that request data from random locations in the same file. All of these factors mean that a comprehensive suite of microbenchmarks requires multiple measurements for some system calls.

Another drawback of the one-benchmark-per-system-call approach is redundancy. Many file system calls have overlapping functionality. The truncate call frees

all storage allocated to a file. Removing a file also frees all of a file's storage in addition to freeing its inode and clearing its directory entry.

In an effort to avoid redundancy, I found several important overlaps between the various calls in the file system interface. I also found several scenarios where system calls might need to be measured with a variety of arguments or with the system in different states. Table 4.3 lists the system calls, identifying those that share functionality and those that require similar measurement techniques.

4.4.3.2.1 Pathname Resolution

As shown in Table 4.2, many of the calls in the file system interface take a pathname as one of their arguments. In processing each of these calls, the file system must parse the specified pathname to determine the target file for the operation. Benchmarking each of these system calls independently would redundantly measure the overhead of pathname parsing and would require each benchmark to account for how the performance of the target system call varies with the length of the pathname argument. Instead, HBench-FS includes a separate microbenchmark for measuring the time the file system takes to resolve a single pathname component.[3] HBench-FS uses this program to measure the time to perform a single lookup under a variety of circumstances, with the name either in the name cache or not, with the attributes for the target file either in the attribute cache or not, and when a series of lookups occur in the same directory or when they occur in different directories.

HBench-FS then benchmarks the file system calls that take a pathname as an argument under controlled circumstances, in order to minimize or eliminate the overhead of pathname processing. HBench-FS measures the performance of each of these calls using short (one or two component) pathnames. The tests also ensure that the

3. To measure the latency of a single pathname lookup, the microbenchmark performs a series of *access* calls. The first call accesses a directory on the test file system. This should bring all of the data needed to resolve the pathname into the cache. A second call measures the time to access the same directory with all of the names and directory attributes in the caches. A third call measures the time to access a file in that directory. This call must do all of the work of accessing the directory, plus the additional work of resolving one pathname. Thus, the difference in the two measurements represents the time to resolve one pathname component.

pathname data is already loaded into the name and attribute caches before they measure the operation under test.

Thus, to predict the latency for any of these pathname parsing system calls, HBench-FS combines the results of the name lookup benchmark with the result of the relevant system call benchmark, as demonstrated in the example in Section 4.4.1.

System Call	Pathname Resolution	Request Size	Access Pattern	Cache State	Truncate	Get Attr	Set Attr	Async
open	✓				✓			
creat	✓							
unlink	✓				✓			
read		✓	✓	✓				
write		✓						✓
close								
truncate	✓	✓		✓	✓			
ftruncate		✓		✓	✓			
mkdir	✓							
rmdir	✓				✓			
getdirentries		✓	✓	✓				
stat	✓					✓		
fstat						✓		
access	✓					✓		
rename	✓							
chmod, chown, chgrp, utimes	✓						✓	
fchmod, fchown							✓	

Table 4.3: Characteristics of File System Calls. This table summarizes common characteristics of the frequently used file system calls. Each column corresponds to a piece of functionality or an aspect of behavior shared by one or more of the calls. The *Pathname Resolution* column indicates calls which must translate one or more pathname arguments to determine the file(s) they operate on. The *Request Size* column indicates calls whose performance depends on a request size argument provided by the caller. Usually this is the size of an I/O request. Truncate calls specify the request size implicitly via the size of the file being truncated. The *Access Pattern* column indicates calls whose performance depends on the caller's pattern of access (sequential vs. random) across multiple requests. The performance of calls designated in the *Cache State* column depends on whether the target data is in the buffer cache. The *Truncate* column indicates calls that truncate their target file either explicitly or implicitly. The *Get Attr* and *Set Attr* columns indicate calls that read or modify (respectively) the target file's attributes. Calls listed in the *Async* column generate file system activity asynchronous to the system call itself.

4.4.3.2.2 Request Size, Cache State, and Access Pattern

The performance of several file system calls depends on the request size passed to them. *Read* and *write* calls transfer a variable amount of data between the file system and the user. The time to transfer data between the user and the buffer cache will be proportional to the request size, which can vary from just one byte to several gigabytes (or more, on 64-bit architectures). In some file system architectures, the performance of the file system varies depending on the size of the file. As we saw in Chapter 3, the overhead of reading the first indirect block in FFS can have a substantial impact on file throughput.

HBench-FS accounts for this relationship between request size and latency in two ways. For cached reads, and all writes (which HBench-FS assumes will be written to disk asynchronously), HBench-FS assumes that performance can be expressed by Equation 3, which only requires two benchmark measurements, one to measure the basic overhead of the *read* and *write* system calls, and another to measure the rate at which the system transfers data between an application and the buffer cache. HBench-FS actually measures these quantities twice, once for *read* calls and once for *write* calls.

$$\text{Latency} = \text{Syscall Overhead} + \text{Request Size} * \text{Data Transfer Rate} \quad \text{(EQ 3)}$$

For read requests that cannot be satisfied out of the buffer cache, the system call latency includes the time to fetch the desired data into the buffer cache from disk. Because this time depends on a variety of factors, including the file system's layout and clustering policies, and the parameters of the underlying disk, it is difficult to reduce uncached read performance to a simple formula. Instead HBench-FS measures uncached read performance for a variety of file sizes, similar to the file throughput benchmark in Chapter 3 (Section 3.3).

The performance of uncached reads can also depend on the requesting application's access pattern. To account for this, HBench-FS also benchmarks uncached reads with both sequential and random access patterns.[4] Thus, for each read request size

4. Traditionally, all non-sequential file access patterns are referred to as "random," even though few of them are truly random.

that HBench-FS benchmarks, it measures the latency for both sequential and random accesses.

In theory, the *getdirentries* system call would be treated in the same way as *read* and *write* calls, with separate measurements of its system call overhead, transfer rate, and uncached throughput for a variety of request sizes and access patterns. Instead, HBench-FS approximates the performance of *getdirentries* using the microbenchmark results for the *read* system call.

4.4.3.2.3 File Truncation

The *truncate* and *ftruncate* calls are two more cases where performance may be a function of request size. In this case, the request size is not actually passed to the file system as an argument, but rather is implicit in the size of the file being truncated. The performance of these calls depends on the size of the target file. As with the benchmarks for uncached reads, HBench-FS measures the time to truncate files of a variety of sizes.

Truncate performance depends not only on the size of the target file, but also on the state of the buffer cache. Most file systems use some form of indirection (e.g., indirect blocks in FFS) to store pointers to the data blocks of large files. In order to truncate a file, the file system must determine which blocks are allocated to the file so they can be freed. If the file's indirect blocks are not in cache, the truncate call must synchronously read them from disk, significantly increasing the call's latency. So it can accurately capture this behavior, HBench-FS measures truncate performance for files that are both cached and uncached.

The *unlink* system call implicitly truncates its target file, freeing its file blocks as part of removing the file. Rather than separately benchmarking the *unlink* call for a variety of file sizes and cache states, as it does with *truncate*, HBench-FS uses the *truncate* benchmark results to predict the latency of *unlink* calls. HBench-FS determines the cost to remove a file, above and beyond the time to deallocate its storage, by measuring the time to remove a zero-length file. To predict the overall latency of a particular *unlink* call, HBench-FS adds the time to truncate a file of the appropriate size and cache state to this basic *unlink* overhead. Since the *unlink* call takes a pathname argument, HBench-FS also

adds the time to resolve the pathname. HBench-FS handles the *rmdir* call, which removes a directory rather than a file, the same way.

The last place where a truncate operation can occur is during an *open* call. An optional argument to the *open* system call allows the caller to specify that operating system should truncate the target file before opening it. In this case, HBench-FS adds the truncation time to its prediction for the latency of the *open* call.

4.4.3.2.4 File Attributes

Table 4.3 shows that there are a number of file system calls that manipulate a file's attribute data. These calls can be divided into two groups, those that return some of a file's attribute data to the caller (*stat, fstat,* and *access*) and those that change the value of one or more file attributes (*chown, chmod, chgrp,* etc.). Instead of benchmarking each of these calls individually, HBench-FS measures the performance of one call from each group (*stat* and *chmod*) and uses those results for all of the calls in the same group.

4.4.3.2.5 Asynchronous Overhead

So far, this discussion has assumed that the performance impact of each file system request is limited to the synchronous execution time of that operation. For many request types this is true. Other requests, however, impose additional asynchronous performance costs on the file system. Many file systems, for example, optimize write performance by implementing a write-behind cache. When an application issues a write request on such a system, the file system copies the target data from the application to the buffer cache and then control returns to the application, which continues executing. The file system marks the data blocks in the buffer cache as *dirty*, indicating that they must be flushed to disk at some future time. At a later time, either when the file system needs to reclaim the buffer space used by the dirty data, or as part of a regularly scheduled *syncer* task, the file system writes the dirty data to disk. From the application's perspective, the only latency for the write request is the time to transfer the data to the buffer cache, as the corresponding disk writes occur asynchronously. Although these disk writes have no effect on the

latency of the original write request, they can affect the performance of subsequent file system operations, which may find their disk requests queued behind these asynchronous writes.

HBench-FS includes a microbenchmark to quantify this asynchronous cost. This benchmark executes a workload with known performance characteristics concurrently with a task that issues write requests at a fixed rate. HBench-FS varies the rate at which it generates write traffic, and measures the corresponding slowdown of the known workload. This provides a measure of the asynchronous overhead of the write operations. The "known workload" that HBench-FS uses in this benchmark is the hot cache file truncation test, which contains a mix of file read, write, create, truncate, and delete operations. As an example, Figure 4.3 shows the slowdown of FFS performance as a function of write load.

In theory, any type of file system operation could generate asynchronous overhead, and HBench-FS should measure the potential asynchronous overhead of every type of file system operation. In practice, however, write operations are the most

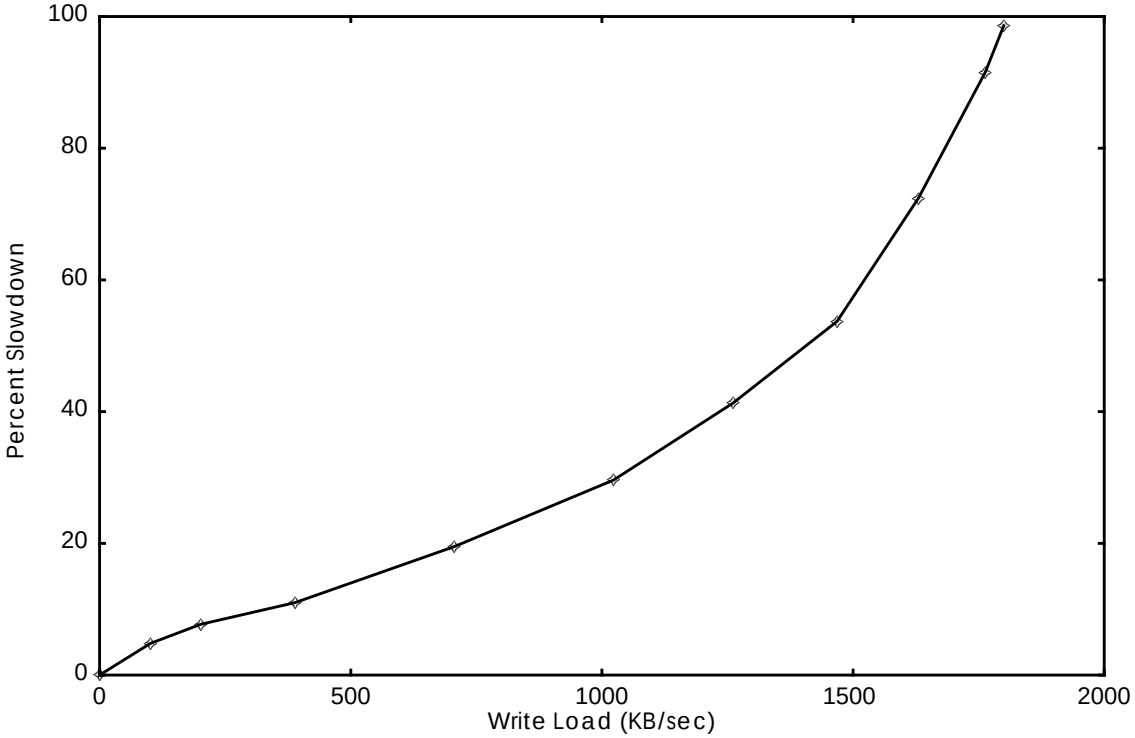

Figure 4.3. Example of Asynchronous Overhead. This graph shows the slowdown of a mixed workload of file system operations as a function of the write load generated by a concurrent process. This data was collected from FFS running on the slow hardware platform described in Section 4.5.1.

common source of asynchronous activity, and that is the only type of operation for which HBench-FS currently measures this overhead.

4.4.3.2.6 Other Microbenchmarks

In addition to the various performance measurements that I have described to this point, HBench-FS includes a handful of measurements that determine the values of various file system parameters, such as the block size and the sizes of the buffer cache, name cache, and attribute cache. Many of the other HBench-FS microbenchmarks use these values. In measuring uncached read times, for example, HBench-FS uses the measured buffer cache size to determine how much garbage data it needs to read from the file system in order to flush all other data from the cache. HBench-FS also uses the values of these file system parameters when it analyzes a workload profile (see Section 4.4.5).

Table 4.4 provides a complete list of the benchmarks included in the HBench-FS system vector.

4.4.3.3 Microbenchmark Design and Implementation

A complete description of the implementation of each microbenchmark is beyond the scope of this chapter. Each benchmark is a stand-alone C program. The various programs share a library that provides functionality for performing common tasks, such as generating test files and initializing the various caches to a desired state. As some benchmarks use the results of other benchmarks, there are dependencies among them. These dependencies are managed using *make* [Oram86] to control benchmark execution.

The benchmarks are designed to try to determine the average latency for each performance characteristic. Therefore, they attempt to vary conditions that might affect performance. Since different areas of disk drives have different performance characteristics, most benchmarks operate on a large number of files deliberately spread over different directories in an attempt to guarantee that the files are distributed across the disk.

Although HBench-FS does not explicitly require it, all of the benchmarks should be run on an aged file system to ensure that their results reflect the performance of the

Name	Description	Units
BC	Buffer cache size	Kilobytes
DNLC	Name cache size	Name entries
MDC	Meta-data cache size	Files
BS	File system block size	Bytes
FS	File system fragment size	Bytes
LUS_BC	Sequential lookup time with both name and attributes cached	Milliseconds
LUS_NC	Sequential lookup time with neither name nor attributes cached	Milliseconds
LUS_AC	Sequential lookup time with attributes cached and name uncached	Milliseconds
LUR_BC	Random lookup time with both name and attributes cached.	Milliseconds
LUR_NC	Random lookup time with neither name nor attributes cached	Milliseconds
LUR_AC	Random lookup time with attributes cached and name uncached.	Milliseconds
RDC	Cached read throughput	Kilobytes / sec
RD1, RD2, RD4, etc.	Sequential read throughput for different size requests (1 KB, 2KB, 4KB, etc.)	Kilobytes / sec
RRD1, RRD2, RRD4, etc.	Random read throughput for different size requests (1 KB, 2 KB, 4KB, etc.)	Kilobytes / sec
WRC	Write throughput	Kilobytes / sec
WR0, WR100, WR200, etc.	Execution time of fixed workload with different amounts of concurrent write load (0 KB / s, 100 KB / s, 200 KB / s, etc.)	Seconds
RDO	System call overhead for read call	Milliseconds
WRO	System call overhead for write call	Milliseconds
OPEN	Open system call time	Milliseconds
CR	Create system call time	Milliseconds
RM	Unlink system call time	Milliseconds
STAT	Stat system call time.	Milliseconds
MKDIR	Mkdir system call time	Milliseconds
RMDIR	Rmdir system call time	Milliseconds
RENAME	Rename system call time	Milliseconds
CHMOD	Chmod system call time	Milliseconds
TR1, TR2, TR4, etc.	Cached truncate time for different size files (1 KB, 2 KB, 4KB, etc.)	Milliseconds
TRC1, TRC2, TRC4, etc.	Uncached truncate time for different size files (1 KB, 2 KB, 4 KB, etc.)	Milliseconds

Table 4.4: HBench-FS Microbenchmarks. This table lists the microbenchmarks used by HBench-FS to characterize the performance of a file system. The name column indicates the keyword associated with each result in the microbenchmark result file. HBench-FS runs some benchmarks repeatedly, varying the file size or background workload. (E.g., RD128 is the throughput reading 128 kilobytes files.) Most of the benchmarks measure the latency of the corresponding file system functionality. The read and write benchmarks measure file system throughput. HBench-FS converts these throughput measurements to latencies when predicting the execution time for read and write system calls.

target file system in real world conditions. All of the results presented in Section 4.5 used aged file systems.

4.4.4 Workload Characterization

A workload characterization has two components, a trace of the file system operations generated when the workload runs and a snapshot of the target file system at the time the trace started.

The snapshot provides a recursive listing of the names of all of the files and directories on the target file system. The snapshot specifies the parent directory and inode number for each file on the test file system.[5] In my workload profiles, I collected the snapshots using UNIX's file listing command, *ls*. The same information could be collected in a variety of other ways, and then converted into the input format HBench-FS requires.

Each file system request in the trace includes a variety of information. Most importantly, all requests include the inode number of the target file. Depending on how the trace is collected, this information may be included in the raw trace, or it may be derived by post-processing the trace to convert pathnames or file descriptors to inodes numbers. HBench-FS also needs many of the original arguments passed to the file system requests. For *read* and *write* calls, it requires the request sizes, as well as the file offset.[6] The trace also includes all pathname arguments. Since pathnames may be specified relative to the calling process's current directory, the inode number of that directory is also needed.

There are a variety of ways of collecting the required trace information. I collected the traces used in this study by instrumenting the BSD/OS kernel. Whenever an application makes a call to the file system, the kernel writes the relevant trace information to an internal buffer. A user-level daemon process periodically copies this

---

5. Throughout this chapter, I use the term *inode* to refer to any unique identifier for a file. In my traces, it is, in fact, the inode number used by FFS. All that HBench-FS requires, however, is that these numbers be unique to each file and that they be used consistently throughout the file system trace and snapshot.
6. In the POSIX definition of *read* and *write*, the file offset is not explicitly provided as a system call argument. Instead it is maintained internally by the file system. If the offsets are not available in the raw trace, they can be computed by post-processing the trace.

data from the kernel to a remote file. This technique is similar to that used by the Linux Trace Toolkit, although the actual trace points and data collected are different [Yaghmour00]. Other tracing techniques can also provide the necessary data. Many operating systems provide tools for tracing all system calls (e.g., *ktrace* on BSD systems). Although these traces seldom provide all of the data HBench-FS needs, it is usually possible to post-process the traces to generate the missing data. Many system calls, for example, specify the target file using a small integer called a *file descriptor*. Taken by itself, the file descriptor is insufficient to identify the target file. If, however, the trace includes the file system operation (usually an *open* or *creat* call) that returned the file descriptor, then we can determine the corresponding file. Appendix A describes the trace format HBench-FS uses for both its input and output.

4.4.5 Workload-Specific Performance Analysis

Section 4.4.3 described how HBench-FS divides file system performance into the various microbenchmark results that comprise the system vector. In order to analyze a workload profile, HBench-FS needs to generate a request vector for each operation in the workload, combine those vectors with the system vector to predict the performance of each operation, and then summarize the performance predictions in a way that is useful to the end-user of HBench-FS. In this section, I examine each of these steps.

For some operations, it is easy to generate the appropriate request vector. Recall that the synchronous cost of a *write* request is expressed by Equation 3. So the request vector for a *write* request is one write system call overhead and N write data transfers, where N is the number of kilobytes of data transferred by the write request. All other elements of the request vector are zero.

Other operations' request vectors are more difficult to generate. An *unlink* operation, for example, consists of one unlink, the appropriate size of truncate, and one or more lookups. For the truncate and lookup operations, however, we need to know whether they hit in the cache or go to disk. In addition, we need to know whether the lookup requests are sequential (i.e., to the same directory) or random (i.e., to different

directories).[7] In order to determine the necessary information, HBench-FS includes several tools that analyze the trace for cache locality and access pattern.

HBench-FS generates its performance predictions using a chain of analysis tools. Each tool makes a pass through the workload trace adding additional information to it. A cache simulator, for example, annotates each operation to indicate whether it hits in the cache or not. The tool chain is implemented as a UNIX pipeline, and each tool reads and writes the trace in a fixed format. The tools share a library of functions for manipulating traces. This makes it easy to add additional tools as needed. For example, a custom preprocessing tool can be added at the beginning of the chain to convert other trace formats into the format expected by HBench-FS. Similarly, additional tools can be added at the end of the chain to perform custom analysis of the performance predictions.

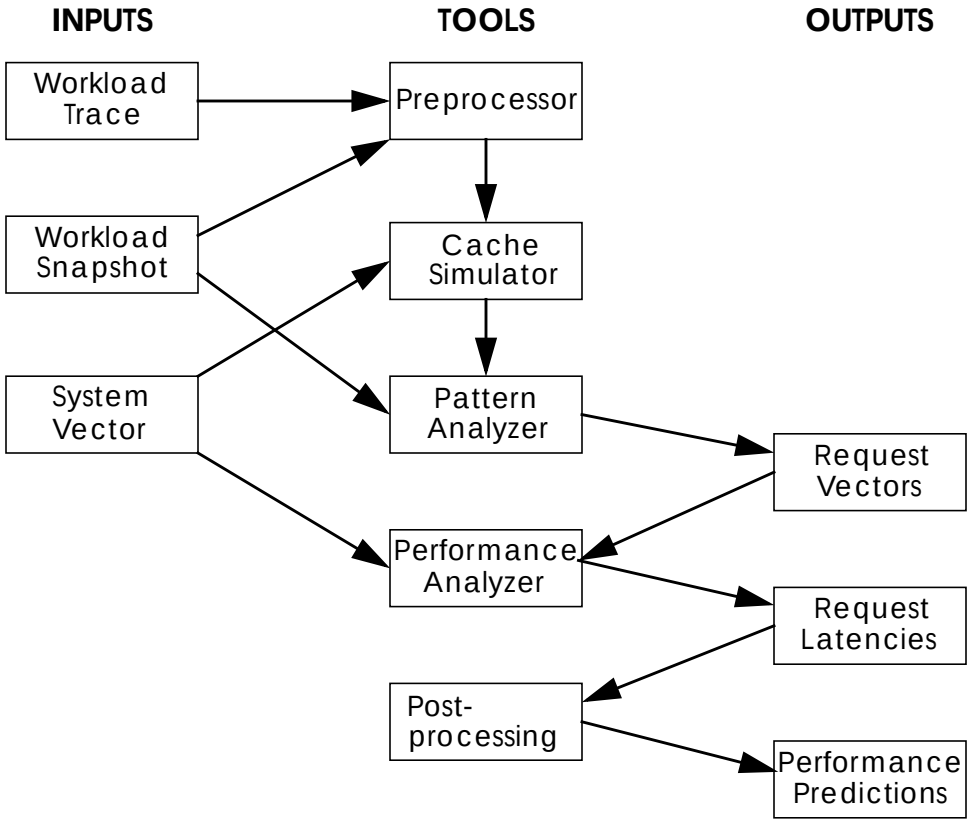

Figure 4.4. HBench-FS Tool Chain. This diagram shows the interactions between the various HBench-FS analysis tools. The boxes in the left column represent the inputs to these tools. The center column contains the tools. The right column shows the various outputs produced by these tools. The outputs from some tools become inputs to other tools.

7. Given a pathname with multiple components, the lookup requests will usually be random. Sequential lookups will occur either when the pathname includes a "." component, or when successive file system requests use pathnames in the same directory.

Figure 4 shows the full tool chain, and Figure 4.5 shows how each of the tools modifies a sample trace. In the following sections, I discuss each of the tools in more detail.

4.4.5.1 Preprocessing

The first thing HBench-FS does with the workload profile is to preprocess it into a format that can be used by the rest of the analysis tool chain. The primary goal of this step is to make sure that all of the information in the workload snapshot is propagated to the appropriate operations in the workload trace. The preprocessor first reads the workload snapshot and builds an in-memory graph representing the file system hierarchy described by the snapshot.[8] The preprocessor then iterates through the requests in the workload trace. Whenever it encounters an operation with a pathname argument, it uses the in-memory image of the file system hierarchy to resolve the pathname and adds the inode number of the resulting file or directory to the request record. The preprocessor updates its in-memory model of the file system hierarchy whenever it encounters a request that creates or deletes a file or directory.

In a typical trace, many commands fail due to incorrect pathnames. In traversing its in-memory image of the file system hierarchy, the preprocessor detects these failures, and annotates the corresponding file system requests to indicate where the failure occurred. With this information, HBench-FS can generate a request vector that includes only the parts of the operation that occur before the file system detects the failure. For example, if an open request has a four component pathname, and the third component is invalid, HBench-FS generates a request vector that only contains three lookup requests. The last lookup and the open do not occur because the file system cannot continue processing the request after the failed lookup.

---

8. The code for building and manipulating the in-memory copy of the file system hierarchy is based on a similar tool written by Diane Tang for her senior thesis [Tang95].

INPUT TRACE:

Open /tmp/file
Read File ID = 123, Offset = 0, Count = 8192
Read File ID = 123, Offset = 8192, Count = 8192
Close File ID = 123

PREPROCESSOR OUTPUT:

Open /tmp/file, **File ID = 123**
Operations: 2 lookups, 1 open
Read **/tmp/file,** File ID = 123, Offset = 0, Count = 8192
Operations: 1 block read
Read **/tmp/file,** File ID = 123, Offset = 8192, Count = 8192
Operations: 1 block read
Close **/tmp/file,** File ID = 123
Operations 1 close

CACHE SIMULATOR OUTPUT:

Open /tmp/file, File ID = 123
Operations: 1 **cached** lookup, 1 **uncached** lookup, 1 open
Read /tmp/file, File ID = 123, Offset = 0, Count = 8192
Operations: 1 **uncached** block read
Read /tmp/file, File ID = 123, Offset = 8192, Count = 8192
Operations: 1 **uncached** block read
Close /tmp/file, File ID = 123
Operations 1 close

PATTERN ANALYZER OUTPUT:

Open /tmp/file, File ID = 123
Operations: 1 cached lookup, 1 uncached **random** lookup, 1 open
Read /tmp/file, File ID = 123, Offset = 0, Count = 8192
Operations: 1 uncached **random** block read
Read /tmp/file, File ID = 123, Offset = 8192, Count = 8192
Operations: 1 uncached **sequential** block read
Close /tmp/file, File ID = 123
Operations 1 close

PERFORMANCE ANALYZER OUTPUT:

Open /tmp/file, File ID = 123
Operations: 1 cached lookup, 1 uncached random lookup, 1 open
Latency = 6.46 ms
Read /tmp/file, File ID = 123, Offset = 0, Count = 8192
Operations: 1 uncached random block read
Latency = 5.88 ms
Read /tmp/file, File ID = 123, Offset = 8192, Count = 8192
Operations: 1 uncached sequential block read
Latency = 0.25 ms
Close /tmp/file, File ID = 123
Operations 1 close
Latency = 0.09 ms

Figure 4.5. Processing a Sample Trace. This figure shows a sample trace, consisting of opening a file (/tmp/file), reading 16 kilobytes from it in two read requests, and closing the file. The data in the trace is shown after each tool in the HBench-FS tool chain processes it. New data added by each tool is shown in **bold face.** The latency numbers in the performance analyzer output were computed using the microbenchmark results from the slow FFS configuration described in Section 4.5.1.

4.4.5.2 Cache Simulator

The cache simulator determines which file system operations hit in the buffer cache, name cache, and attribute cache. The simulator uses the parameters in the system vector to determine the sizes of the three caches and the file system fragment size, which it uses as the unit of granularity for the buffer cache. The simulator assumes that all of the caches are empty at the beginning of the trace, and keeps track of the caches' contents throughout the trace, updating them as it processes each request.

The cache simulator assumes that all three caches use an LRU replacement scheme. It also assumes that all dirty buffers have been flushed to disk and are ready to be re-used by the time they reach the head of the LRU list. The buffer cache simulation only tracks file data blocks ignoring meta-data, such as inode blocks and indirect blocks, which would typically be stored in the buffer cache in an actual file system. The simulator may, therefore, be over-optimistic in predicting which requests hit in the buffer cache. Since meta-data typically makes up only a small portion of the total data on a file system, this should have a minimal impact on the predicted hit rate.

For each *read* or *getdirentries* request that the simulator processes, it determines how many of the requested blocks, if any, are in the buffer cache, and annotates the request to indicate that this number of blocks hit in the buffer cache. When it encounters a request that has a pathname argument, the simulator indicates which, if any, of the name lookups hit in the name and attribute caches. For *truncate, ftruncate,* and *unlink* requests we would like the cache simulator to indicate whether the target file's indirect blocks are in the buffer cache, since the cache state of this data can have a significant effect on performance. (See Section 4.4.3.2.) Because the cache simulator does not keep track of this information, HBench-FS approximates the cache state of indirect blocks by assuming that if a file's inode and first and last data blocks are cached, then the indirect blocks are also cached. While this heuristic may not always be accurate, in practice it works well.

This tool is based on a buffer cache simulator written by Mary Baker to analyze the benefits of adding Non-volatile RAM (NVRAM) to computer systems [Baker92]. I

modified her code to eliminate the NVRAM simulation and to add support for the name and attribute caches.

4.4.5.3 Pattern Analyzer

As described in Section 4.4.3.2, HBench-FS's system vector differentiates between sequential and random access patterns. When analyzing a workload trace, the pattern analyzer annotates individual file system operations to indicate which access pattern they follow. Because performance usually depends only on the access pattern of requests that miss in the cache and go to disk, this tool ignores all requests that are satisfied out of the cache. Thus when we talk about a request being sequential, we mean that it is sequential relative to the previous request that generated disk I/O.

There are two request types that we analyze for access patterns, file reads (and the comparable getdirentries operation for directories) and the lookup operations associated with requests that include a pathname argument. Analyzing reads is simple. If two disk-bound read-requests are to successive ranges of the same file, then the second request is considered sequential. All other read requests are random. Successive lookup requests are considered to be sequential if they are looking up files in the same directory.

4.4.5.4 Performance Analyzer

The final component of the tool chain in the performance analyzer. This tools computes the predicted latency for each operation in the workload trace. It also aggregates data about the asynchronous costs of the write requests in the trace.

For each operation in the workload trace, the performance analyzer computes the dot product of the operation's request vector and the target file system's system vector. The result is HBench-FS's prediction of the operation's latency on the target file system. The performance analyzer's output is the fully annotated workload trace, including the request vector and predicted latency for each operation.

The analyzer also computes the total write throughput for the workload along with the file system utilization. The throughput computation uses the total run time of the workload (from the timestamp of the first operation to the timestamp of the last

operation) as the divisor. The file system utilization represents the fraction of the workload's runtime when the file system is active. It is computed by dividing the total latency of all the file system operations by the total run time of the workload. If the computed file system utilization is greater than one (as can happen if the target file system is slower than the system from which the trace was collected), then the performance analyzer uses a value of one. The performance analyzer uses these values to predict the asynchronous overhead of the write requests in the workload.

The performance analyzer interpolates the write overhead data included in the system vector to determine the amount of slowdown associated with the expected throughput level. It then predicts the asynchronous overhead using Equation 4.

$$\text{Asynchronous Overhead} = \text{Slowdown} * \text{Total Latency} * \text{Utilization} \quad \text{(EQ 4)}$$

The intuition behind this equation is that the estimated slowdown factor multiplied by the total latency of all file system operations predicts the cost of all of the asynchronous activity associated with the workload. Not all of this overhead will effect the operations in the workload, however. If the file system (and hence the disk) is idle half the time, then we would expect roughly half the asynchronous operations to occur when the disk would otherwise be idle. Hence we scale the total overhead by the utilization.

Ideally, we would like to be able to assign the asynchronous overhead to the individual operations in the workload trace that will suffer from this additional latency. In practice, this is impossible without empirically determining the system's write behind and disk scheduling policies. Since this would add a considerable amount of additional complexity, the performance analyzer simply reports the write and meta-data update overheads in addition to producing a trace annotated with the predicted latencies of each operation.

For example, consider a hypothetical workload being analyzed on the test platform whose write overhead is shown in Figure 4.3. Assume that this workload has a total latency of 10 seconds spread over a run time of 25 seconds and 500 kilobytes per second of write traffic. Using the write overhead data, the pattern analyzer would

estimate that this write load would result in a 13.9% slowdown. Multiplying this slowdown factor by the total predicted latency, the analyzer would predict the total overhead from asynchronous write traffic to be 1.39 seconds. The utilization of the system is only 40%, however (10 seconds of predicted latency divided by 25 second of runtime), so the performance analyzer would scale the total asynchronous overhead by this amount, and predict that the additional latency from asynchronous write traffic would be 0.556 seconds.

4.4.5.5 Postprocessing

The output from the performance analyzer, described in Section 4.4.5.4, is the official HBench-FS output. By providing the complete, annotated workload trace, HBench-FS allows the user to perform whatever additional analysis or summarization of the results she wishes. The user, for example, can sum the latencies of all of the individual operations in the trace to get a prediction of the total amount of time the file system will add to the workload's latency. Another useful type of analysis is to sum the latencies by request type. This provides information about the types of operations that consume the most time, identifying bottlenecks either in the application or file system.

The trace processing library makes it easy generate custom post-processing tools. It took me approximately five minutes to build a tool to compute the predicted hit rate for the buffer cache.

Figure 4.6 shows sample output from the postprocessing tool used to analyze the HBench-FS predictions in the rest of this chapter. This tool computes the total latency for the trace, including the asynchronous overheads from writes and meta-data updates, as well as the latencies for each operation type. It reports these totals both in absolute time, and as percentages.

4.4.5.6 Understanding the Results

It is important to understand exactly what HBench-FS is predicting. For each operation in the workload trace, HBench-FS predicts the amount of latency that the file system will contribute to the workload. This is subtly different from predicting the actual latency of each operation. The key difference is that HBench-FS does not account for the time a disk-bound request may spend waiting in disk queue for unrelated I/O requests to complete.

Anytime the file system issues a request to the disk, that request may be queued behind other pending I/O operations. These operations can come from one of two sources, concurrent file system operations, or write-behind operations triggered by earlier file system operations. In either case, HBench-FS does not attempt to predict the amount of time a request spends blocked in the disk queue. This prevents HBench-FS from double-billing the same I/O time to two different file system requests.

4.5 Validation

In order to evaluate the predictions HBench-FS produces, I used it to analyze the performance of several workloads on a variety of file systems and hardware platforms. For each workload and test platform combination I compared actual measurements of the work-

```
Operation       Count   Time (sec)      Percent
---------       -----   ----------      -------
OPEN             3758    0.411           3.96%
READ             5500    6.878          66.16%
WRITE            3171    0.028           0.27%
CREATE           2401    1.156          11.11%
REMOVE           2295    0.475           4.57%
MKDIR             202    0.105           1.01%
RMDIR             194    0.040           0.39%
TRUNC               0    0.000           0.00%
GETATTR          6798    0.442           4.25%
SETATTR           715    0.056           0.53%
READDIR          1650    0.805           7.74%

TOTAL           26684   10.396          100.00%

Predicted Write overhead = 0.184 sec.
```

Figure 4.6. Sample HBench-FS Output. This table shows the output of a HBench-FS after processing a sample trace. The raw output has been postprocessed by a simple tool that computes the total latency, and collects some statistics (latencies and number of operations) for each type of operation in the trace. The latency for each operation type is listed by in seconds and as a percentage of the total latency for the workload.

load's performance to the predictions generated by HBench-FS. By selecting workloads that place different demands on the underlying file system, I show that HBench-FS can make accurate predictions in a variety of conditions and that it can help isolate the performance critical operations in each case.

4.5.1 Test Environment

I evaluated HBench-FS using two different hardware platforms each running two different file systems. The two machines were a 500 MHz Pentium III, and a 133 MHz Pentium. I refer to these as the *fast* and *slow* machines. Table 4.5 provides a full description of the hardware configurations. Both machines ran BSD/OS Release 4.1. This operating system is derived from 4.4BSD UNIX [McKusick96] and uses FFS as its native file system.

	Slow Machine	**Fast Machine**
CPU	Pentium	Pentium III
Clock Speed	133 MHz	500 MHz
Memory	64 MB	512 MB
Disk Controller	NCR/SYM825	Adaptec AHA-2940UW
SCSI Bandwidth	10 MB/sec	20 MB/sec
Disk Type	Seagate ST34520N	Seagate ST39102LW
Total Disk Space	4.3 GB	8.5 GB
Rotational Speed	7,200 RPM	10,000 RPM
Average Access (r/w)	9.5 ms / 10.5 ms	5.4 ms / 6.2 ms

Table 4.5: Test Hardware Configuration. This table describes the hardware configurations of the two test machines**.**

Configuration	**Hardware**	**Operating System**	**File System**
Slow FFS	Slow Machine	BSD/OS	FFS
Slow FFS+SU	Slow Machine	BSD/OS	FFS with Soft Updates
Fast FFS	Fast Machine	BSD/OS	FFS
Fast FFS+SU	Fast Machine	BSD/OS	FFS with Soft Updates

Table 4.6: Test configurations. This table lists the hardware platform, operating system, and file system that I use in each of the four test configurations.

BSD/OS includes support for Soft Updates (see Section 2.1.3.2.2). Enabling or disabling Soft Updates support provides the two different file systems that I evaluate. Table 4.6 summarizes the four resulting test configurations.

Both of the test machines had an extra disk that was used exclusively for test file systems. This eliminated contention for the disk head between the test workloads and I/O generated by loading executables and swapping. All of the test file systems were located in a 1.5 gigabyte partition at the beginning of the test disks. Before running HBench-FS, or any of the test programs, I aged the test file systems using the aging workload described in Section 3.2.3. The partition size is slightly larger than the file system from which the aging workload was generated to allow room to create the various files needed by the test workloads.

4.5.2 Methodology

For this evaluation, I select several workloads, each of which stresses different parts of the underlying file system, and execute them on multiple file system platforms, measuring their actual performance. I also trace each workload on one of the platforms. Using this trace, along with file system profiles of each platform, I use HBench-FS to predict the performance of the workload on all of the platforms. Finally, I compare these predictions to the measured performance on each platform.

In predicting the overall performance of a workload, I use the total latency computed by HBench-FS. For workloads that spend most of their time interacting with the file system, it would be sufficient to compare HBench-FS's prediction to the total run time of the workload. Most workloads, however, spend significant amounts of time outside of the file system. Since HBench-FS only predicts the latency generated by the file system, we cannot compare the total execution time for these workloads to HBench-FS's predictions. Instead, I trace the workload on each platform of interest. In addition to the data required by HBench-FS, these traces include the measured latency of each operation. I use this information to determine how much time the workload spends in the file system, and compare this figure to the predictions from HBench-FS.

As described in Section 4.4.5.5, the postprocessing tools for HBench-FS provide a breakdown of the total time spent on each type of operation. I also compare these predictions with measurements from running the workloads on each of the target systems. Here again, I use traces of the workloads running on each platform to determine the actual performance of the system, this time dividing the timing information in the trace by operation type.

4.5.3 Workloads

In evaluating HBench-FS, I present results for several different workloads executing on a range of different file system platforms. By varying the applications, I examine how HBench-FS responds to workloads that place different demands on the underlying file system. By varying the file system, I show that HBench-FS can accurately predict which system(s) offer the best performance for different applications.

In this section, I describe each of the workloads I used to evaluate HBench-FS. Because I need to compare the measured and predicted performance of the same workload across multiple platforms, I need to accurately reproduce the workloads on each test platform. Therefore, I use macrobenchmarks for my test workloads.

4.5.3.1 Kernel Build

The first workload is a traditional compiler workload, a complete build of the generic BSD/OS 4.1 kernel. This workload consists primarily of file read and write operations, as the compiler reads each source file, along with the requisite headers, and writes the resulting object files. The linker then reads all of the object files and writes the kernel binary. There are approximately four file reads for each write. This workload also performs a small number of file create and delete operations.

4.5.3.2 SDET

This is a former SPEC benchmark designed to emulate a typical time sharing workload. It was deprecated as the computing environments shifted from time-shared central computers to networked client and server machines [Gaede99]. Interestingly, the advent of

thin-client computing, which relocates most computations to a central server machine, suggests that this type of workload may again be relevant for evaluating system performance [Schmidt99][Wong00].

The basic SDET benchmark is a script of user commands designed to emulate a typical software-development environment. The script includes commands for editing, compiling, searching, comparing, copying, and spell checking a variety of files. SDET executes one or more copies of this script concurrently, reporting system performance in scripts per hour as a function of the degree of concurrency. For my evaluation of HBench-FS, I consider several workloads with different levels of concurrency, varying from one to ten scripts.

4.5.3.3 PostMark

PostMark is a macrobenchmark designed by Jeffrey Katcher to reproduce the file system activity generated by e-mail, Usenet news, and e-commerce applications at internet service providers [Katcher97]. These workloads typically involve a large number of small (less than 100 kilobytes) files that are in constant flux. To model this workload, PostMark creates a large set of files with random sizes uniformly distributed over a preset range. PostMark performs a fixed number of operations on these files. The operations alternate between creating or deleting a random file and reading or appending data to a random file. For the create operation, PostMark selects the size of the new file at random from the same range of sizes used for the initial file set. When it reads a file, PostMark reads the file in its entirety. For an append operation, PostMark opens the file, seeks to the end, and writes a random amount of data to the file, not exceeding the maximum file size.

The PostMark workload uses 5,000 initial files ranging in size from 512 bytes to 16 kilobytes. The files are distributed over fifty directories on the test file system. The workload consists of 20,000 file operations, as described above.

4.5.4 Evaluation

4.5.4.1 Kernel Build Results

Figure 4.7 shows the measured total latency on each system, and the latency predictions generated using the *Slow FFS* trace for the workload profile. The graph shows that HBench-FS accurately ranks the performance of the four systems, and captures the large-scale performance differences between them. For this workload, switching from the slow machine to the fast machine provides a much larger performance gain than adding Soft Updates to FFS.

In addition to predicting the total latency of the kernel build workload on the four test configurations, HBench-FS also provides a breakdown of how much of that latency is generated by different types of system calls. Figure 4.8 shows the results of this breakdown, along with the actual breakdowns determined from the traces. From this data we can see that the kernel build workload spends most of its time performing file

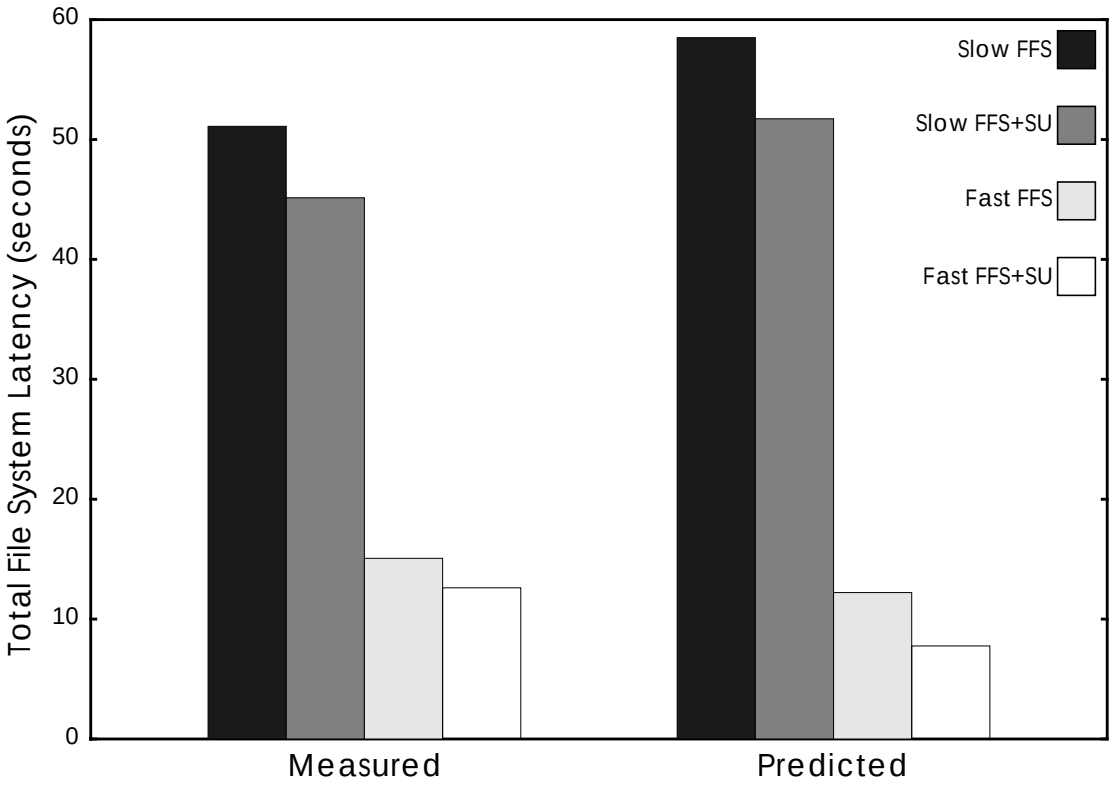

Figure 4.7. Measured and Predicted Kernel Build Performance. This graph shows the measured and predicted performance for the kernel build workload on the four BSD/OS test configurations. The performance metric is the total latency generated by the file system. The predictions were generated using the *Slow FFS* trace for the workload profile.

reads. This accounts for both the large performance improvement when switching from the slow to the fast platform and the relatively modest improvement offered by Soft Updates. The fast platform offers much higher I/O throughput, both for cached and uncached reads. In contrast, Soft Updates only improves the performance of meta-data updates, which do not occur during read requests. In the kernel build workload, the impact of Soft Updates is most apparent in the almost total elimination of time spent performing *create* operations in the *Slow FFS+SU* and *Fast FFS+SU* configurations.

Figure 4.8 also shows that most of the error in HBench-FS's predictions for this workload can be attributed to the *read* predictions. HBench-FS overestimates the *read* time for the configurations that use the slow machine, and underestimates the *read* time for the

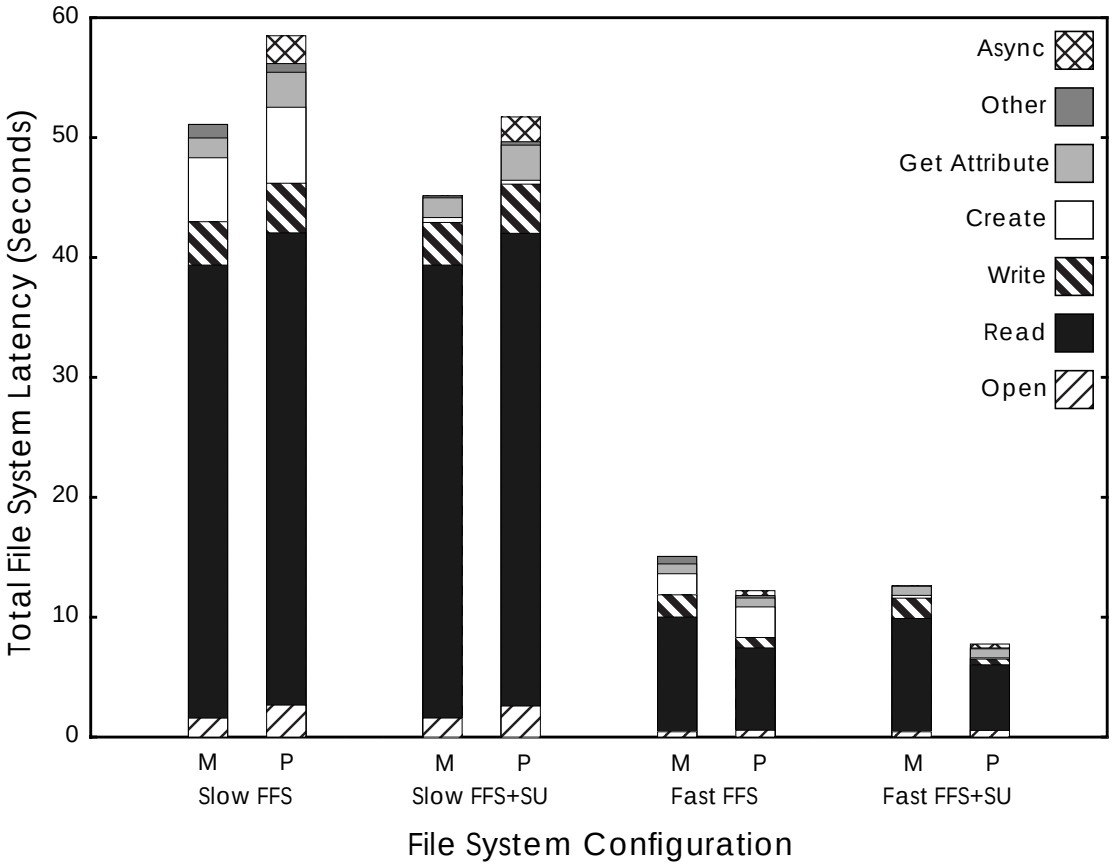

Figure 4.8. Kernel Build Performance Breakdown. This chart compares the measured (M) and predicted (P) performance of the kernel build workload on the four BSD/OS test configurations. The bars are subdivided to show the amount of time consumed by each type of operations. For the performance predictions, I also show the predicted amount of asynchronous overhead from write requests. This overhead is impossible to isolate on the measured systems, and hence the additional latencies caused by asynchronous disk writes are included with the operations that experience them.

configurations that use the fast machine. On the slow machine, HBench-FS also slightly over predicts the time consumed by Get Attribute (*stat*, *fstat*, and *access*) and open operations. On the fast machine, HBench-FS slightly under predicts the time spent during *write* requests.

In addition to predicting the latencies for individual requests, HBench-FS predicts the total additional overhead caused by asynchronous file system activity. In Figure 4.8, this quantity is show separately from the per-operation-type costs in the predicted results. Since it is impossible to isolate the cost of this activity in the measured data, this overhead is necessarily included with the individual requests that incur the added latency. The impact of this asynchronous overhead from write requests is one of the most difficult results to evaluate in HBench-FS's predictions. Since it is impossible to gauge how much this overhead affects the measured performance of operations in the trace, there is no easy measurement against which to compare HBench-FS's prediction. We can, however, examine how the predictions of asynchronous overhead affect the overall accuracy of HBench-FS's results. Figure 4.8 shows that the asynchronous overhead predictions increase the errors for the slow configurations, by increasing the predicted total latency further past the measured latency. On the fast platform, the opposite happens; asynchronous overhead brings HBench-FS's predictions closer to the measured performance in those configurations.

Another way to examine the accuracy of HBench-FS's predictions is to compare the distribution of predicted and measured latencies among all of the individual requests in the workload. Figure 4.9 shows this data for each of the four test configurations. For ease of readability, this data is displayed using a logarithmic scale for latency (on the x-axis). All of the systems show large discrepancies between the measured and predicted distributions for short latency (less than 0.1 millisecond) events, but much smaller differences for longer latency events. Because file system performance is dominated by the longer latency disk-bound requests, HBench-FS produces accurate performance predictions despite its mispredictions for faster events.

To better understand the errors in HBench-FS's predictions, we can examine the distributions of latencies for different types of file system requests. Figure 4.10 shows the distribution of measured and predicted latencies for read, open, and get attribute requests on the Slow FFS configuration. (The distributions for the Slow FFS+SU configuration are nearly identical.) In these graphs, we see two different causes for HBench-FS's mispredictions. For the get attribute and open requests, HBench-FS consistently predicts higher latencies than were measured on the actual system. This indicates that the microbenchmark results for these operations were overly pessimistic. Note, however, that in both graphs, the general shape of the predicted distribution curve closely matches that of the real distribution, indicating that the HBench-FS tool chain is accurately predicting information such as cache hits and access patterns.

The read requests on the Slow FFS platform exhibit a different problem. In the predicted data, there is a sharp jump at approximately six milliseconds. This corresponds to a large number of eight kilobyte uncached random read requests in the trace. Since the

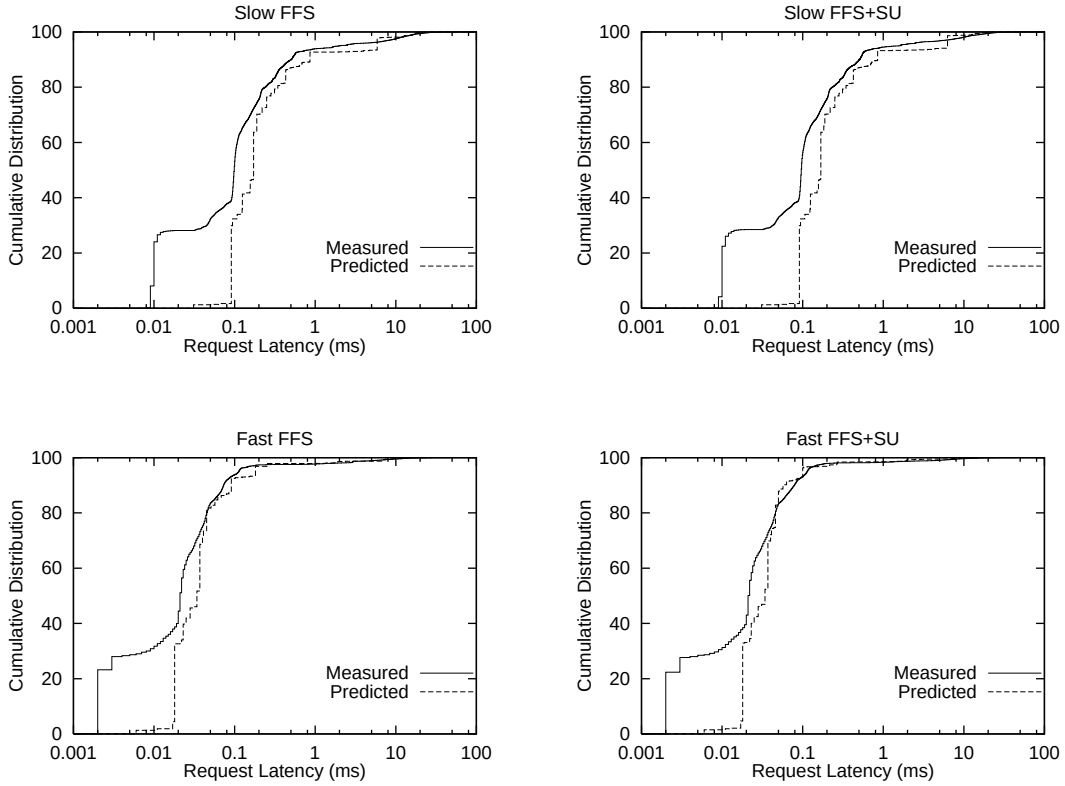

Figure 4.9. Kernel Build Latency Distribution. Each graph shows the cumulative distribution of measured and predicted request latencies for the kernel build workload on one of the four BSD/OS configurations. The x-axis in these graphs uses a log scale.

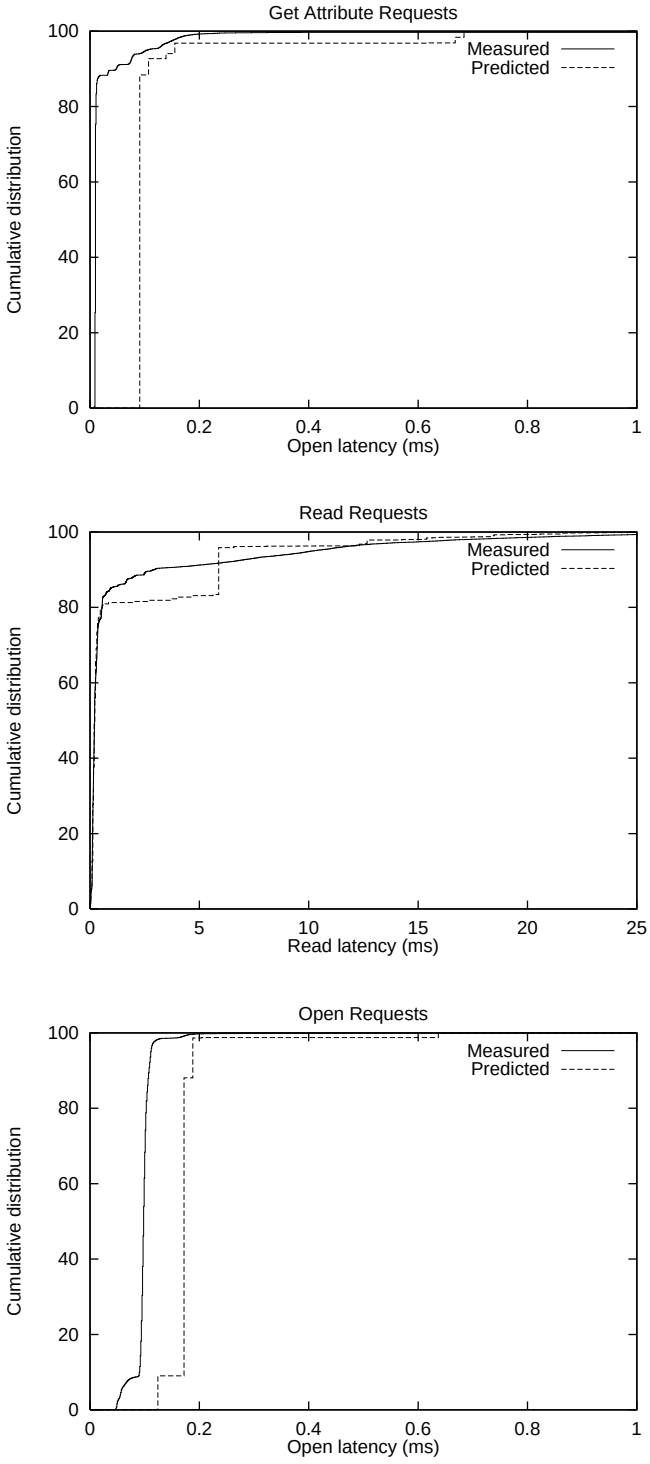

Figure 4.10. Slow FFS Kernel Build—Distribution of Latencies. These graphs show the cumulative distribution of predicted and measured latencies of get attribute, read, and open requests in the kernel build workload running on the Slow FFS configuration. These distributions include 22,308 get attribute requests, 24,801 read requests and 15,010 open requests.

HBench-FS microbenchmarks provide a single measurement for this value, all of these requests are predicted to have the same latency. On the real system these requests experience a range of latencies, as indicated by the gradual slope of the measured read distribution. In this case, it turns out that the sum of the measured latencies is slightly less than the sum of the predicted latencies, resulting in HBench-FS's over prediction for read latency on the Slow FFS configuration.

Figure 4.11 shows the distribution of measured and predicted latencies for read and write requests on the Fast FFS configuration. (The Fast FFS+SU configuration distributes are nearly identical.) As with the reads on the Slow FFS, there is a noticeable

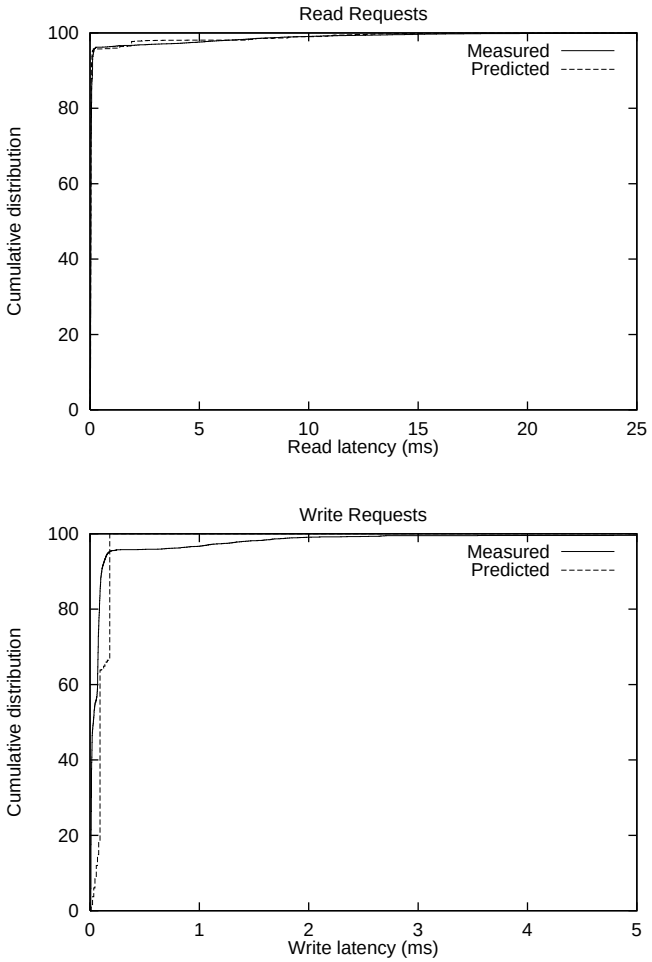

Figure 4.11. Fast FFS Kernel Build—Distribution of Latencies. These graphs show the cumulative distributions of predicted and measured latencies for read and write requests in the kernel build workload running on the Fast FFS configuration. These distributions include 24,801 read requests and 7,447 write requests.

jump in the predicted latencies where HBench-FS predicts the same latency for eight kilobyte uncached random read requests. In this case, the actual distribution includes longer latencies for uncached reads than HBench-FS predicts.

For the write requests on the Fast FFS configuration, HBench-FS predicts relatively fast service times. The longer predicted latencies correspond to the larger request sizes where the system spends more time transferring data between the application and the buffer cache. The measured latencies, however, include a number of longer latencies. The magnitude of these latencies (greater than a millisecond) indicates that the requests are blocking on the disk. These writes occur when the assembler is writing object files. The assembler writes its output in non-sequential order. As a result, it periodically writes data to a block that the file system has already flushed to disk. While this disk I/O is pending, the file system locks the corresponding cache buffer, forcing the subsequent write call to block until the disk request completes. This phenomenon occurs more often on the fast machine than the slow machine because of the greater difference between CPU speed and disk speed on that platform. On the slow machine, the file system flushes buffers to disk in exactly the same point in a sequence of writes, but because of the slower CPU, the disk I/O is more likely to complete before the assembler writes additional data to the corresponding file block. HBench-FS attempts to account for these delays in its aggregate prediction of asynchronous overhead. On the fast configuration, the predicted asynchronous overhead plus the predicted time for write operations is less than the measured time for write operations (see Figure 4.8), indicating that HBench-FS is under predicting the asynchronous overhead associated with this type of background write activity.

4.5.4.2 One Script SDET Results

We turn now to examine the results of the SDET workload, starting with the results of a workload that consists of running just one SDET script. As with the kernel build workload, I use the trace collected on the *Slow FFS* configuration as the workload profile, and use HBench-FS to generate performance predictions for all four of the test platforms. Figure 4.12 shows the measured and predicted total latency for this workload. The graph

also shows how different types of file system operations contribute to these latencies. As with the kernel build workload discussed in Section 4.5.4.1. HBench-FS captures the large scale performance differences between the various platforms. Unlike the kernel build workload, adding Soft Updates to FFS has a larger impact on performance than switching from the slow to the fast hardware platform. The reason for this is readily apparent in the performance breakdown. On the vanilla FFS configurations, this workload spends most of its time performing create and remove operations. Without Soft Updates, these operations require synchronous disk accesses. With Soft Updates, these meta-data updates occur asynchronously, allowing the create and remove requests to execute at

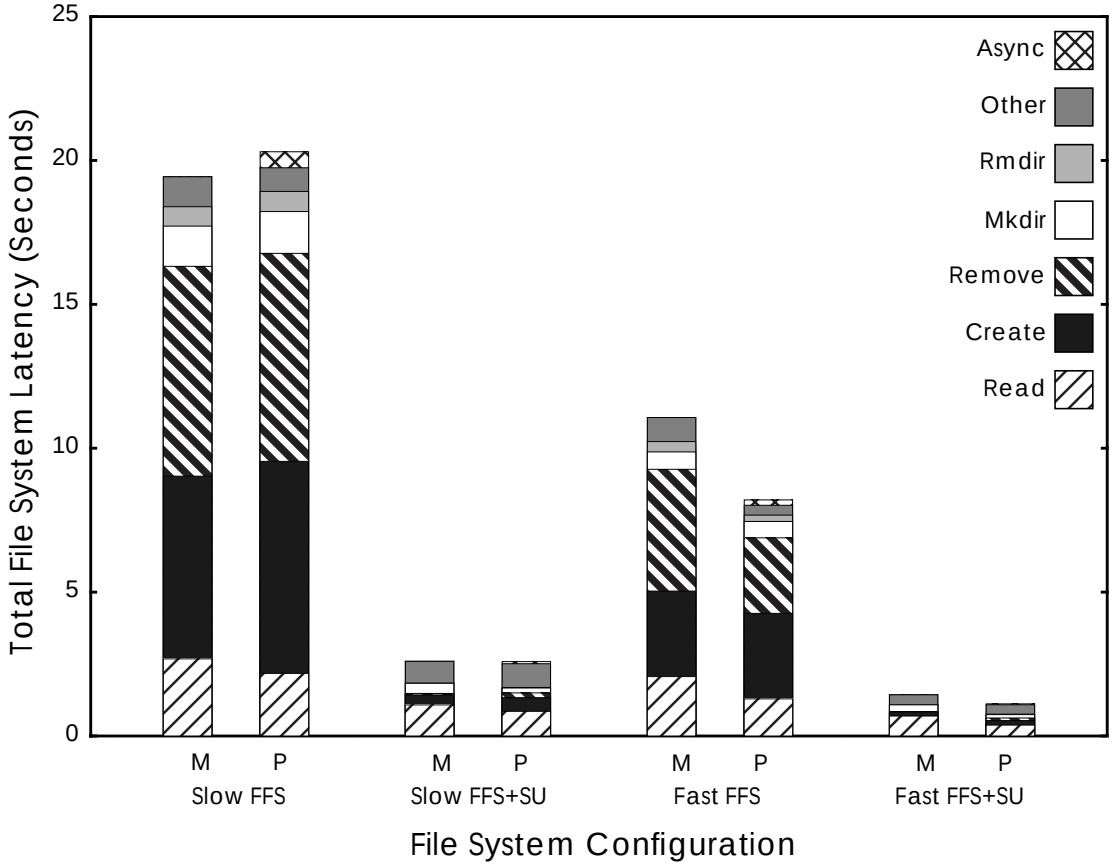

Figure 4.12. One Script SDET Performance. This chart shows the measured (M) and predicted (P) performance of the one script SDET workload. The bars are subdivided to show the amount of time consumed by each type of operation. For the performance predictions, I also show the predicted amount of asynchronous overhead from write requests. This overhead is impossible to isolate on the measured systems, and hence the additional latencies caused by asynchronous disk writes are included with the operations that experience them.

memory speeds instead of disk speeds. The Fast FFS platform also improves the performance of create and remove requests, by providing a faster disk and I/O subsystem, but this is not as large an improvement as that offered by Soft Updates.

Figures 4.12 to 4.15 show the distribution of predicted and measured latencies for the individual requests in the one script SDET workload. Each figure shows latencies of all requests on each of the four platforms and provides additional graphs that show the operations where HBench-FS has the greatest errors. As with the kernel build workload, all of the configurations show that HBench-FS consistently over-predicts the duration of short latency events (less than 0.1 millisecond). These requests are mostly Get Attribute and open requests where all of the required pathname and attribute data is cached. In additional to this discrepancy, the different configurations show different amounts of error for longer latency events. It is these differences that have the greatest effect on the total latency predictions.

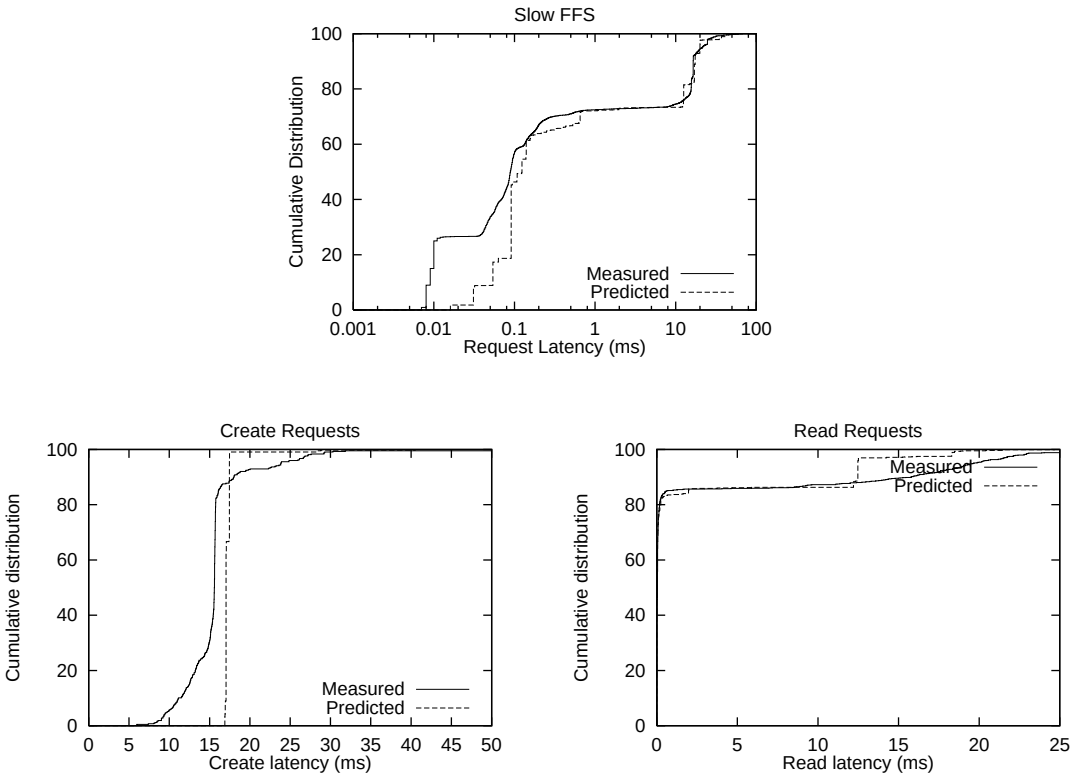

Figure 4.13. Slow FFS latency distributions for one script SDET workload. The top graph shows the cumulative distribution of predicted and measured latencies for all requests in the one script SDET workload. The bottom graphs show the distribution of latencies for creates and reads, the operations that contributed the most to HBench-FS's mispredictions. Note that for readability, the x-axis in the top graph uses a logarithmic scale.

There are several different sources of errors that can be observed in this data. For some operations, such as read requests on all four configurations and create requests on the Slow FFS configuration, HBench-FS predicts a single average performance metric for a class of operations. These cases introduce errors when the actual average performance is different from HBench-FS's prediction. This phenomenon is best illustrated in the create performance on the Slow FFS configuration in Figure 4.13. HBench-FS predicts that almost all create requests will have a latency of approximately 17 milliseconds, with a few operations taking longer due to uncached pathname components. The distribution of measured latencies shows broader range of times, tightly clustered around 15 milliseconds. This workload contains 426 create operations. With an average error of two milliseconds per create, we get a total error of almost a full second, which closely matches the error seen in Figure 4.12.

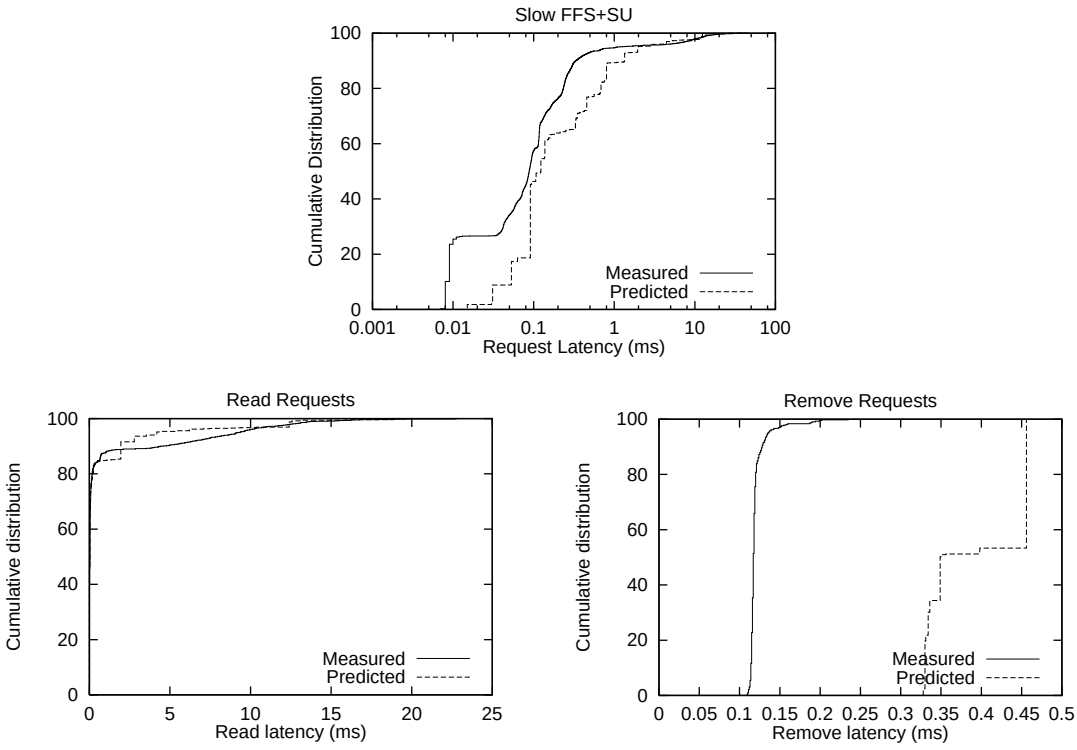

Figure 4.14. Slow FFS+SU latency distributions for one script SDET workload. The top graph shows the cumulative distribution of predicted and measured latencies for all requests in the one script SDET workload. The other graphs show the distribution of latencies for read, remove, and rmdir calls, the operations that contributed the most to HBench-FS's mispredictions. Note that for readability, the x-axis in the top graph uses a logarithmic scale.

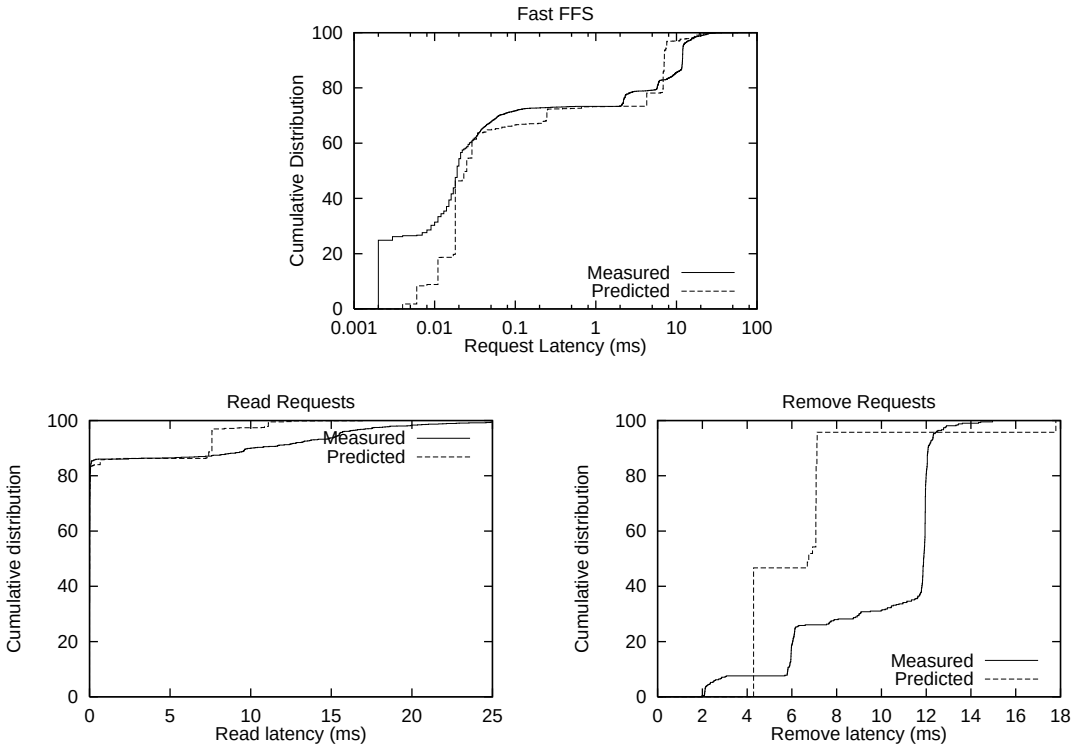

Figure 4.15. Fast FFS latency distributions for one script SDET workload. The top graph shows the cumulative distribution of predicted and measured latencies for all requests in the one script SDET workload. The bottom charts show the distribution of latencies for reads and removes, the operations that contributed the most to HBench-FS's mispredictions. Note that for readability, the x-axis in the top graph uses a logarithmic scale.

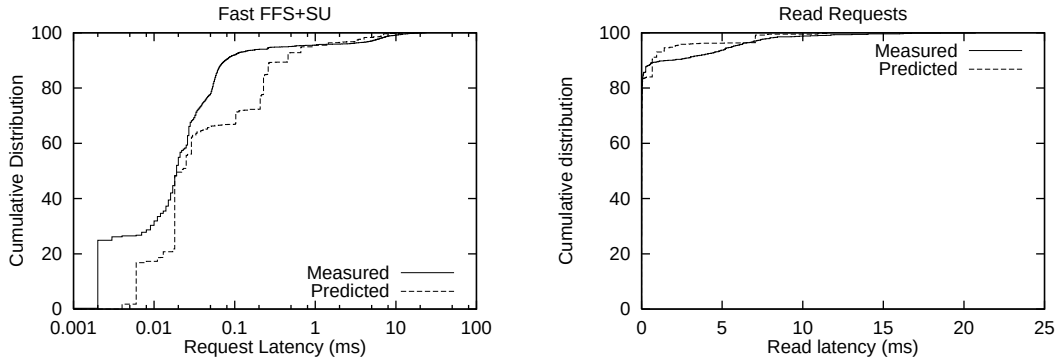

Figure 4.16. Fast FFS+SU latency distributions for one script SDET workload. The left graph shows the cumulative distribution of predicted and measured latencies for all requests in the on script SDET workload. The right graph shows the distribution of latencies for read requests, the operations that contributed the most to HBench-FS's mispredictions. Note that for readability, the x-axis in the left graph uses a logarithmic scale.

On both the Slow FFS+SU and the Fast FFS configurations HBench-FS has noticeable errors in its predictions for file remove operations. The latency distributions are shown in Figure 4.14 and Figure 4.15, respectively. On the Slow FFS+SU configuration, the average remove latency predicted by HBench-FS is considerably higher than any seen in the measured data. Since both the predicted and measured latencies are quite small this difference has a minor impact (0.115 milliseconds) on the overall performance prediction.

HBench-FS incurs much larger errors for the remove requests on the Fast FFS configuration. Because file remove operations execute at disk speeds, errors here have a significant impact on the overall accuracy of HBench-FS's performance predictions.

4.5.4.3 Multiple Script SDET Results

As described in Section 4.5.3.2, the SDET benchmark allows the user to vary the number of scripts that it executes concurrently. I generated additional workloads using up to ten concurrent SDET scripts. Figure 4.17 shows the measured and predicted total latencies for these workloads. In general, the performance here is similar to the one script workload, and HBench-FS predicts the important performance differences. In the eight and ten script workloads, however, HBench-FS incorrectly predicts that the Fast FFS configuration will outperform the Slow FFS+SU configuration. In all of the SDET workloads, HBench-FS slightly under-predicts the latency for the Fast FFS configuration, and slightly over-predicts the latency for the Slow FFS+SU configuration. As the number of concurrent scripts increases, the measured performance of these two platforms converges. In the eight and ten script workloads, the HBench-FS errors are large enough to change the ordering of these systems.

4.5.4.4 PostMark Results

The final workload that I examine is the PostMark benchmark. Figure 4.18 shows the measured and predicted latency for this workload, as well as the contributions of different request types to this latency. Unlike the kernel build and SDET workloads, HBench-

FS consistently under-predicts the performance of PostMark across all of the test platforms. This is largely due to substantial errors in the predictions for the amount of time spent processing read and write requests.

Figure 4.19 shows the distribution of request latencies for the entire workload, and for reads and writes on the Slow FFS configuration. (These distributions are similar

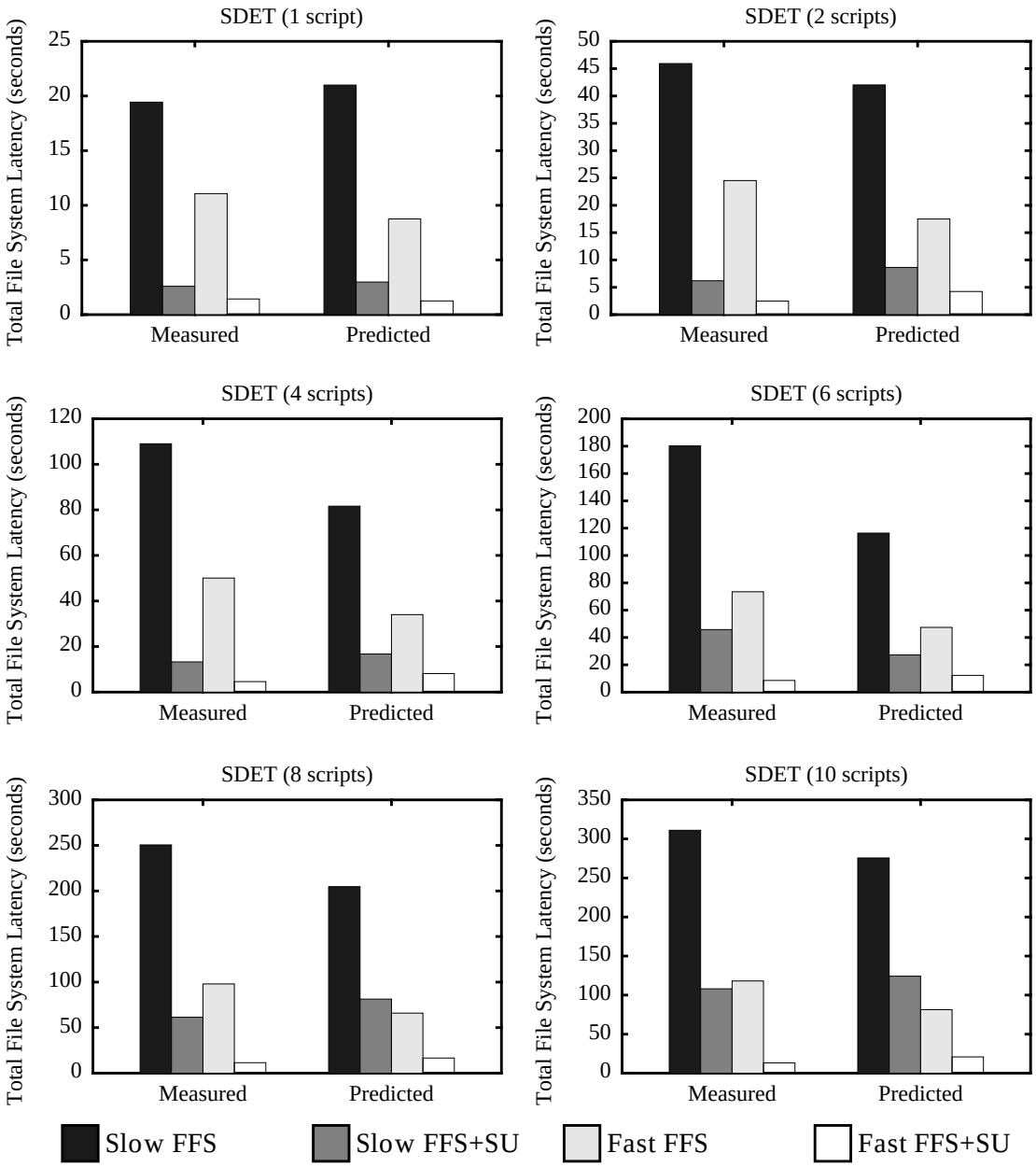

Figure 4.17. Multiple Script SDET Performance. These graphs show the measured and predicted performance for SDET workloads with different levels of concurrency (from one to ten).

on all of the other test platforms.) While HBench-FS predicts the approximate shape of the overall latency distribution, it consistently under-predicts the latency of disk-bound requests (i.e., requests with latencies of more than one millisecond). Almost all of these errors occur on read and write requests, where measured latencies are sometimes as much as 100 milliseconds longer than HBench-FS's predictions, as can be clearly seen in the latency distributions for those request types.

There are two causes for these discrepancies. The primary cause is that HBench-FS underestimates the asynchronous overhead caused by write traffic. This workload writes 95 megabytes of data to the test file system, compared with 24 megabytes for the kernel build workload, and 12 megabytes for the ten script SDET workload. Furthermore, since much of this data is written to small files, there is a large amount of internal

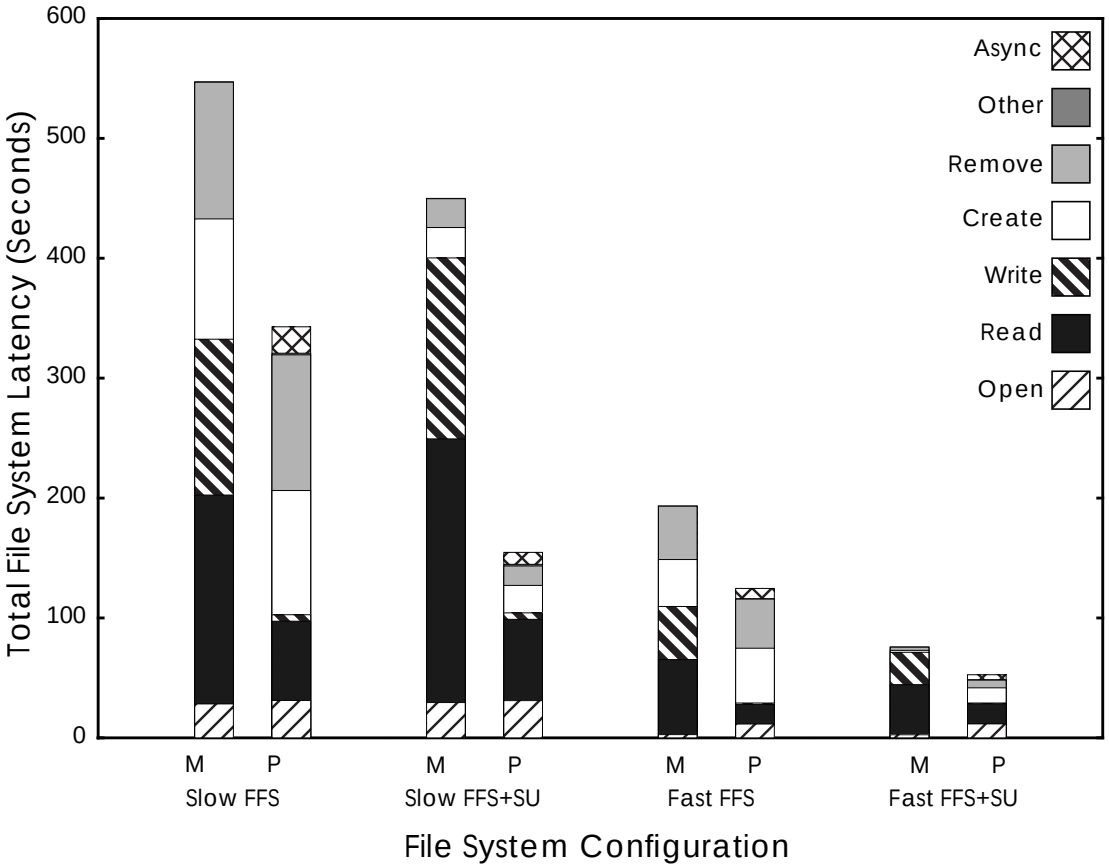

Figure 4.18. PostMark Performance. This chart shows the measured (M) and predicted (P) performance of the one script SDET workload. The bars are subdivided to show the amount of time consumed by each type of operation. For the performance predictions, I also show the predicted amount of asynchronous overhead from write requests. This overhead is impossible to isolated on the measured systems, and hence the additional latencies caused by asynchronous disk writes are included with the operations that experience them.

fragmentation, and the file system actually writes far more data to the disk. In fact, disk writes outnumber disk reads in the PostMark workload by a ratio of approximately eight to one. In the other workloads, disk reads outnumber disk writes. The net result of all this write traffic is to impose significant latency on other disk-bound requests. Figure 4.18 shows that HBench-FS does predict some asynchronous overhead for the PostMark workload, but not nearly enough to account for the large differences between the measured and predicted time spent performing read and write operations.

Another phenomenon that accounts for the some of the differences in the write latencies on the PostMark workload is *partial block writes*. When an application appends data to the end of a file, the last block of the file may not be completely full. In this case, the file system will need to add the new data to the existing file block. If that block is not in memory, the file system will synchronously read it from disk. Unfortunately, this phenomenon occurs often in the PostMark workload, and HBench-FS's cache simulator

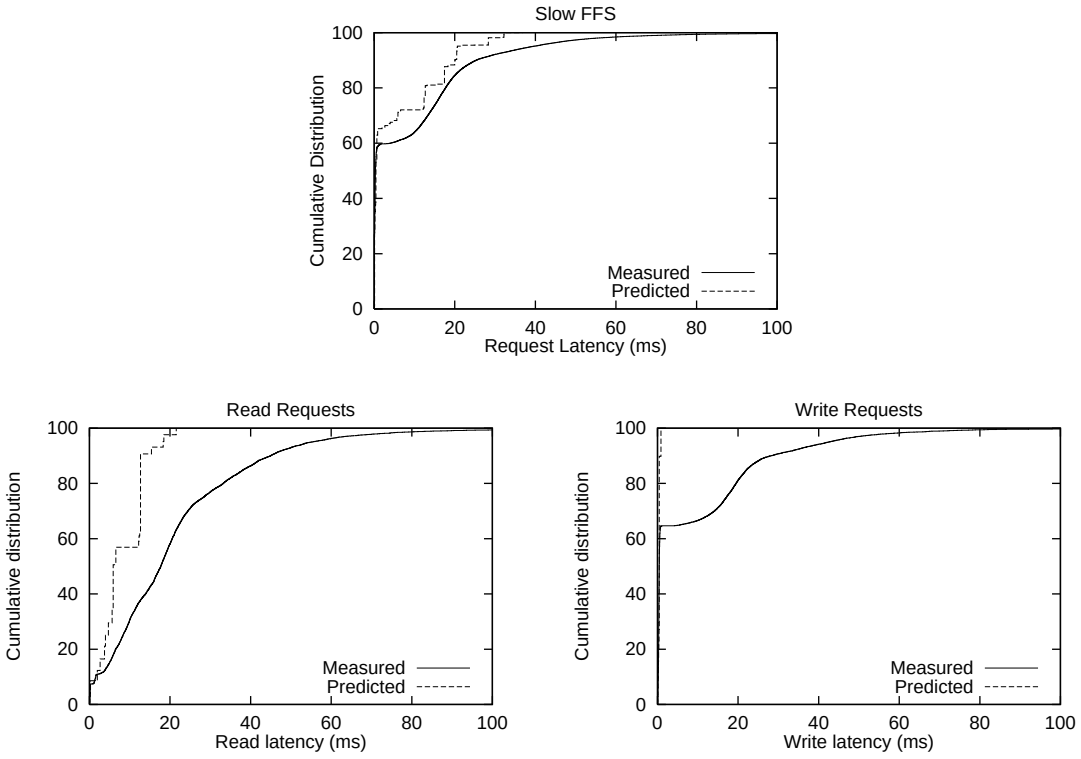

Figure 4.19. Slow FFS Latency Distributions for PostMark Workload. The top graph shows the cumulative distribution of predicted and measured latencies for all requests in the PostMark workload. The bottom graphs show the latency distributions for read and write requests, the operations that contributed the most to HBench-FS's error. The distributions for the PostMark workload on the other test platforms are similar to those seen here.

does not charge the corresponding write requests with the cost of reading the partial block. This accounts for the fact that HBench-FS predicts that all of the writes in this workload will execute at memory speeds. The real latencies show that approximately 35 percent of the write requests have to go to disk. The extremely long latencies seen by some of these write requests is due to additional overhead from write-back traffic.

4.5.4.5 Summary of Results

In the previous sections, I've described HBench-FS's predictions on a case-by-case basis. Now I examine all of these results together in an attempt to understand how well HBench-FS predicts file system performance.

One of the primary goals for HBench-FS is to allow users and researchers to compare how a workload will perform on different file systems. For these predictions to be useful, it is important that HBench-FS properly rank the performance of the test systems. The ideal benchmark, given two file systems and a workload, would always correctly predict which file system would perform best for that workload (i.e., would cause the least latency). Combining the data from the three workloads and four test platforms I studied, there are 48 such comparisons we can make. In all but two cases, HBench-FS correctly predicts the faster file system. The exceptions are the comparisons of the Slow FFS+SU and Fast FFS configurations with eight and ten script SDET workloads. (See Section 4.5.4.3.) This gives HBench-FS an accuracy rate of 95.8%.

It is worth noting that the two mispredictions occurred in cases where the relative performance difference between the two systems was small (32% for the eight script SDET workload, and 9% for the ten script workload). While cases such as these, where the real performance difference between systems is small, are the hardest to predict correctly, the negative consequences of mispredicting the performance ordering for them is also the smallest. When the performance difference between two systems is small, other factors, such as price, reliability, and usability, become far more important in comparing the systems.

The best we could hope for from a system like HBench-FS would be that each predicted latency be a precise match for the corresponding measured latency. To get an

overall understanding of how closely HBench-FS approaches this ideal, Figure 4.20 plots the measured and predicted latency for each test (i.e., for each workload and file system pair). A correlation coefficient of 0.966 between the performance measurements and predictions indicates a strong linear relationship between them.

Table 4.7 shows the raw data from Figure 4.20, listing the measured and predicted latency for each test. This table also shows the error in each HBench-FS prediction, both in absolute terms and as a percentage of the measured performance. These errors range from 8.1%, for the one script SDET workload running on the Slow FFS configuration to 76.2% for the four script SDET workload running on the Fast FFS+SU configuration. The average error is 30.8% and the median is 31.6%. The root mean square (RMS) error is 35.5%.

Table 4.8 shows the average absolute error for the tests on each of the four file system configurations. The Slow FFS platform provided the least error of the test configurations. The other platforms have higher error rates. This is not surprising since

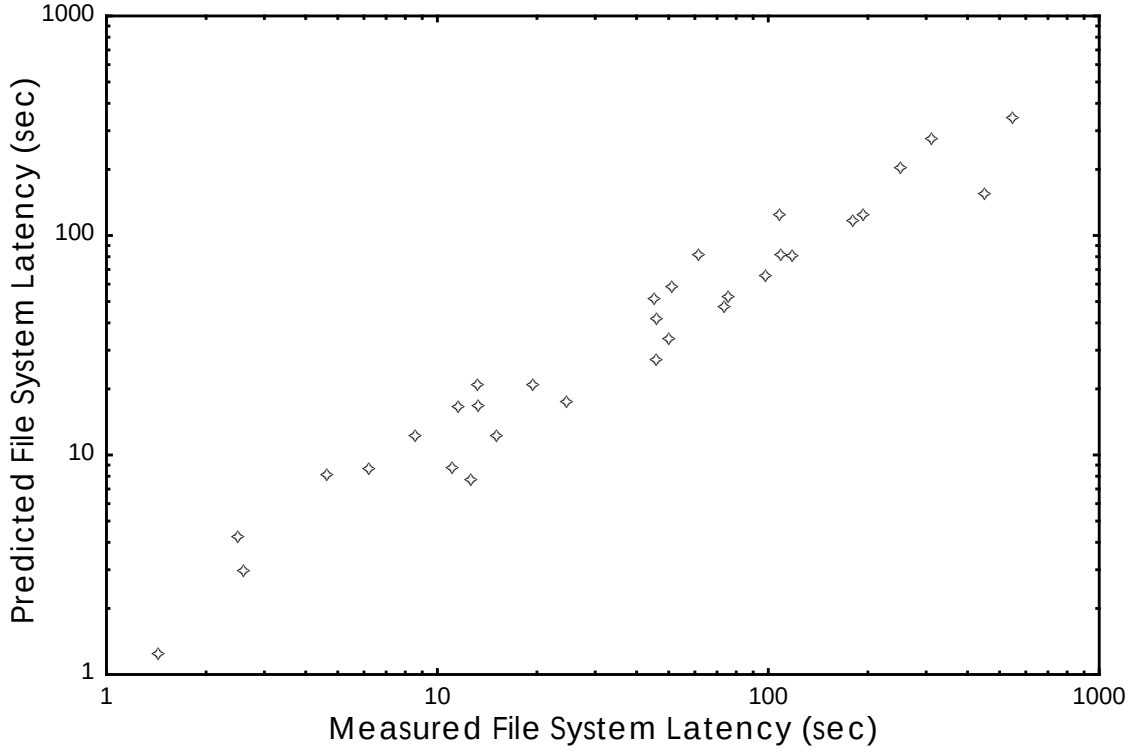

Figure 4.20. HBench-FS Accuracy. This scatter plot shows the predicted and measured latency for each validation test. The data shows a strong linear relationship between the measured performance of each workload and HBench-FS's predictions. (Correlation coefficient = 0.966) Note that both axes use logarithmic scales.

Workload	File System	Measured Latency (sec)	Predicted Latency (sec)	Prediction Error (sec)	Prediction Error (%)
Kernel Build	Slow FFS	51.10	58.49	7.39	14.5
	Slow FFS+SU	45.14	51.73	7.39	14.6
	Fast FFS	15.07	12.21	–2.85	–18.9
	Fast FFS+SU	12.61	7.76	–4.85	–38.4
SDET 1 script	Slow FFS	19.43	21.00	1.57	8.1
	Slow FFS+SU	2.60	2.98	0.39	15.0
	Fast FFS	11.07	8.76	–2.31	–20.9
	Fast FFS+SU	1.43	1.25	–0.18	–12.6
SDET 2 scripts	Slow FFS	45.94	42.02	–3.92	–8.5
	Slow FFS+SU	6.20	8.63	2.43	39.2
	Fast FFS	24.53	17.50	–7.03	–28.7
	Fast FFS+SU	2.49	4.23	1.74	69.8
SDET 4 scripts	Slow FFS	109.04	81.65	–27.38	–25.1
	Slow FFS+SU	13.27	16.75	3.48	26.2
	Fast FFS	50.06	34.02	–16.05	–32.1
	Fast FFS+SU	4.63	8.16	3.53	76.2
SDET 6 scripts	Slow FFS	180.20	116.41	–63.79	–35.4
	Slow FFS+SU	45.79	27.26	–18.53	–40.5
	Fast FFS	73.50	47.41	–26.09	–35.5
	Fast FFS+SU	8.57	12.27	3.71	43.4
SDET 8 scripts	Slow FFS	250.60	204.70	–45.90	–18.3
	Slow FFS+SU	61.44	81.35	19.91	32.4
	Fast FFS	97.98	65.92	–32.05	–32.7
	Fast FFS+SU	11.56	16.68	5.12	44.3
SDET 10 scripts	Slow FFS	311.04	275.65	–35.39	–11.4
	Slow FFS+SU	108.12	124.27	16.15	14.9
	Fast FFS	118.09	81.33	–36.75	–31.1
	Fast FFS+SU	13.22	20.90	7.68	58.1
PostMark	Slow FFS	547.01	342.86	–204.15	–37.3
	Slow FFS+SU	449.89	154.71	–295.18	–65.6
	Fast FFS	193.27	124.62	–68.64	–35.5
	Fast FFS+SU	75.67	52.63	–23.04	–30.4

Table 4.7: HBench-FS Error Summary. This table lists the predicted and measured total latency and the prediction errors for all test workloads and configurations. The predicted and measured latencies are listed in seconds. The errors are listed both in absolute terms, as the difference between the predicted and measured latency (in seconds) and as a percentage of the measured latency.

once of the largest sources of error in the HBench-FS predictions is the overhead cache write-back traffic, and on the faster configurations, the workloads will write data to the file system faster, thus generating more overhead.

Table 4.9 shows the absolute error for each of the workloads averaged across all four test configurations. Again, asynchronous write-back traffic is the primary cause of error, and the workloads with the least write traffic (kernel build and one script SDET). have the smallest errors, and the workload with the greatest amount of write traffic (PostMark) has the largest error.

Test Configuration	Average Error
Slow FFS	19.8%
Slow FFS+SU	31.1%
Fast FFS	29.4%
Fast FFS+SU	46.7%

Table 4.8: HBench-FS Prediction Errors by Test Configuration. This table lists the average absolute prediction error for each of the four test configurations. Each row is the average error for all of the workloads tested on the corresponding file system configuration.

Workload	Average Error
Kernel Build	21.6%
SDET—1 script	14.2%
SDET—2 scripts	36.6%
SDET—4 scripts	39.9%
SDET—6 scripts	38.7%
SDET—8 scripts	31.9%
SDET—10 scripts	28.9%
SDET—all	31.7%
PostMark	42.2%

Table 4.9: HBench-FS Prediction Errors by Workload. This table lists the average absolute prediction error for each of the test workloads. Each row is the average of the errors for all of the file system configurations on which I ran the corresponding workload. In addition to listing the error for each different SDET workload, the “SDET—all” row lists the average error across all of the different SDET workloads.

4.6 Future Work

The major weakness in the current version of HBench-FS is the way it predicts overhead from asynchronous file system activity. In developing HBench-FS, I tried to develop a simple model that would provide a single estimate for the aggregate cost of the background activity generated by write requests. The resulting model provides useful estimates for workloads with moderate amounts of write activity, such as the kernel build and SDET workloads, but it substantially underestimates the impact of asynchronous I/O in workloads, such as PostMark, with large numbers of small write requests. In the current implementation, HBench-FS computes a workload's write throughput using the number of bytes written by all of the write requests in the workload. This assumes that the overhead generated by eight kilobytes of write traffic is the same regardless of whether the workload writes eight kilobytes to a single file, or one kilobyte to eight different files. In reality, the latter case generates eight times as many disk writes. Thus it might be more accurate to compute asynchronous overhead using the number of blocks written instead of the number of bytes written.

Another possible approach would be to develop a more sophisticated model of how these delayed write interact with subsequent I/O requests. If there were a standard write-back policy used by file system buffer caches, I could update HBench-FS's cache simulator to mimic it. Unfortunately, there are a range of write-back strategies employed by different file systems, so I would need to develop a technique to empirically determine the algorithm used by each target file system.

The other area where HBench-FS consistently makes errors is in handling remove requests. Intuitively, the approach of computing the latency of an unlink request using the time to truncate a file plus the time to remove a zero-length file makes sense. Based on the results, however, it might be more accurate to handle removes and truncates separately. To do this I would need to run a series of remove microbenchmarks with different file sizes and update the analysis tools to compute remove latencies using these results.

I could also add extensions to handle a wider range of architectures. I'm most interested in multiple disk systems (RAID or vanilla striping). To support this, I would need to develop a method for modeling the parallelism in a workload, and microbenchmarks for measuring the parallelism available on a system. The other large class of architectures that it would be useful to support is networked file systems. There are two non-trivial challenges in predicting the performance for these systems. HBench-FS would need to model network latencies, which would depend not only on the amount of file system traffic, but also on the possible presence of large amounts on non-file system traffic. The second challenge I would face in supporting client-server file systems is the fact that the workload may be spread across a number of client systems.

There are also a variety of minor improvements that could be added to HBench-FS. One of the most important would be to add support for memory mapped files. In principle, this should be a simple extension of HBench-FS's current functionality. Mapping a file into memory is similar to opening a file. Page in requests correspond to file reads and writes of dirty pages would provide a metric for the amount of asynchronous disk I/O generated by an application. I would need to find a way to capture the paging activity in the input traces. This wouldn't be a problem with my current tracing tools, since I have the sources, but unlike all of the other things I trace, this information could not be derived from standard tracing tools or from most existing file system traces.

There are a variety of minor details and corner cases that HBench-FS does not currently handle, such as hard and symbolic links. I know of no workloads where this functionality has a significant (or even a visible) effect on overall performance, but for completeness, it should be included in HBench-FS.

4.7 Conclusions

File system performance is heavily dependent on the workload that runs on the file system. Different workloads issue different mixes of requests, placing different demands on the underlying file system. HBench-FS provides automated tools that allow an end user or a researcher to predict how a particular file system architecture will perform with a

particular workload. By separating the file system analysis from the workload analysis, HBench-FS allows the user to make these predictions without having physical access to the target system. HBench-FS also automates many parts of performance analysis that researchers traditionally perform by hand, including microbenchmarking of the target platform, and determining how different parts of the file system interface contribute to the overall performance of a workload.

HBench-FS produces predictions of the contribution of the file system to the total latency of a workload, and also predicts how much time each type of file system operation contributes to this latency. By providing fine-grained predictions for each operation in a workload, HBench-FS also allows researchers to perform customized analysis of their workloads.

The predictions produced by HBench-FS are typically accurate when compared to the measured performance of the system workloads and file systems. The predictions of total latency have an average error of 30.8%, and correctly predict which file system will provide the best performance for a particular workload in 96% of the test cases. The major sources of error arose in workloads that generated large amounts of write traffic.

Chapter 5

Related Work

In this chapter I survey other research related to file system aging and workload-specific benchmarking. I first describe other techniques researchers have used to artificially fill or age file systems. I then examine a variety of techniques that have been used to evaluate system performance in the context of specific workloads. Finally, I describe work aimed at developing analytic models of file system performance.

5.1 File System Aging

Chapter 3 described the use of file system aging to accurately reproduce the real-world state of a file system. To date, only a few other file system studies have explicitly evaluated the performance of file systems that were not empty. The existing studies that measure non-empty file system performance use a variety of ad hoc techniques to fill the test file system.

The simplest way to fill a test file system is to create one or more files that consume the desired amount of space. Margo Seltzer and her colleagues used this approach in evaluating the performance of transaction processing workloads on a log-structured file system (LFS), showing that LFS performance declined as free space on the file system decreased [Seltzer95]. For these measurements, Seltzer filled the test file system by creating a single large file of the appropriate size [Seltzer00b].

Other researchers have used stochastic methods to age a file system. Herrin and Finkel tested their Viva file system using an aging technique that created and deleted files at random, selecting file sizes from a hyper exponential distribution [Herrin93]. Ganger and Kaashoek used a similar technique to evaluate their clustering FFS, using the distribution of file sizes seen on their servers instead of a hyper exponential distribution [Ganger97]. In both cases, the authors were interested in understanding how fragmentation affected their file system optimizations. This type of aging is useful for fragmenting a file system, but because the aging workload is not based on real file access patterns, it is impossible to determine whether the amount of fragmentation (and therefore performance degradation) seen in these studies matches what users would see in the real world.

Karl Swartz has used a technique similar to file system aging in his study of Usenet news performance [Swartz96]. Swartz used a benchmark that repeatedly executed many of the time consuming tasks that a news server must perform, unbatching, batching, and expiring news articles. Rather than separately aging his test file systems before running this benchmark, Swartz used the benchmark itself as his aging workload, repeatedly running the benchmark and observing the decrease in server performance as the file system became increasingly fragmented.

5.2 Workload-Specific Benchmarks

HBench-FS evaluates file system performance in the context of a workload of interest. In addition to providing a prediction of overall performance, HBench-FS provides an analysis of which aspects of the file system are the most performance critical for the application. HBench-FS also allows users to perform this analysis without access to the target file system.

Other researchers have used a variety of techniques to provide similar information, ranging from executing the desired workload on the target platform to performing a detailed simulation of the target file system and underlying hardware. In this section I discuss the spectrum of options available, citing prominent examples from the research literature.

5.2.1 Application-Based Benchmarks

The simplest approach to determining how a particular workload will perform on a target file system platform is to execute that workload on the system in question. This ad hoc approach has been used by numerous researchers and has the benefit of providing accurate results with minimal effort. This approach, however, assumes that the researcher has ready access to the target file system and is of little use in the case where the hardware or software is unavailable. Depending on the availability of tracing and profiling tools on the target platform, this technique may or may not provide the kind of detailed performance analysis that is available from HBench-FS. The customized nature of this technique also makes it difficult to compare results from multiple sources, as the workloads are seldom identical.

In an effort to standardize the comparison of workloads across different platforms, a number of researchers and industry groups have proposed standardized benchmarks for common workloads. Jeffrey Katcher's PostMark benchmark [Katcher97], described in Section 4.5.3.3, is an example of this type of benchmark. PostMark is intended to model the file system activity generated by internet server applications such as e-mail, e-commerce, and Usenet news. This benchmark issues requests directly to the file system API, rather than using any of the applications that might generate the workload. This provides a direct understanding of how the file system contributes to performance. Without actually executing any of the target applications, however, there is no assurance that the PostMark workload is actually representative of the file system requests generated by these applications. The documentation for PostMark, unfortunately, provides no argument or evidence that the benchmark accurately captures the file system requests generated by the target internet applications or that it accurately predicts their performance, leaving users to rely on the intelligence and good faith of the author.

Most other application-based benchmarks include the execution of the actual software that generates file system traffic. Typically, these benchmarks are intended as measurements of the overall performance of the application, including the performance

of the file system, network, memory system, and any other parts of the system that the workload exercises. The Transaction Processing Performance Council (TPC), for example, has standardized a family of benchmarks for different types of database workloads, including on-line transaction processing, decision support, and web commerce. These tests measure the performance of a complete database solution, including the database software, the operating system, and the underlying hardware. By carefully specifying the measurement and reporting processes, the TPC benchmarks attempt to provide cross platform performance comparisons for an important class of applications, while minimizing the opportunities for vendors to manipulate the results to their advantage [Burgess].

Other similar benchmarks include those from the Storage Performance Evaluation Council (SPEC) for measuring the performance of web servers, mail servers, and java virtual machines, [SPEC00][SPECa][SPECb] and Usenet news benchmarks created by Yasushi Saito and his colleagues [Saito98].

Standardized benchmarks such as the TPC benchmarks make it easier to compare results achieved by different researchers or vendors, but still provide no generalized way to understand the performance demands of the workload, or bottlenecks of the underlying file system.

5.2.2 Trace-Driven Simulations

Researchers who have wanted to understand the behavior of file system implementations without necessarily building or buying the desired file system have usually used simulations in the place of an actual implementation. The most common simulation studies have been trace-driven cache simulations. These studies are similar to the cache simulation performed by HBench-FS, only instead of using the results of the simulation to predict the overall performance of the workload, they simply report on the performance of the cache, typically in terms of cache events such as misses and write-backs. Early trace-based cache simulations studied the effects of increasing the size of the file system buffer cache or changing its write-back policy [Ousterhout85]. More recent studies have examined a variety of other aspects of cache design, including name and attribute caching

[Shirriff92], cache consistency in distributed file systems [Baker91], the use of NVRAM in caches [Baker92], and a variety of cooperative caching strategies [Dahlin94a][Dahlin94b][Sarkar96].

Researchers have also used trace-driven simulations to study other aspects of file system performance. In general, these simulations are more difficult to develop than the cache simulations described above, since they need to simulate not only the file system cache, but also one or more pieces of file system functionality. The log-structured file system (LFS) architecture [Rosenblum92] has been the topic of several simulation studies. The design of the cleaner in the original Sprite-based implementation of LFS was informed by the results of simulation studies that used random file system workloads [Rosenblum92]. Subsequent studies used trace-based simulations to study alternative cleaning algorithms [Blackwell95][Matthews97].

5.2.3 File System Simulations

In contrast to the time-based performance predictions provided by HBench-FS, the simulations described so far have only provided abstract metrics of performance, such as cache hit rates or, in the case of LFS, write costs. To understand the absolute performance of a simulated file system architecture, researchers need to include the underlying disk system in their simulation. Accurate disk simulators are available, both for individual disks [Ruemmler94][Kotz94a] and for disk arrays [Wilkes96], and several studies have used this approach to predict the performance of various file system architectures.

Whenever a researcher includes a disk simulator in a file system performance study, he faces the problem of translating file-level activity into the disk requests that can be fed to the simulator. In a trace-based cache study, for example, the cache simulator will decide which requests miss in the cache, but this typically provides no information about where the corresponding file blocks may be allocated on disk. Different researchers have used a variety of approaches to solve this problem.

The simplest approaches are to include the on-disk location of files in the traces that drive the cache-simulation or to use a random mapping of files to on-disk locations. Kimbrel and his colleagues used both of these strategies in their trace-based study of file

prefetching and caching policies [Kimbrel96]. More complicated approaches incorporate some or all of the file system code into the simulation, increasing the detail of the simulation as well as the work required to build the simulator. Several researchers have built trace-based simulation environments that provide the scaffolding required to run an entire file system on top of a disk simulator [Thekkath94][Bosch96].

Randolph Wang and his colleagues used a different approach. Rather than simulating the file system on top of a disk simulator, they incorporated a disk simulator into a running operating system. When the test file system issued I/O requests, it sent them to the disk simulator, which determined the appropriate delay for servicing the requests and suspended the requesting process the appropriate amount of time [Wang99]. This approach allowed the researchers to evaluate the performance of a file system that relied on disk features that were not available in contemporary disks.

HBench-FS offers a middle ground between the detailed simulations described above and the cache simulations described in Section 5.2.2. With a simulator that includes the full file system source from the systems under study, researchers can collect arbitrarily detailed performance data by instrumenting the simulator. This, however, requires both access to the file system source code and a detailed understanding of the file system implementation, both of which are unlikely to be available to end users. HBench-FS, in contrast, automatically provides most of the detailed performance data needed by users and researchers.

5.2.4 Machine Performance Characterization

Several researchers have used techniques similar to HBench-FS for analyzing and predicting processor performance. Peuto and Shustek, for example, built an *instruction timing* model for predicting the performance of high-performance (circa 1977) processors such as the IBM/370 and AMDAHL 470 V/6 [Peuto77]. This model used instruction-level traces to predict the performance of application programs. The authors used timing formulas derived from the manufacturers' specifications to predict the latency of each instruction in the instruction set architecture. They also simulated performance critical subsystems, such as the caches, memory interlocks, and instruction prefetch, to account

for the time they add to instruction latencies. Like HBench-FS, this work was motivated by the desire to compare the performance of different systems with similar interfaces (file system interfaces in the case of HBench-FS, instruction set architectures in the case of the instruction timing model). The resulting system accurately predicted processor performance and provided useful analysis of how the execution time was spent. In a recent retrospective, Peuto and Shustek observed that the results of this analysis was one of the lasting and unexpected benefits of their work [Shustek98]. These results included information about branch distances and directions, common instruction pairs, operand overlap, etc., and were one of the lasting, and unexpected, benefits of their work.

The overall design of this instruction timing model is similar to that of HBench-FS. Both systems use a combination of trace-based analysis and fine-grained performance data to predict the performance of both the overall workload and the individual operations that comprise the workload. Unlike HBench-FS, however, the instruction timing model cannot provide cross-platform performance comparisons unless the platforms support the same instruction set architecture. The instruction timing model relies on manufacturer's specifications to derive performance models for individual instructions. The accuracy of the model thus depends on the accuracy and availability of information about processor performance. In contrast, HBench-FS determines system performance using a suite of microbenchmark programs, allowing it to treat the target file system as a black box.

Saavedra and Smith's *abstract machine model* uses a similar approach to predict the performance of Fortran programs [Saavedra96]. They view the execution of a Fortran program in terms of abstract Fortran operations called *AbOps*. Saavedra and Smith summarize the behavior of a program in a *program characterization*, which they generate by profiling the program's execution to determine how many times each AbOp executes. They generate a *machine characterization* using a suite of microbenchmarks to determine the performance of each of the AbOps. In contrast to the instruction timing model described above, Saavedra and Smith do not include a trace-driven simulation of the

memory hierarchy, instead relying in published cache and TLB miss data to account for the impact of the memory system on performance [Saavedra95].

By characterizing application behavior at the programming language level, the abstract machine model allows performance comparisons across systems with different processor architectures. The use of a suite of microbenchmarks to characterize system performance and the vector-based technique for predicting application performance are both similar to HBench-FS. Because they only analyze the performance of widely studied benchmark programs, Saavedra and Smith were able to use previously published cache miss data for their workloads, and did not need to develop trace-based analysis tools used by HBench-FS and in Peuto and Shustek's instruction timing model.

5.2.5 HBench

Several other projects at Harvard University are developing workload-specific benchmarks for a variety computer systems. In all of these projects, the goal has been to provide a framework for analyzing the performance of a specific workload of interest. Like HBench-FS, some of these benchmarks (such as HBench-OS, see Section 5.2.5.1) allow the workload and test system to be measured separately. Other members of the HBench family (such as HBench-Web, see Section 5.2.5.4) provide techniques for distilling complex workloads into benchmarks that can be run on the target systems. All of these projects rely on either the vector-based or trace-based methodologies described by Seltzer and her colleagues [Seltzer99]. HBench-FS is currently the only HBench benchmark to use the hybrid approach of combining vector-based and trace-based benchmarking.

5.2.5.1 HBench-OS

Aaron Brown initially developed HBench-OS to measure the performance of NetBSD operating system primitives on a variety of hardware platforms [Brown97b]. Using the results of this benchmarking suite, he used a vector-based methodology to analyze Apache web server performance on several operating system and hardware platforms [Brown97a]. Brown used the microbenchmark results produced by HBench-OS as his system vector, and used system call tracing facilities to characterize Apache's use of operat-

ing system functionality. Because the components of the two vectors did not match exactly, Brown applied a change of basis to the application vector, mapping each component either to the corresponding microbenchmark result, or approximating it with a related one. Seltzer and her colleagues extended this technique to a collection of standard UNIX utilities (such as *ls* and *tar*) [Seltzer99]. They were able to predict the correct ranking of the systems they tested and achieved reasonable predictions of the performance ratios of the different systems.

5.2.5.2 HBench-Java

Xiaolan Zhang is using vector-based benchmarking to develop workload-specific benchmarks for predicting the performance of Java Virtual Machines (JVMs) [Zhang00]. Zhang identifies four high-level components of a JVM that must be represented in the system vector:

- system classes,
- memory management, including allocation and garbage collection costs,
- the execution engine, including the costs of bytecode interpretation, context switching, etc., and
- just-in-time compilation.

The current version of HBench-Java only measures a small subset of the Java core API. This is sufficient to accurately predict the performance of a variety of large-scale Java applications. Zhang demonstrates the value of workload-specific benchmarking by comparing the predictions of HBench-Java to the results of SPEC JVM98 [SPECa], a standardized workload-independent benchmark. Not only are the SPEC JVM98 predictions less accurate, in several cases they fail to correctly rank the performance of the different JVMs.

5.2.5.3 HBench-JGC

Xiaolan Zhang has also used vector-based benchmarking to evaluate the application-specific performance of JVM garbage collectors [Zhang01]. In this benchmark, she characterizes three aspects of garbage collector performance in the system vector:

- the fixed cost of running the garbage collector,
- the per-object cost of collecting dead objects, and
- the per-object cost of examining live objects.

The fixed cost is parameterized by the heap size, and the per-object costs for live and dead objects are parameterized by object size, similar to the way HBench-FS parameterizes read and write performance by request size. Zhang uses profiling data to generate her application vector.

5.2.5.4 HBench-Web

In contrast to the other HBench benchmarks described in this section, Manley and his colleagues used trace-based techniques in their workload-specific web-server benchmark [Manley98]. HBench-Web analyzes web server logs to generate site-specific stochastic workloads that can be used to measure web server performance. HBench-Web collects three types of data as it processes server logs:

- logical documents, which are collections of web documents that are requested together, typically because they appear on the same web page,
- user session profiles, which characterize the pattern of requests made by different users (i.e., from different IP addresses), and
- the inter-arrival time of new user sessions.

Using this data, HBench-Web generates a stochastic workload mimicking the characteristics of the original workload seen in the server logs. By adjusting the interarrival times users can easily scale the resulting benchmark to model anticipated traffic increases.

Although the fundamental goals of HBench-Web and HBench-FS are similar, namely workload-specific performance analysis, HBench-Web has few other similarities to HBench-FS. Because it does not incorporate vector-based benchmarking, HBench-Web provides no way to characterize the performance of a web server. HBench-FS, on the other hand, might benefit from extensions that allowed the development of parameterized stochastic workloads from file system traces.

5.3 Modeling

Probabilistic modeling techniques, such as Markov chains and queuing networks have been widely used to predict the performance of computer systems, most notably in the field of capacity planning. Like HBench-FS, these models typically combine a workload characterization with a system characterization to provide performance predictions. The strategy in capacity planning is to build and validate a model of an existing system, and then use that model to study the performance impact of increased workload or changes to the system configuration [Lazowska84].

Models of complex software systems typically reduce their behavior to a probabilistic model of the demands they place on the underlying hardware. In modeling a web server workload, for example, a researcher might determine the distribution of request types, requested files, CGI script executions, etc. The researcher would then measure the average CPU time, memory requirements, and I/O operations for each type of request. After validating an initial model against the performance of the actual system, the researcher could estimate the performance of alternate hardware configurations by estimating the effect of the proposed changes on the resource demands of each type of request [Menascé99]. This is a complex process requiring detailed familiarity with the operating characteristics of the system. Adding memory to a system, for example, might decrease the miss rate in the file system buffer cache. To accurately model this effect, the researcher would have to estimate the resulting change in the number of I/O operations for each request type in the workload.

HBench-FS, in contrast, analyzes file system performance in terms of a standard API. Thus, HBench-FS can be used to evaluate the performance of a workload and file system combination without requiring the types of detailed information about the underlying system components that may be required by analytic models. HBench-FS also provides more detailed performance estimates than probabilistic modeling techniques. Where formal models typically represent a workload as a probability distribution of different request types, HBench-FS explicitly models the exact sequence of requests that comprise a workload. This allows HBench-FS to make performance estimates based on

the sequence of requests as well as on the types of requests. While the use of actual workload traces in HBench-FS may allow for more detailed performance estimates, it also has the drawback of making it more difficult to model hypothetical changes in the workload. With a probabilistic model, it is usually a simple matter to increase the request rate or change the request mix in a workload.

Most efforts to model the behavior of file systems have relied on techniques like those described above to reduce the file system to a probabilistic model of the resource demands at places on the underlying hardware system. Shriver and her colleagues are the only group that has attempted to model the internal behavior of a file system. They have developed a simple model for the performance of one or more streams of file read requests in FFS [Shriver99]. This model explicitly incorporates details of the FFS on-disk layout and prefetching strategy, limiting its use to FFS. It is unclear how easily this model could be extended to include the full range of file system activity or, more significantly, whether it is possible to build a model of file system performance that is independent of the internal details of the target file system architecture.

Researchers have focused more effort on developing models for disk performance. Disks are better suited to performance modeling. Unlike file systems, many disks share similar designs and performance characteristics, making it possible to develop models that support a wide range of disks. Some of the most sophisticated models for disk performance have been developed in the past few years by researchers such as Menascé, Shriver, and Barve. Menascé and his colleagues used queuing networks to model the performance of hierarchical storage systems built using network attached devices. Shriver and her colleagues have used *composite device models* to model complex aspects of modern disk behavior, including read-ahead caches and request reordering [Shriver98]. Barve and his associates have extended these models to multiple disks sharing a common I/O bus [Barve99].

Chapter 6

Conclusions

6.1 Lessons Learned

Developing HBench-FS was an iterative process involving repeated tests and improvements to the design. Design improvements were driven by both errors in performance predictions and increased understanding of the performance critical aspects of file system design. In this section, I describe some of the practical lessons learned from this development experience.

6.1.1 Development process

HBench-FS was developed intermittently over a period of several years. Through this time the high-level architecture, combining vector-based and trace-based benchmarking, remained largely unchanged. Many of the details of the system, however, changed substantially, including the quantities measured by the microbenchmark suite and the methods of combining these fine-grained measurements with a workload trace to produce large-scale performance predictions.

The initial design of HBench-FS identified three different types of data that a file system call might manipulate—file data, naming data, and file attributes. Each type of data could be either read or written, might be cached, and could be accessed either sequentially or randomly. This simple taxonomy of file system functionality provided 24

different operations that HBench-FS needed to microbenchmark, such as cached, sequential reads of naming data. In this design, a set of tools, similar to the cache simulator and pattern analyzer in the final HBench-FS design divided each operation in the workload profile into one or more of these basic *file system micro-operations*. In the current implementation of HBench-FS, this strategy is still evident in the way it predicts the performance of read and write system calls.

While the simplicity of this approach was appealing, in practice it ran into a number of difficulties.

- Not all of the micro-operations could be benchmarked independently. Most problematic were operations that read or write name and attribute data. File system operations that operate on naming data almost always read or write the attributes of the named file too, making it impossible to separately measure the cost of naming and attribute micro-operations.
- The cost of performing certain micro-operations varies depending on the higher-level file system operation. In FFS, for example, it takes considerably longer to write file attribute data when you create a file than it does when you change the ownership of a file because the create operation is synchronous and the change ownership call is not.
- The initial model did not account for the asynchronous overhead associated with write operations.

In successive designs, I eliminated the quantities that could not be measured and replaced them with other microbenchmarks, which measured larger pieces of file system functionality. At this point, the idea of modeling the time to perform a file system operation as the time to parse the target file name plus the time to perform the operation itself emerged.

Subsequent improvements involved further refinements to the set of microbenchmarks and processing algorithms. I discovered the need to treat cached and uncached truncate operations (and, more importantly, remove operations) differently. I also added separate benchmarks for creating and removing directories, rather than

approximating the time for those operations using the results of the file create and remove microbenchmarks.

Throughout the development of HBench-FS, I followed a strategy of incremental implementation. The initial implementation of HBench-FS only measured and predicted the performance of lookup and read operations. My initial validation therefore used a workload (running *make depend* on a kernel source tree) that made heavy use of these operations. Over time, I added support for more file system operations to HBench-FS and tested it with increasingly complex workloads. Once I reached the point where I was testing HBench-FS with complex workloads, I continued to make incremental improvements until the predictions attained a sufficient level of accuracy.

Although I knew from an early stage that HBench-FS needed to address the problem of how to account for the overhead of write-back activity, it was actually one of the later pieces of development. In essence, I waited to see if the rest of the system would produce reasonable results before I tackled one of the harder outstanding problems.

6.1.2 Benefits of HBench-FS Architecture

Many aspects of the HBench-FS architecture and my development strategy facilitated the evolution, analysis, and tuning of HBench-FS. While most of these benefits are not surprising in retrospect, many of them came about by happenstance. Only a few software engineering aspects of the development methodology (in particular, the incremental approach to adding functionality) were fully planned.

File systems are complex software systems. They include many different components—caches, prefetching and write-back algorithms, disk layout policies, etc., Each of these components, as well as the interactions between components, can have a substantial impact on performance. Not surprisingly, this complexity is the greatest obstacle to developing a flexible benchmarking tool such as HBench-FS. On first considering the multitude of factors that can affect a file system's performance, it seems that accurate performance predictions must require measurements, simulation, or other analysis of every aspect of the file system. Yet HBench-FS provides accurate performance

predictions despite the fact that it is considerably less complex than the file systems it analyzes.

In practice, it turns out that a relatively small fraction of file system functionality is critical to file system performance. whether because it is frequently executed, such as the read and write system calls, or because of the magnitude of its performance impact, such as the various caches. This phenomenon greatly simplified the development of HBench-FS. By accurately characterizing the behavior of the most performance critical aspects of file system architecture, I was able to quickly develop HBench-FS to the point that the results were promising enough to warrant further investigation.

The iterative development strategy provided a natural way to exploit the fact that different aspects of file system architecture have different importance in determining overall performance. It allowed me to focus my initial efforts on measuring and simulating the performance of a few important parts of the file system, adding additional detail as needed until HBench-FS provided accurate predictions for a wide range of file system functionality. By postponing many minor details of file system performance until late in the development process, I was ultimately able to eliminate them altogether as it turned out that their effect on performance was too small to be worth the effort of modeling them. The amount of accuracy desired in the results determined the amount of detail captured by HBench-FS. For a commercial quality system, rather than a research prototype, more effort would undoubtedly be required.

The underlying strategy of using vector-based benchmarking to determine the performance of individual file system operations proved very flexible, and greatly facilitated the incremental development of HBench-FS. Because vector-based analysis encouraged dividing file system performance into independent components, expanding HBench-FS, or changing the way it handled individual operations, seldom required substantial changes to the bulk of the tool chain. Expanding the system and request vectors (one of the most common changes during development) typically required only the development of a new microbenchmark and the addition of one or two simple functions to the pre-processing and post-processing tools.

A final benefit from the architecture of HBench-FS was the amount of data it provides. This wealth of data not only facilitates the analysis of file system performance, it also proved invaluable in tuning and debugging HBench-FS. During development, as well as during the validation described in Section 4.5, I used workload profiles that included the measured latency of each request in the trace. By comparing the predictions and measurements for both individual operations and different classes of operations, I was usually able to quickly identify the causes of prediction errors. The intermediate traces output by each program in the analysis tool chain provided another invaluable source of debugging information. Because these tools are structured as a pipeline, rather than as a single monolithic program, it was trivial to look at the output from each one to quickly ascertain which tool was the source of a problem. With the wealth of data HBench-FS provides about its predictions and how it computes them, I seldom needed to add instrumentation to the analysis tools to better understand their behavior.

6.1.3 HBench-FS Details

In the course of developing HBench-FS, I uncovered many peculiarities of file system behavior and usage that affect the development of a workload-specific benchmarking tool. In this section I briefly discuss some of these issues.

6.1.3.1 Simulating Partial Block Writes

In Section 4.5.4.4, I describe how partial block writes account for some of the misprediction error in the PostMark workload. Since file sizes are seldom an exact multiple of the file system block size, the final block of a file is seldom full. If this partially empty block is not in the buffer cache when a write operation appends data to the file, then a synchronous read is required to load it from disk. This causes the corresponding write request to execute at disk speed rather than at memory speed. The same phenomenon occurs when overwriting existing data in a file. If a write request is not block aligned, then data at the beginning and/or end of the request must be added to existing data blocks, triggering disk reads if those blocks are not in the buffer cache.

Preliminary versions of HBench-FS captured this behavior in the cache simulator, adding an uncached read to the request vector whenever the cache simulator detected a partial block write to an uncached data block. Unfortunately, *sparse files* make it more difficult for the cache simulator to accurately predict when partial block writes occur. A sparse file is one that has one or more *holes*, extents where no data has been written. A common way to produce a sparse file is to create a new file, then seek to an offset past the beginning of the file, and start writing data. In the resulting file, the extent from the beginning of the file to the target of the seek will contain no data, and in FFS (and most other file system architectures) will have no on-disk blocks associated with it.[1]

The difficulty arises when an application performs an partial block write to a hole in a sparse file. If there is no disk block allocated to the offset of the write, then the file system simply copies the data to an empty cache buffer, and allows the buffer to be written asynchronously to disk. At first glance, it may seem that this must be an unusual scenario, but in practice it occurs regularly, since several common applications write their output in non-sequential order (e.g., the assembler and linker used in the kernel build workload described in Section 4.5.3.1). Thus, in order to know how to handle an unaligned write, the cache simulator must be able to determine whether the corresponding block in the target file has been allocated.

The current implementation of the HBench-FS cache simulator does not model sparse files. Therefore, I was faced with the choice of either simulating reads triggered by partial block writes and over predicting write times in workloads that contain writes to sparse files, or not simulating such reads, and under predicting write times in workloads that contain non-aligned appends or overwrites. I chose the latter option. Although I have not examined the data, I suspect that appends are the most common cause of partial block writes. Therefore, a more balanced approach might have been to include the overhead of reads triggered by appending data to the end of a file, but to ignore those generated by other types of partial block writes. The best solution, of course, would be to

---

1. POSIX semantics dictate that the operating system return zeros when an application reads from a location in a file that has never been written. This applies to holes in sparse files as well as to reads that occur past the end of a file.

modify the cache simulator to accurately model sparse files. This would require substantial modifications to the simulator, and presents the difficulty of determining which files (if any) in the workload snapshot are sparse.

6.1.3.2 Simulating Sparse Files

Sparse files can cause other inaccuracies in HBench-FS. Reading from a hole in a sparse file should never generate disk reads, since the file system simply provides zero data for the offsets corresponding to the hole. Since the cache simulator does not model sparse files, however, it always charges an application with an uncached read if the target data is not in the buffer cache. In the workloads I have examined, this is seldom an issue, as there are few, if any, reads from sparse files. Other workloads, such as transaction processing, may make heavier use of sparse files. To accurately predict the performance of these workloads it may be necessary to add support for sparse files to the cache simulator.

6.1.3.3 Additional Sources of Background I/O

When measuring the overhead from background I/O operations, HBench-FS only considers write-back activity generated by write system calls. Preliminary versions of HBench-FS included a similar benchmark to measure the background overhead generated by file create and remove operations. Studies of Soft Updates have shown that these meta-data updates can sometimes generate substantial asynchronous disk traffic [Seltzer00a]. On the workloads that I studied, however, the inaccuracies in measuring this effect were greater than the small amounts of background load caused by create and remove calls. Therefore I did not include this benchmark in the final version of HBench-FS.

In theory, any file system request could generate background disk traffic. Ideally, HBench-FS would model this by running tests similar to the existing write-back overhead benchmark for each type of file system call.

6.1.3.4 Pathname lookup

As described in Section 4.4.3.2.1, the microbenchmark for pathname lookup time varies the state of both the attribute and name caches. It provides measurements for the cases when both the name and attributes for the target file are cached, when neither is cached, and when the attributes are cached but the name is not.[2] The fourth possible scenario, cached name data and uncached attribute data, defied measurement, as I was unable to generate a sequence of operations that would reliably cause a file's name to be cached without its attributes. Since this case rarely occurred in the traces I examined, I approximated its performance using the time for a lookup with neither name nor attribute data cached.

In practice, a lookup request that misses in either the name or attribute cache may not have to pay the cost of going to disk to fetch the required data. The data may be in the buffer cache. To simulate this case, whenever a lookup misses in either the name or attribute cache, the cache simulator determines whether the first data block of the target directory is in the buffer cache. If it is, then the cost of the lookup is assumed to be the same as a lookup that hits in both caches. (I.e., it is satisfied at memory speeds rather than at disk speeds.) It is probably possible to devise a microbenchmark for this scenario, but in practice it has not been necessary.

6.1.3.5 Read Request Sizes

Although request size is one of the arguments to read calls, this value does not always reflect the amount of data actually transferred to the user. If the request size is larger than the amount of data between the current file offset and the end of the file, the file system only transfers the existing data. When generating the request vector for a read call, HBench-FS uses mimics this behavior when it determines the number of blocks transferred to the user.

---

2. In benchmarking this scenario, I use hard links, which allow the creation of multiple directory entries that refer to the same file (and hence the same attributes). This allows me to warm the attribute cache by accessing a file via one link. Then, during the measurement phase of the benchmark, I measure the time to access the same file using a different (uncached) link.

6.1.3.6 Truncate Size Argument

An argument to the truncate and ftruncate system calls specifies the size to which the target file should be truncated. By truncating a file to a non-zero size, an application can eliminate part of the data in a file, or grow a file by truncating it to a larger size, essentially creating a hole at the end of the file. It would require a multitude of additional truncate benchmarks to accurately simulate this behavior in HBench-FS. Examining a variety of workload traces, however, shows that in practice these system calls are rarely used except to truncate a file to size zero, so I have ignored the uncommon case of truncating files to a non-zero sizes.

The traces I examined also showed that most file truncation occurs as the result of open calls which specify the *O_TRUNC* flag, which directs the operating system to truncate the target file before opening it.

6.1.3.7 Lookup Failures

In developing HBench-FS, I noticed that most traces contain a large number of lookup operations that fail, usually during *open*, *stat*, or *access* system calls. This typically occurs when a program searches several different directories to find a desired file, (e.g., the shell looking for an executable in each directory listed in a user's, or a compiler looking for header files in several directories). This behavior occurs so often that it can have a noticeable effect on workload performance, and therefore needed to be modeled by HBench-FS.

6.2 Conclusions

Each new generation of computer hardware provides stunning performance improvements. Processor speeds, network bandwidths, and memory sizes continue to grow rapidly. Disk performance has also improved, but at a much slower rate. Increased disk capacities and network bandwidths have sparked a similar increases in data set sizes. As a result of these two trends, storage systems are becoming an increasing bottleneck for many contemporary applications.

Many techniques have been proposed, implemented, and sold as solutions to this "I/O bottleneck," including file system optimizations such as Soft Updates [Ganger00],

hardware solutions, such as striping data across multiple disks to achieve higher bandwidths [Patterson88], and integrated solutions combining new file systems and disk array techniques [Hitz94]. Distressingly, the technology for evaluating file system performance has not kept pace with this rapid evolution of storage system design. Research papers continue to evaluate new architectures using simple benchmarking techniques that have changed little over the past twenty years.

Realistic techniques for evaluating the performance of a file system should allow users to answer the question, “How will my workload perform on this file system?” Current benchmarks suffer from a variety of shortcomings that prevent them from effectively answering this question. Benchmarks almost always evaluate empty or near-empty file systems, a condition real users rarely encounter. Another shortcoming of most benchmarks is that they operate by executing a fixed workload on the target file system. This provides little, if any, information about how workloads with different mixes of operations will perform.

In this dissertation, I have presented techniques for addressing both of these problems. File system aging provides a deterministic technique for evaluating a file system in a realistic state. Because aging generates this state by replaying a sequence of operations mimicking the long term workload on a file system, it accurately reproduces both visible (e.g., file system utilization) and invisible (e.g., file fragmentation) aspects of file system state, including types of state that may be unique to the underlying file system architecture, such as the clustering or fragmentation of related inodes in FFS. File system aging also provides a technique to evaluate design decisions only affect performance over the life of the file system.

Workload-based benchmarking addresses the second problem. HBench-FS provides a framework that allows users to predict how a specific workload will perform on different file systems. In addition to allowing workload-specific benchmarking, HBench-FS offers a number of other advantages to potential users. By separating the evaluation of a file system from the evaluation of a workload, it allows a user to predict the performance of a file system workload using only published data about the file

system's performance. This makes it possible to evaluate a file system that might not be physically available for testing, whether due to cost or other factors. Similarly, HBench-FS can be used to evaluate file systems that do not even exist, allowing users to conduct "what if" studies by providing hypothetical system vectors. In addition to providing simple predictions about how a workload will perform on a specific file system platform, HBench-FS provides a wealth of data that researchers and developers can use to better understand application behavior and file system performance.

Appendix A

HBench-FS Input Formats

As described in Chapter 4, HBench-FS takes two inputs, a workload profile, consisting of a file system trace and a file system snapshot, and a system profile containing the results of a suite of microbenchmarks. In this appendix, I document the formats of these inputs and briefly describe the library functions that the HBench-FS tools use to manipulate them.

A.1 Trace Format

An HBench-FS trace consists of multiple *records*, each of which describes all or part of a single file system operation. The records are written in binary format. This allows the processing tools to manipulate the traces without extensive parsing but means that traces are not portable across big- and little-endian machines.

A trace record consists of an *fsop_t* (**f**ile **s**ystem **op**eration) structure followed by one or more strings. Some file system operations are described by multiple trace records. The *fsop_t* structure, along with related substructures, is shown in Figure A.1. The *fsop_t* structure may contain pointers to one or more strings associated with the file system operation, such as the name of the target file and its directory. These pointers are invalid in raw traces; the trace processing library fills them it as it reads records from a trace.

Multiple *fsop_t* structures may be *chained* together to specify a single file system operation. An open system call, for example, would use one *fsop_t* for each component of

the target file's pathname and one for the open request. When multiple operations are chained together in this manner, all except the last have the *FLAG_CHAIN* flag set in the *op_flags* field of the *fsop_t* structure. The last operation in the chain has the *FLAG_CHAIN_END* flag set. All operations that take a pathname argument must be broken into chains in the input to the HBench-FS tools.[1] The HBench-FS preprocessing tool adds truncate operations to chains for file and directory remove operations and for open requests that specify the *O_TRUNC* flag. This tool also substitutes a create operation for open operations that have the *O_CREAT* flag set. In its final output, HBench-FS collapses each chain to a single record.

The *fsop_t* structure contains the following fields.

op_type

This field specifies the type of operation described by the record. Table A.1 lists the valid operation types. Note that some of these operations types are not supported in the current version of HBench-FS.

op_flags

This field contains one or more flags. Valid flag values are described in Table A.2.

op_time_sec

op_time_nsec

These two fields provide a time stamp (seconds and nanoseconds, respectively) for the current operation. This indicates the time that the operation was issued in the original trace. HBench-FS uses these timestamps to determine the elapsed time between operations and the total time from the beginning to the end of the trace. Operations that are chained together have the same timestamp. The value of the timestamp for the first operation in a trace does not matter, as long as subsequent timestamps increase to indicate the passage of time.

1. For maximum usability, HBench-FS should include a preprocessing tool that divides individual operations into chains. In the current version, however, this functionality has been combined with the tool that parses the trace data produced by an instrumented kernel.

```
/* Descriptor for a file */

typedef struct {
        ino_t   inum;
        u_int   filenamelen;
        char    *filename;
        ino_t   dirinum;
        u_int   dirnamelen;
        char    *dirname;
} file_t;

/* Parameters for an I/O operation */

typedef struct {
        off_t   filesize;       /* File size */
        off_t   offset;         /* Offset for start of I/O */
        u_int   count;          /* Size of I/O in bytes */
} io_t;

/* Settable attributes */

typedef struct {
        u_int   mode;
        u_long  uid;
        u_long  gid;
        u_long  size;
        time_t  atime;
        time_t  mtime;
} attr_t;

/* The fsop_t structure defines the internal format HBench-FS tools use. */

typedef struct {
        u_int           op_type;        /* Operation type */
        u_int           op_flags;       /* Flags */
        time_t          op_time_sec;    /* Time of operation */
        u_int           op_time_nsec;
        pid_t           op_pid;         /* Process ID of calling process */
        u_long          op_client;      /* Client ID */
        u_int           op_packsize;    /* Size of NFS packet */
        u_int           op_open_flags;  /* Flags to open sys call */
        u_int           op_class;       /* General class of operation. */
                                        /*   specifies which of the following */
                                        /*   structures are valid. */
        file_t          op_target;      /* Target file */
         union {
            io_t        op_un_io;       /* Operation will use at most one of */
            attr_t      op_un_attr;     /*   these.  Second file is needed */
            file_t      op_un_file2;    /*   for rename and link. */
        } op_un;
        u_int           op_linklen;     /* Symlink data */
        char            *op_linkdata;

        u_int           subop[NSUBOP];  /* Sub-operations in this one */
        int             relation;       /* Relation to prev. op */
        double          op_latency;     /* Predicted latency of op. (in msec) */
        double          op_latency_real;/* Measured latency of op. (in msec) */
        double          op_dsched;      /* Measured time process was de- */
                                        /* scheduled---runnable w/o running */
        double          op_sleep_time;  /* Measured time process was sleeping */
                                        /* (I.e., blocked) */
} fsop_t;
```

Figure A.1. Trace Data Structures. This figure shows the C structures that define the trace format used by the HBench-FS tools. One or more *fsop_t* structures are used for each operation in the trace.

Operation Name	Operation Number	Description
OP_NULL	0	Null operation.
OP_GETATTR	1	Get Attributes. Used for stat and fstat system calls.
OP_SETATTR	2	Set Attributes. Used for chmod, chown, etc.
OP_LOOKUP	3	Lookup. Resolve one name in a specified directory
OP_READ	4	File read operation
OP_WRITE	5	File write operation
OP_WRITECACHE	6	*Not Used*
OP_CREATE	7	File create operation.
OP_REMOVE	8	File remove operation.
OP_RENAME	9	File rename operation.
OP_LINK	10	Create link to existing file. *Not Supported.*
OP_READLINK	11	Read symbolic link. *Not Supported.*
OP_SYMLINK	12	Create symbolic link. *Not Supported.*
OP_MKDIR	13	Directory create operation.
OP_RMDIR	14	Directory remove operation
OP_READDIR	15	Directory read operation
OP_STATFS	16	Return file system attributes. *Not Supported*
OP_ROOT	17	Return handle for root of file system. *Not Supported*
OP_OPEN	18	File open operation.
OP_OPENDIR	19	Directory open operation.
OP_CLOSE	20	File close operation.
OP_CLOSEDIR	21	Directory close operation.
OP_ACCESS	22	Access system call.
OP_TRUNCATE	23	File truncate operation.
OP_CHDIR	24	Change directory call.

Table A.1: Operation Types. This table lists the operation types supported by the HBench-FS trace format. Note that some of the types are not supported, and some (e.g., OP_WRITECACHE and OP_ROOT) are NFS operations, rather than system calls.

Flag Name	Flag Value	Description
FLAG_ISFILE	0x001	Target of this operation is a file.
FLAG_ISDIR	0x002	Target of this operation is a directory.
FLAG_ISLINK	0x004	Target of this operation is a symbolic link.
FLAG_FAIL	0x008	This operation fails.
FLAG_CHAIN	0x010	This operation is part of a chain of operations.
FLAG_CHAIN_END	0x020	This is the last operation in a chain
FLAG_IGNORE	0x100	This operation should be ignored.

Table A.2: Flag Values. This table list the legal flag values for the *op_flags* field of the *fsop_t* structure. HBench-FS sets the FLAG_FAIL flag on operations that it expect to fail, for example attempts to open non-existent files. HBench-FS sets the FLAG_IGNORE flag on operations that it can't process. This typically occurs when an operation in the middle of a chain fails. Subsequent operations in that chain are ignored.

op_pid

This field provides an identifier for the process that issued the current operation. Current versions of HBench-FS do not use this field.

op_client

This field provides an identifier for the client that issued the current operation. Combined with the *op_pid* field, above, this should provide a unique identifier for every job running in a distributed system. As the current version of HBench-FS only supports workloads running on a single machine, this field is not used.

op_packetsize

This field is not used. It was originally intended to store the size of an NFS request for traces that were generated from NFS traffic.

op_op_flags

For open requests (*OP_OPEN* in the *op_type* field), this field indicates the flags that were passed to the open request. HBench-FS uses these fields to determine the proper behavior to simulate for the open request. I.e., whether to create or truncate the target file.

op_class

This field indicates the *class* of the operation. HBench-FS divides all file system operations into six different classes. The class indicates which of the optional fields (esp. in the *op_un* field) are valid. Table A.3 shows the possible values for this field and the mapping from operation types to classes. Because this mapping is fixed, the class field is redundant. It is a relic from earlier versions of HBench-FS.

op_target

This sub-structure describes the target file or directory of the operation. It is a *file_t* structure, as defined in Figure A.1. This structure contains the inode number and name of the target file / directory and its parent directory.

op_un

This is a union of three structures. Table A.3 shows which (if any) of these sub-structures is valid for each class of operation.

op_linklen

If the operation is type *OP_SYMLINK* (corresponding to the *symlink* system call), this field will contain the length of the symbolic link.

op_linkdata

If the operation is type *OP_SYMLINK* (corresponding to the *symlink* system call), this field is used as a pointer to the link data. I.e., to the pathname contained in thy symbolic link.

subop

This represents a partial request vector. It contains 17 elements, which are described in Table A.4. Note that this array does not contain the complete request vector. The full request vector is not (currently) stored in one place. Other data used in the request vector is stored in the *relation* and *op_type* fields.

relation

This field indicates the relationship (if any) between the current operation and the previous disk-bound operation. Valid values for this field are listed in Table A.5.

Class Name	Class Value	Operations	Valid Fields
CLASS_SIMPLE	1	All operations not listed in other classes	
CLASS_IO	2	OP_READ, OP_WRITE, OP_READDIR, OP_TRUNCATE	*op_un.op_un_io*
CLASS_ATTR	3	OP_SETATTR	*op_un.op_un_attr*
CLASS_FILE	4	OP_RENAME, OP_LINK	*op_un.op_un_file2*
CLASS_SYMLINK	5	OP_SYMLINK	*op_linklen*, *op_linkdata*
CLASS_IGNORE	6	OP_NULL, OP_WRITECACHE, OP_STATFS, OP_ROOT	

Table A.3: Operation Class Definitions. This table lists the legal values for the *op_class* field of the *fsop_t* structure. For each class, the table lists the operations which belong to that class and the fields in the *fsop_t* structure that are valid for operations of that class. For brevity, a complete list of the operations that use CLASS_SIMPLE is not included here. All operations that don't appear elsewhere in this table use CLASS_SIMPLE. For truncate operations, only the filesize field of the *io_t* substructure is used (i.e.,

op_latency

This field contains the predicted latency of the operation, in milliseconds.

op_latency_real

This optional field contains the measured latency for the current operation, in milliseconds.

Component Name	Component Index	Description
SUBOP_WRBLOCKS	0	Total number of blocks written
SUBOP_WRFRAGS	1	Total number of fragments written (only includes fragments contained in partial block at end of file)
SUBOP_RDBLOCKS	2	Total number of blocks read
SUBOP_RDFRAGS	3	Total number of fragments read (only includes fragments contained in partial block at end of file)
SUBOP_RDMISSBLOCKS	4	Number of blocks read that were not in buffer cache
SUBOP_RDMISSFRAGS	5	Number of fragments read that were not in buffer cache
SUBOP_LUHIT	6	Number of lookups hitting in both the name and attribute caches
SUBOP_LUMISS	7	Number of lookups missing in both the name and the attribute caches
SUBOP_LUMISSBC	8	Number of lookups missing in both the name and attribute caches, but where the name is in the buffer cache
SUBOP_LUATTRHIT	9	Number of lookups hitting in the attribute cache but missing in the name cache
SUBOP_LUNAMEHIT	10	Number of lookups hitting in the name cache but missing in the attribute cache
SUBOP_TRUNCCACHED	13	Number of truncate operations with cached meta-data
SUBOP_TRUNCUNCACHED	14	Number of truncate operations with uncached meta-data
SUBOP_FILEREMOVE	15	Number of file remove operations
SUBOP_DIRREMOVE	16	Number of directory remove operations
SUBOP_FILECREATE	17	Number of file create operations
SUBOP_DIRCREATE	18	Number of directory create operations

Table A.4: Request Vector Components. This table lists the components of the request vector that are stored in the *subop* array in an *fsop_t* structure. Note that the values 11 and 12 are not used.

op_dsched

This optional field contains the measured amount of time (in milliseconds) that the current operation was not running. This includes all time not spent in the run state, whether the process is blocked or ready to run and waiting for a processor.

op_sleep_time

This optional field contains the measured amount of time (in milliseconds) that the current operation spent in the blocked state. This difference between the *op_dsched* and *op_sleep_time* fields is the amount of time that the calling process spent runnable, but waiting for a processor.

Relation Name	Relation Value	Description
REL_INVALID	0	The relation field is uninitialized.
REL_SAME	1	The targets of the current and previous operations are the same.
REL_CHILD	4	The target of the previous operation is a child of the target of the current operation.
REL_PARENT	5	The target of the previous operation is the parent directory of the target of the current operation.
REL_SAMEDIR	6	The targets of the current and previous operations are in the same directory.
REL_UNKNOWN	7	There is no relationship between the targets of the current and previous operations.
REL_CACHED	8	The current operation is satisfied out of the cache.

Table A.5: Relationship Values. This table lists the valid values for the *relation* field of the *fsop_t* structure.

String	Description
op_target.filename	Name of target file.
op_target.dirname	Name of target directory.
op_un.op_un_file2.filename	Name of second file.
op_un.op_un_file2.dirname	Name of second file's directory
op_linkdata	Contents of symbolic link.

Table A.6: String Order. This table shows the order of strings in a trace file. The directory names may be either absolute or relative pathnames. Note that a second file and directory name can never appear in the same trace record as a symbolic link.

Each record in a trace file may be followed by one or more strings. There are five different strings that may be associated with a trace record—the file and directory names for the target file, the file and directory names for a second file (stored in *op_un.op_un_file2* in the case of *link*, and *rename* operations), and the contents of a symbolic link. For each of these strings the *fsop_t* structure contains a *char* * pointer and an unsigned integer field containing the length of the string. If a string is present in the trace file, the corresponding length field will be non-zero. Any strings that are present after a record are always stored in fixed order, shown in Table A.6.

A.2 Trace Library

In practice, there is seldom a need to directly manipulate trace files. All of the HBench-FS tools share a library of common functions for reading and writing trace records. This library can also be used to write filters to either pre-process or post-process workload profiles. The primary functions in the trace library are described in the following paragraphs.

```
fsop_t *
new_op (void)
```

This function allocates memory for a trace record, initializes the new record to contain null data, and returns a pointer to the new record.

```
fsop_t *
read_op (int fd)
```

This function reads a trace record from the file descriptor specified by *fd*. It reads the record and any associated strings from the current offset in the target file, allocating memory for them as needed. It returns a pointer to the new record or NULL if it reaches the end of the input file.

```
void
write_op (int fd, fsop_t *record)
```

This function writes the trace record specified by *record* to the file descriptor specified by *fd*. It writes the trace record along with any strings it contains.

```
#include <stdio.h>
#include <sys/types.h>
#include "trace.h"

main (argc, argv)
int argc;
char *argv[];
{
        fsop_t  *rec;
        int     read_blocks = 0;
        int     read_miss_blocks = 0;

        while ((rec = read_op(0)) != NULL) {
                if (rec->op_flags & FLAG_IGNORE) {
                        delete_op(rec);
                        continue;
                }

                read_blocks += rec->subop[SUBOP_RDBLOCKS];
                read_miss_blocks += rec->subop[SUBOP_RDMISSBLOCKS];
                delete_op (rec);
        }

        printf ("Miss rate = %4.3f\n",
           (double) read_miss_blocks / (double) read_blocks);
}
```

Figure A.2. Sample Trace-Processing Program. This sample program demonstrates the use of the trace library functions. The program iterates through the records in a trace file, reading them from standard input. It sums the total number of blocks read across all operations, along with the total number of reads that miss in the buffer cache. When it finishes processing the trace file, the program prints the predicted cache miss rate for the workload.

```
void
delete_op (fsop_t *record)
```

This function deletes the trace record specified by *record*. It frees all storage allocated to the record itself as well as any strings it contains.

```
void
print_op (FILE *output, fsop_t record)
```

This function prints a human-readable summary of the contents of a trace record.

The standard paradigm for using these functions is to iterate over a trace reading each record (with *read_op*), processing it, possibly writing it (with *write_op*), and then deleting it (with *delete_op*). Figure A.2, shows a simple program that post-processes a trace to determine the predicted cache miss rate.

A.3 Snapshot Format

In addition to a file system trace, HBench-FS also requires a snapshot of the file system from which the trace was collected. Unlike the trace format, the snapshot uses a simple text format, typically generated by post-processing the output from the UNIX *ls –liArR* command. The snapshot contains a listing of the files in each directory on the target file system. Each listing starts with the pathname of the directory (relative to the root of the file system) on a single line. Each following line describes one entry in that directory. These lines have the format:

```
inum type size name [link]
```

The *inum* field contains the inode number of the file. The *type* field specifies the type of the file (0 for regular files, 1 for directories, or 2 for symbolic links). The size field contains the file size in bytes. The name field is the name of the file. The optional link field contains the link data if the file is a symbolic link.

The last line of the directory listing is a line containing two tilde characters (i.e., "~~").

The directories are typically listed in depth-first order, as that is the order they appear in the output from the *ls* command. Any order is acceptable, as long as no directory is listed until after its parent directory has been listed.

The first line of the snapshot file necessarily contains the name of the root directory. In addition to the name, this line also contains the inode number and size of that directory.

Figure A.3 contains a small sample snapshot.

```
./ 2 512
21376 1 512 src
32064 1 512 lib
42752 1 512 include
10688 1 512 bin
~~
./src/
21386 0 3196 trim.c
21385 0 1958 total_time.c
21384 0 907 summary.c
21380 0 445 print_trace.c
21378 0 19124 logparse.c
21377 2 10 include ../include
53440 1 512 RCS
21388 0 1166 Makefile
~~
./src/RCS/
53446 0 4626 trim.c,v
53447 0 1056 summary.c,v
53443 0 669 print_trace.c,v
53445 0 23779 logparse.c,v
53441 0 547 Makefile,v
~~
./lib/
32065 0 26152 libfssim.a
~~
./include/
42753 0 7502 trace.h
~~
./bin/
10697 0 31288 trim
10696 0 29196 total_time
10689 0 51712 timing
10695 0 28420 summary
10690 0 161918 simulator
10694 0 28114 print_trace
10693 0 54927 logparse
10692 0 83295 computeRelation
10691 0 83236 computeInodes
```

Figure A.3. Sample Snapshot. This figure shows a small sample snapshot in the format expected by HBench-FS. The target file system contains four top level directories, *src*, *lib*, *include*, and *bin*, each of which contains a variety of files.

A.4 System Profile Format

The system profile is a text file containing the results of running the HBench-FS microbenchmark suite on a file system. Each line of this file contains the results of a single microbenchmark. The lines are of the format:

```
benchmark result
```

The *benchmark* field contains the name of a benchmark as listed in the first column of Table 4.4. The *result* field contains the benchmark result in the units listed in the third

column of Table 4.4. The system profile may contain comment lines starting with the '#' character.

This file is generated automatically by the *Makefile* that controls the running of the HBench-FS microbenchmarks.

References

[Baker91] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, John K. Ousterhout. Measurements of a Distributed File System. *Proceedings of the 13th Symposium on Operating Systems Principles (SOSP),* pp. 198 – 212. Monterey, CA. October 1991.

[Baker92] Mary Baker, Satoshi Asami, Etienne Deprit, John Ousterhout, Margo Seltzer. Non-Volatile Memory for Fast, Reliable File Systems. *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V),* pp. 10 – 22. Boston, MA. October 1992.

[Barve99] Rakesh Barve, Elizabeth Shriver, Phillip B. Gibbons, Bruce K. Hillyer, Yossi Matias, Jeffrey Scott Vitter. Modeling and optimizing I/O throughput of multiple disks on a bus. *Proceedings of Sigmetrics '99,* pp. 83 – 92. Atlanta, GA. May 1999.

[Beck98] Michael Beck, Harald Böhme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus, Dirk Verworner. *Linux Kernel Internals,* Second Edition. Addison Wesley Longman Limited. Harlow, England. 1998.

[Bennet91] J. Michael Bennett, Michael A. Bauer, David Kinchlea. Characteristics of Files in NFS Environments. *Proceedings of the 1991 ACM Symposium on Small Systems,* pp. 33 – 40. 1991.

[Biswas90] P. Biswas, K.K. Ramakrishnan. File Access Characterization of VAX/VMS Environments. *Proceedings of the 10th International Conference on Distributed Computing Systems,* pp. 227 – 234. Paris, France. May 1990.

[Blackwell95] Trevor Blackwell, Jeffrey Harris, Margo Seltzer. Heuristic Cleaning Algorithms in Log-Structured File Systems. *Proceedings of the 1995 USENIX Technical Conference,* pp. 277 – 288. New Orleans, LA. January 1995.

[Bosch96] Peter Bosch, Sape J. Mullender, Cut-and-Paste file-systems: integrating simulators and file-systems. *Proceedings of the 1996 USENIX Technical Conference,* pp. 307 – 318. San Diego, CA. January 1996.

[Bozman91] G.P. Bozman, H.H. Ghannad, E.D. Weinberger. A trace-driven study of CMS file references. *IBM Journal of Research and Development,* Vol. 35, Num. 5/6, pp. 815 – 828. September/November 1991.

[Brown97a] Aaron Brown. A Decompositional Approach to Performance Evaluation. Harvard University Computer Science Technical Report TR-03-97. April 1997.

[Brown97b] Aaron Brown. Operating System Benchmarking in the Wake of *Lmbench*: A Case Study of the Performance of NetBSD on the Intel x86 Architecture. *Proceedings of Sigmetrics '97,* pp. 214 – 224. Seattle, WA. June 1997.

[Burgess] Gary Burgess. What is the TPC Good For? or, the Top Ten Reasons in Favor of TPC Benchmarks. Transaction Processing Performance Council. http://www.tpc.org/articles/TopTen.html.

[Cao94] Pei Cao, Edward W. Felton, Kai Li. Implementation and Performance of Application-Controlled File Caching. *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI),* pp. 165 – 177. Monterey, CA. November 1994.

[Cao96] Pei Cao, Edward W. Felton, Anna R. Karlin, Kai Li. Implementation and Performance of Integrated Application-Controlled File Caching, Prefetching, and Disk Scheduling. *ACM Transactions on Computer Systems,* Vol. 14, Num. 4, pp. 311 – 343. November, 1996.

[Chang90] A. Chang, M. Mergen, R. Rader, J. Roberts, S. Porter. Evolution of storage facilities in AIX Version 3 for RISC System/6000 processors. *IBM Journal of Research and Development*, Vol. 34, No. 1, pp. 105 – 109. January 1990.

[Chang99] F. Chang, G. Gibson. Automatic Hint Generation through Speculative Execution. *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI),* pp. 1 – 14. New Orleans, LA. February 1999.

[Chiang93] Chi-ming Chiang, Matt W. Mutka, Characteristics of User File Usage Patterns. *Systems and Software,* Vol. 23, Num. 3, pp. 257 – 268. December 1993.

[Christenson97] Nick Christenson, David Beckemeyer, Trent Baker. A Scalable News Architecture on a Single Spool. *;login:,* Vol. 22, Num. 5, pp. 41 – 45. June 1997.

[Coker00] Russell Coker. Bonnie++ now at 1.00c (experimental). http://www.coker.com.au/bonnie++/ (site checked September 18, 2000).

[Cranor99] Charles D. Cranor, Gurudatta M. Parulkar. The UVM Virtual Memory System. *Proceedings of the 1999 USENIX Technical Conference,* pp. 117 – 130. Monterey, CA. June 1999.

[Dahlin94a] Micheal D. Dahlin, Clifford J. Mather, Randolph Y. Wang, Thomas E. Anderson, David A. Patterson. A Quantitative Analysis of Cache Policies for Scalable Network File Systems. *Proceedings of Sigmetrics '94,* pp. 150 – 160. Nashville, TN. May 1994.

[Dahlin94b] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, David A. Patterson. Cooperative Caching: Using Remote Client memory to Improve File System Performance. *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI),* pp. 267 – 280. Monterey, CA. November 1994.

[Douceur99] John R. Douceur, William J. Bolosky. A Large-Scale Study of File-System Contents. *Proceedings of Sigmetrics '99,* pp. 59 – 70. Atlanta, GA. May 1999.

[Duncan88] Ray Duncan. *Advanced MSDOS Programming*, Second Edition. Microsoft Press. Redmond, WA. 1988.

[Forin94] Alessandro Forin, Gerald R. Malan. An MS-DOS File System for UNIX. *Proceedings of the Winter 1994 USENIX Technical Conference,* pp. 337 – 354. San Francisco, CA. January 1994.

[Fritchie97] Scott Lystig Fritchie. The Cyclic News Filesystem: Getting INN To Do More With Less. *Proceedings of the 11th Systems Administration Conference (LISA '97),* pp. 99 – 112. San Diego, CA. October 1997.

[Gaede99] Steven L. Gaede. Perspectives on the SPEC SDET Benchmark. January 1999. Available at http://www.spec.org/osg/sdm91/sdet.

[Ganger97] Gregory R. Ganger, M. Frans Kaashoek. Embedded Inodes and Explicit Grouping: Exploiting Disk Bandwidth for Small Files. *Proceedings of the 1997 USENIX Technical Conference*, pp. 1 – 17. Anaheim, CA. January 1997.

[Ganger00] Gregory R. Ganger, M. Kirk McKusick, Craig A. N. Soules, Yale N. Patt. Soft Updates: A Solution to the Metadata Update Problem in File Systems. *ACM Transactions on Computer Systems*, Vol. 18, Num. 2, pp. 127 – 153. May 2000.

[Gingell87] Robert A. Gingell, Joseph P. Moran, William A. Shannon. Virtual Memory Architecture in SunOS. *Proceedings of the Summer 1987 USENIX Technical Conference*, pp. 81 – 94. Phoenix, AZ. June 1987.

[Herrin93] Eric H. Herrin II, Raphael A. Finkel. The Viva File System. University of Kentucky Department of Computer Science Technical Report Number 225-93. 1993.

[Hitz94] D. Hitz, J. Lau, M. Malcolm. File System Design for an NFS File Server Appliance. *Proceedings of the Winter 1994 USENIX Technical Conference*, pp. 235 – 246. San Francisco, CA. January 1994.

[Howard88] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, Michael J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computing Systems*, Vol. 6, No. 1, pp. 51 – 81. February 1988.

[Katcher97] Jeffrey Katcher. PostMark: A New File System Benchmark. Technical Report TR3022. Network Appliance Inc. October 1997.

[Kimbrel96] Tracy Kimbrel, Andrew Tomkins, R. Hugo Patterson, Brian Bershad, Pei Cao, Edward W. Felton, Garth A. Gibson, Anna R. Karlin, Kai Li. A Trace-Driven Comparison of Algorithms for Parallel Prefetching and Caching. *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 19 – 34. Seattle, WA. October 1996.

[Kleiman86] S.R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. *Proceedings of the Summer 1986 USENIX Technical Conference*, pp. 238 – 247. Atlanta, GA. June 1986.

[Kotz94a] David Kotz, Song Bac Toh, Sriram Radhakrishnan. A Detailed Simulation Model of the HP 97560 Disk Drive. Dartmouth College Department of Computer Science technical repot PCS-TR94-220. July 1994.

[Kotz94b] David Kotz, Nils Nieuwejaar. Dynamic File-Access Characteristics of a Production Parallel Scientific Workload. *Proceedings of Supercomputing '94*, pp. 640 – 649. Washington, DC. November 1994.

[Kuenning94] Geoffrey H. Kuenning, Gerald J. Popek, Peter L. Reiher. An Analysis of Trace Data for Predictive File Caching in Mobile Computing. *Proceedings of the Summer 1994 USENIX Technical Conference*, pp. 291 – 303. Boston, MA. June 1994.

[Lazowska84] Edward D. Lazowska, John Zahorjan, G. Scott Graham, Kenneth C. Sevcik. *Quantitative System Performance.* Prentice-Hall Inc. Englewood Cliffs, NJ. 1984.

[Manley97] Stephen Manley, Michael Courage, Margo Seltzer. A Self-Scaling and Self-Configuring Benchmark for Web Servers. Harvard University Computer Science Technical Report, TR-17-97. 1997

[Manley98] Stephen Manley, Margo Seltzer, Michael Courage. A Self-Scaling and Self-Configuring Benchmark for Web Servers. *Proceedings of Sigmetrics '98,* pp. 270 – 271. Madison, WI. June 1998.

[Manley00] Stephen Manley, Network Appliance Inc. Personal Communication. September 2000.

[Matthews97] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, Thomas E. Anderson. Improving the Performance of Log-Structured File Systems with Adaptive Methods. *Proceedongs of the 16th Symposium on Operating Systems Principles (SOSP),* pp. 238 – 251. Saint-Malo, France. October 1997.

[McKusick84] M. Kirk McKusick, William Joy, S. Leffler, R. Fabry. A Fast File System for UNIX. *ACM Transactions on Computing Systems,* Vol. 2, Num. 3, pp. 181 – 197. August 1994.

[McKusick96] M. Kirk McKusick, Keith Bostic, Michael Karels, John Quarterman. *The Design and Implementation of the 4.4BSD Operating System.* Addison-Wesley. Reading, MA. 1996.

[McVoy91] L. W. McVoy, S. R. Kleiman. Extent-like Performance from a UNIX File System. *Proceedings of the Winter 1991 USENIX Technical Conference,* pp. 33 – 43. Dallas, TX. January 1991.

[Menascé96] Daniel A. Menascé, Odysseas I. Pentakalos, Yelena Yesha. An Analytic Model of Hierarchical Mass Storage Systems with Network Attached Storage Devices. *Proceedings of SIGMETRICS '96,* pp. 180 – 189. Philadelphia, PA. June 1996.

[Menascé99] Daniel A. Menascé, Virgilio A. F. Almeida. Evaluating Web-Server Capacity. *Web Techniques.* April 1999. http://www.webtechniques.com/archives/1999/04/menasce.

[Miller91] Ethan L. Miller, Randy H. Katz. Input/Output Behavior of Supercomputing Applications. *Proceedings of the 1991 Conference on Supercomputing,* pp. 567 – 576. Albuquerque, NM. November 1991.

[Mowry96] Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications. *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI),* pp. 3 – 17. Seattle, WA. October, 1996.

[Mummert96] L. Mummert, M. Satyanarayanan. Long Term Distributed File Reference Tracing: Implementation and Experience. *Software—Practice and Experience,* Vol. 26, Num. 6, pp. 705 – 736. June 1996.

[Nelson88] Michael N. Nelson, Brent B. Welch, John K. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems,* Vol. 6, Num. 1, pp. 134 – 154. February 1988.

[Oram86] Andrew Oram and Steve Talbott. *Managing Projects with make.* O'Reilly & Associates, Inc. Sebastopol, CA. 1986.

[Ousterhout85] J. Ousterhout, H. Costa, D. Harrison, J. Kunze, M. Kupfer, J. Thompson. A Trace-Driven Analysis of the UNIX 4.2BSD File System. *Proceedings of the 10th Symposium on Operating Systems Principles (SOSP),* pp. 15 – 24. Orcas Island, WA. December 1985.

[Park90] A. Park, J.C. Becker. IOStone: A Synthetic File System Benchmark. *Computer Architecture News,* Vol. 18, Num. 2, pp. 45 – 52. June 1990.

[Patterson88] David Patterson, Garth Gibson, Randy Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). *Proceedings of the 1988 SIGMOD Conference on Management of Data,* pp. 109 – 116. Chicago, IL. June 1988.

[Patterson94] David Patterson. How to Have a Bad Career in Research / Academia. Keynote Address at 1st Symposium on Operating Systems Design and Implementation (OSDI). Monterey, CA. November 1994. Slides and audio available at http: / / www.cs.utah.edu / ~lepreau / osdi / keynote / abstract.html.

[Patterson95] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, Jim Zelenka. Informed Prefetching and Caching. *Proceedings of the 15th Symposium on Operating Systems Principles (SOSP),* pp. 79 – 95. Copper Mountain, CO. December 1995.

[Peacock88] J. Kent Peacock. The Counterpoint Fast File System. *Proceedings of the Winter 1988 USENIX Technical Conference,* pp. 243 – 249. Dallas, TX. February 1988.

[Peuto77] Bernard L. Peuto, Leonard J. Shustek. An Instruction Timing Model of CPU Performance. *Proceedings of the 4th Annual Symposium on Computer Architecture,* pp. 165 – 178. March 1977.

[Porcar82] J.M. Porcar. *File Migration in Distributed Computer Systems.* Ph.D. Dissertation, University of California, Berkeley. July 1982.

[Powell77] Michael L. Powell. The DEMOS File System. *Proceedings of the 6th Symposium on Operating Systems Principles (SOSP),* pp. 33 – 42. West Lafayette, IN. November, 1977.

[Purakayastha95] A. Purakayastha, Carla Ellis, David Kotz, N. Nieuwejaar, Michael L. Best. Characterizing Parallel File-access Patterns on a Large-scale Multiprocessor. *Proceedings of the 9th International Parallel Processing Symposium,* pp. 165 – 172. IEEE Computer Society Press. April 1995.

[Ramakrishnan92] K.K. Ramakrishnan, Prabuddha Biswas, Ramakrishna Karedla. Analysis of File I / O Traces in Commercial Computing Environments. *Proceedings of Sigmetrics '92,* pp. 78 – 90. Newport, RI. June 1992.

[Ritchie74] Dennis M. Ritchie, Ken Thompson. The UNIX Time-Sharing System. *Communications of the ACM*, Vol. 17, Num. 7, pp. 365 – 375. July 1974.

[Rosenblum92] Mendel Rosenblum, John Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computing Systems,* Vol. 10, No. 1, pp. 26 – 52. February 1992.

[Roselli98] Drew Roselli, Thomas Anderson. Characteristics of File System Workloads. University of California at Berkeley Technical Report CSD-98-1029. December 1998.

[Roselli00] Drew Roselli, Jacob R. Lorch, Thomas E. Anderson. A Comparison of File System Workloads. *Proceedings of the 2000 USENIX Technical Conference,* pp. 41 – 54. San Diego, CA. June 2000.

[Ruemmler94] Chris Ruemmler, John Wilkes. An introduction to disk drive modeling. *IEEE Computer,* Vol 27, Num. 3, pp. 17 – 29. March 1994.

[Saavedra95] Rafael H. Saavedra, Alan Jay Smith. Measuring Cache and TLB Performance and Their Effect on Benchmark Runtimes. *IEEE Transactions on Computers,* Vol. 44, Num. 10, pp. 1223 – 1235. October 1995.

[Saavedra96] Rafael H. Saavedra, Alan J. Smith. Analysis of Benchmark Characteristics and Benchmark Performance Prediction. *ACM Transactions on Computer Systems,* Vol 14, Num. 4, pp. 344 – 384. November 1996.

[Saito98] Yasushi Saito, Jeffrey C. Mogul, and Ben Verghese. A Usenet Performance Study. Unpublished manuscript. http://www.research.compaq.com/wrl/projects/newsbench/usenet.ps. November, 1998.

[Sandberg85] Russell Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, Bob Lyon. Design and Implementation of the Sun Network Filesystem. *Proceedings of the Summer 1985 USENIX Technical Conference,* pp. 119 – 130. Portland, OR. June 1985.

[Sarkar96] Prasenjit Sarkar, John Hartman. Efficient Cooperative Caching using Hints. *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI),* pp. 35 – 46. Seattle, WA. October 1996.

[Satyanarayanan81] M. Satyanarayanan. A Study of File Sizes and Functional Lifetimes. *Proceedings of the 8th Symposium on Operating Systems Principles (SOSP),* pp. 96 – 108. Pacific Grove, CA. December 1981.

[Schmidt99] Brian K. Schmidt, Monica S. Lam, J. Duane Northcutt. The interactive performance of SLIM: a stateless, thin-client architecture. *Proceedings of the 17th Symposium on Operating Systems Principles (SOSP),* pp. 32 – 47. Kiawah Island Resort, SC. December 1999.

[Seltzer93] Margo Seltzer, Keith Bostic, M. Kirk McKusick, Carl Staelin. An Implementation of a Log-Structured File System for UNIX. *Proceedings of the Winter 1993 USENIX Technical Conference,* pp. 307 – 326. San Diego, CA. January 1993.

[Seltzer95] Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, Venkata Padmanabhan. File System Logging Versus Clustering: A Performance Comparison. *Proceedings of the 1995 USENIX Technical Conference,* pp. 249 – 264. New Orleans, LA. January 1995.

[Seltzer99] Margo Seltzer, David Krinsky, Keith Smith, Xiaolan Zhang. The Case for Application-Specific Benchmarking. *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HOTOS-VII),* pp. 102 – 107. Rico Rio, AZ. March 1999.

[Seltzer00a] Margo I. Seltzer, Gregory R. Ganger, M. Kirk McKusick, Keith A. Smith, Craig A. N. Soules, Christopher Stein. Journaling versus Soft Updates: Asynchronous Meta-data Protection in File Systems. *Proceedings of the 2000 USENIX Technical Conference,* pp. 71 – 84. San Diego, CA. June 2000.

[Seltzer00b] Margo Seltzer, Harvard University. Personal communication. November 20, 2000.

[Shirriff92] Ken W. Shirriff, John K. Ousterhout. A Trace-Driven Analysis of Name and Attribute Caching in a Distributed System. *Proceedings of the Winter 1992 USENIX Technical Conference,* pp. 315 – 331. San Francisco, CA. January 1992.

[Shriver98] Elizabeth Shriver, Arif Merchant, John Wilkes. An analytic behavior model for disk drives with readahead caches and request reordering. *Proceedings of Sigmetrics '98,* pp. 182 – 191. Madison, WI. June 1998.

[Shriver99] Elizabeth Shriver, Chistopher Small, Keith A. Smith. Why does file system prefetching work? *Proceedings of the 1999 USENIX Technical Conference,* pp. 71 – 84. Monterey, CA. June 1999.

[Shustek98] Leonard J. Shustek, Bernard L. Peuto. Retrospective: An Instruction Timing Model of CPU Performance. *25 Years of the International Symposia on Computer Architecture,* pp. 11 – 12. Gurindar Sohi, ed. ACM Press. New York, NY. 1998.

[Small97] Christopher Small, Narendra Ghosh, Hany Saleeb, Margo Seltzer, Keith A. Smith. Does Systems Research Measure Up? Harvard University Computer Science Technical Report TR-16-79. November 1997.

[Smith81] A. J. Smith. Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms. *IEEE Transactions on Software Engineering.* Vol. SE-7, No. 4, pp. 403 – 417. July 1981.

[Smith94] Keith Smith, Margo Seltzer. File Layout and File System Performance. Harvard University Computer Science Technical Report TR-35-94. December 1994.

[Smith96] Keith A. Smith, Margo Seltzer. A Comparison of FFS Disk Allocation Policies. *Proceedings of the 1996 USENIX Technical Conference,* pp. 15 – 25. San Diego, CA. January 1996.

[SPEC00] Standard Performance Evluation Council. SPECweb99 Release 1.02. On-line whitepaper. http://www.spec.org/osg/web99/docs/whitepaper.html.

[SPECa] Storage Performance Evaluation Council. SPEC JVM98 web site. http://www.spec.org/osg/jvm98.

[SPECb] Storage Performance Evaluation Council. SPEC mail2001 web site. http://www.spec.org/osg/mail2001.

[Spencer98] Henry Spencer, David Lawrence. *Managing Usenet.* O'Reilly & Associates, Inc. Cambridge, MA. 1998.

[Staelin91] Carl Staelin, Hector Garcia-Molina. Smart Filesystems. *Proceedings of the Winter 1991 USENIX Technical Conference,* pp. 45 – 51. Dallas, TX. January 1991.

[Swartz96] Karl L. Swartz. The Brave Little Toaster Meets Usenet. *Proceedings of the Tenth Systems Administration Conference (LISA '96),* pp. 161 – 170. Chicago, IL. October 1996.

[Tang95] Diane Tang. Benchmarking Filesystems. Senior Thesis. Harvard University. Cambridge, MA. April 1995.

[Thekkath94] Chandramohan A. Thekkath, John Wilkes, Edward D. Lazowska. Techniques for File System Simulation. *Software—Practice and Experience,* Vol. 24, Num. 11, pp. 981 – 999. November 1994.

[Tomkins97] Andrew Tomkins, R. Hugo Patterson, Garth Gibson. Informed Multi-Process Prefetching and Caching. *Proceedings of Sigmetrics '97,* pp. 100 – 114. Seattle, WA. June 1997.

[TPC90] Transaction Processing Performance Council. *TPC Benchmark B Standard Specification.* Waterside Associates. Fremont, CA. August 1990.

[VanMeter97] Rodney Van Meter. Observing the Effects of Multi-Zone Disks. *Proceedings of the 1997 USENIX Technical Conference,* pp. 19 – 30. Anaheim, CA. January 1997.

[Vogels99] Werner Vogels. File system usage in Windows NT 4.0. *Proceedings of the 17th Symposium on Operating Systems Principles (SOSP),* pp. 93 – 109. Charleston, SC. December 1999.

[Wang99] Randolph Y. Wang, Thomas E. Anderson, David A. Patterson, Virtual Log Based File Systems for a Programmable Disk. *Proceedings of the 2nd Symposium on Operating Systems Principles (OSDI),* pp. 29 – 42. New Orleans, LA. February 1999.

[Wilkes95] John Wilkes, Richard Golding, Carl Staelin, T. Sullivan. The HP AutoRAID hierarchical storage system. *Proceedings of the 15th Symposium on Operating Systems Principles (SOSP),* pp. 96 – 108. Copper Mountain, CO. December 1995.

[Wilkes96] John Wilkes. The Pantheon storage-system simulator. Hewlett-Packard Laboratories technical report HPL-SSP-95-14, revision 1. May 1996. Available from http://www.hpl.hp.com/research/itc/csl/ssp/Pantheon.

[Wittle93] M. Wittle, B. Keith. LADDIS: The Next Generation in NFS File Server Benchmarking. *Proceedings of the Summer 1993 USENIX Technical Conference,* pp. 111 – 128. Cincinnati, OH. June 1993.

[Wong00] Alexander Ya-li Wong, Margo Seltzer. Operating System Support for Multi-User, Remove, Graphical Interaction. *Proceedings of the 2000 USENIX Technical Conference,* pp. 183 – 196. San Diego, CA. June 2000.

[Yaghmour00] Karim Yaghmour, Michel R. Dagenais. Measuring and Characterizing System Behavior Using Kernel-Level Event Logging. *Proceedings of the 2000 USENIX Technical Conference,* pp. 13 – 26. San Diego, CA. June 2000.

[Ylonen96] Tatu Ylonen. SSH—Secure Login Connections Over the Internet. *Proceedings of the 6th USENIX Security Symposium,* pp. 37 – 42. San Jose, CA. July 1996.

[Zadok99] Erez Zadok, Ion Badulescu, Alex Shender. Extending File Systems Using Stackable Templates. *Proceedings of the 1999 USENIX Technical Conference,* pp. 57 – 70. Monterey, CA. June 1999.

[Zhang00] Xiaolan Zhang, Margo Seltzer. HBench:Java: An Application-Specific Benchmarking Framework for Java Virtual Machines. *Proceedings of the ACM Java Grande 2000 Conference,* pp. 62 – 70. San Francisco, CA. June 2000.

[Zhang01] Xiaolan Zhang, Margo Seltzer. HBench:JGC—An Application-Specific Benchmark Suite for Evaluating JVM Garbage Collector Performance. To appear in *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '01).* San Antonio, TX. January 2001.