A Comparison of File System Workloads

Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson
University of California, Berkeley and University of Washington
{drew, lorch}@cs.berkeley.edu, tom@cs.washington.edu

Abstract

In this paper, we describe the collection and analysis of file system traces from a variety of different environments, including both UNIX and NT systems, clients and servers, and instructional and production systems. Our goal is to understand how modern workloads affect the ability of file systems to provide high performance to users. Because of the increasing gap between processor speed and disk latency, file system performance is largely determined by its disk behavior. Therefore we primarily focus on the disk I/O aspects of the traces. We find that more processes access files via the memory-map interface than through the read interface. However, because many processes memory-map a small set of files, these files are likely to be cached. We also find that file access has a bimodal distribution pattern: some files are written repeatedly without being read; other files are almost exclusively read. We develop a new metric for measuring file lifetime that accounts for files that are never deleted. Using this metric, we find that the average block lifetime for some workloads is significantly longer than the 30-second write delay used by many file systems. However, all workloads show lifetime locality: the same files tend to be overwritten multiple times.

1 Introduction

Like other computer systems, file systems provide good performance by optimizing for common usage patterns. Unfortunately, usage patterns vary both over time and across different user communities. To help delineate current workload patterns, we decided to measure a wide range of file systems in a number of different environments, specifically, UNIX and Windows NT, client and server, instructional, research, and production. We compare our results with those from the Sprite study, conducted in 1991. Although we were interested in tracking how behavior has changed since the Sprite study, we do not directly reproduce all of their results. Their study focused on cache and virtual memory behavior. Since the relative performance of hardware has changed since that time, we focus instead on the I/O bottleneck.

We collected traces from four different groups of machines. Three of the groups run HP-UX, a variant of the UNIX operating system. One of these is an instructional laboratory, another is a set of computers used for research, and another is a single web server. The last group is a set of personal computers running Windows NT. This diversity of traces allows us to make conclusions not only on how current file system usage differs from past file system usage, but also how file system usage varies among machines used for different purposes.

Because improvements in disk latency are increasingly lagging behind those of processors and disk bandwidth, we chose to focus our study on measurements that elucidate how disk behavior is affected by workload and file system parameters. As the I/O gap grows, one way to provide good performance is to cache as many file reads and writes as possible and to minimize latencies for the remainder. For example, one way to avoid disk reads is by employing large file caches. Our results show that while small caches can avert many disk reads, there are diminishing benefits for large cache sizes. In addition to file reads, memory-mapping has become a popular file access method. We examine memory-mapping behavior in order to see the effect of memory-mapped files on the file cache. We find that more processes access files via memory-mapping than through reads or writes. For the UNIX workloads, we find that a small set of memory-mapped files tend to be shared among many processes. As a result, cache misses on these files are unlikely.

To avoid disk writes, the file system can increase the time between an application's write and flushing the data to disk, for example, by using NVRAM. By delaying writes, blocks that are deleted in the interval need not be written at all. We find that most blocks live longer than the standard 30-second write delay commonly employed by file systems. In UNIX systems, most blocks die within an hour; in NT, many blocks survive over a day. Most blocks die due to overwrites, and these overwrites have a high degree of locality—that is, most overwritten files are multiply overwritten. Because of this locality, even a small write buffer is sufficient to handle a day's

This research was supported by the National Science Foundation (Grant No. CCR-9972244), the State of California MICRO program, Cisco Systems, Fujitsu Microelectronics, IBM, Intel Corporation, Maxtor Corporation, Microsoft Corporation, Quantum Corporation, Sony Research Laboratories, Sun Microsystems, Toshiba Corporation, and Veritas Software. In addition, Roselli was supported by a GAANN fellowship.

worth of write traffic.

To reduce disk seeks, most file systems organize their layout to optimize for either reads or writes. We find that whether read traffic or write traffic dominates varies depending on the workload and the file system configuration. However, for all workloads, we find that individual files tend to have bimodal access patterns—they are either read-mostly or write-mostly. This tendency is most clear in frequently accessed files.

2 Related Work

Characterizing file system behavior is difficult due to both the wide range of workloads and the difficulty in obtaining data to analyze. Obviously, no trace analysis project has the scope to analyze all relevant features of all relevant workloads. Instead, each study lets us understand a piece of the greater picture.

In order to minimize the complexity of trace collection, many studies concentrate on static data, which they collect by examining file system metadata at one or several frozen instants in time [Douc99] [Sien94] [Chia93] [Benn91] [Saty81] [Smit81]. These studies of *snapshots* are useful for studying distributions of file attributes commonly stored in metadata, such as file size, last access time, last modification time, file name, and directory structure.

Dynamic traces of continuous file system access patterns yield more detailed information about file system usage. However, these traces are considerably more difficult to collect both because of the volume of data involved and because the collection process typically involves modifying the operating system kernel. Some tracing methods avoid altering the kernel by recording file system events that pass over a network [Blaz92] [Dahl94]. However, this method misses file system events that do not cross the network, such as local file system calls. Also, artifacts of the network file system being measured can affect these types of traces.

Modifying the kernel to obtain local file system behavior has its own set of drawbacks. First, the kernel source code is not always available. Second, the modified kernels must be deployed to users willing to run their applications on an altered kernel. Finally, the overhead of collecting fine-grained traces must be kept low so that overall system performance is not significantly degraded. Due to these limitations, most researchers limit their trace collection to only the data that is necessary to perform specific studies. For example, the traces collected to perform analysis of directory access behavior in [Floy89] do not include file read or write requests. The disk activity study in [Ruem93] is at the disk level and does not include specific file system calls. Mummert et al. focused on results relevant to disconnected file system operation [Mumm94]. Zhou and Smith collected traces on personal computers for research in low-power computing [Zhou99].

In 1985, Ousterhout et al. presented a general characterization of dynamically collected traces [Oust85]. In this work, they traced three servers running BSD UNIX for slightly over three days. This paper introduced a framework for workload analysis using metrics such as run length, burstiness, lifetime of newly written bytes, and file access sequentiality. Henceforth, we refer to this work as the BSD study. In 1991, Baker et al. conducted the same type of analysis on four two-day sets of traces of the Sprite file system [Bake91]. They collected these traces at the file servers and augmented them with client information on local cache activity. For the rest of this paper, we refer to this work as the Sprite study. The data analysis techniques developed in the BSD and Sprite studies were repeated in several subsequent studies. In 1991, Bozman et al. repeated many of the Sprite studies using traces from two separate IBM sites [Bozm91]. This study confirmed that the results from the Sprite study applied to non-academic sites. In 1999, the same studies were repeated on three sets of two-week traces taken from 45 hosts running Windows NT [Voge99]. This workload is close to our NT workload, and for the analyses that are directly comparable (file size, file lifetime and access patterns), our results are similar.

In this work, we repeat some of the influential studies introduced by the BSD study, such as file access patterns. In addition, we contribute new studies that have become relevant to modern systems, such as the effect of memory-mapping files on the file cache. A more complete comparison of the Sprite studies against our UNIX traces can be found elsewhere [Rose98]. Because the Sprite traces are publicly available, we generate results for the Sprite traces wherever possible for purposes of comparison.

3 Trace Collection

3.1 Environment

We collected the traces discussed in this paper in four separate environments. Three of these environments use Hewlett-Packard series 700 workstations running HP-UX 9.05. Each of the HP-UX machines has 64MB of memory. The first group consists of twenty machines located in laboratories for undergraduate classes. For the rest of this paper, we refer to this workload as the Instructional Workload (INS). The second group consists of 13 machines on the desktops of graduate students, faculty, and administrative staff of our research group project. We refer to this workload as the Research Workload (RES). Of all our traces, the environment for this workload most closely resembles the environment in which the Sprite traces were collected. We collected the third set of traces from a single machine that is the web server for an online library project. This host maintains a database of images using the Postgres database management system and exports the images via its web interface. This server received approximately 2,300 accesses per day during the period of the trace. We refer to this as the WEB workload. The INS machines mount home directories and common binaries from a non-traced Hewlett-Packard workstation. In total, we collected eight months of traces from the INS cluster (two semesters), one year of traces from the RES cluster, and approximately one month of traces from the WEB host.

We collected the fourth group of traces from eight desktop machines running Windows NT 4.0. Two of these machines are 450 MHz Pentium IIIs, two are 200 MHz Pentium Pros, and the other four are Pentium IIs ranging from 266–400 MHz. Five of them have 128 MB of main memory, while the others have 64, 96, and 256 MB. These hosts are used for a variety of purposes. Two are used by a crime laboratory director and his supervisor, a state police captain; they use these machines for time management, personnel management, accounting, procurement, mail, office suite applications, and web browsing and publishing. Another two are used for networking and system administration tasks: one primarily runs an X server, email client, web browser, and Windows NT system administration tools; the other primarily runs office suite, groupware, firewall, and web browsing applications. Two are used by computer science graduate students as X servers as well as for software development, mail, and web browsing. Another is shared among the members of a computer science graduate research group and used primarily for office suite applications. The final machine is used primarily as an X server, but occasionally for office suite and web browsing applications. Despite the different uses of the NT machines, the results are similar for all the machines, so we include them together as one group.

3.2 Trace Collection Methodology

We used separate tools to collect traces for the HP-UX and Windows NT systems. While both of our collection techniques trace similar file system events, their implementations are quite different.

3.2.1 HP-UX Collection Methodology

For the UNIX machines, we used the auditing subsystem to record file system events. Although the auditing system was designed for security purposes, it is ideal for tracing since it catches the logical level of requests using already-existing kernel functionality. The auditing subsystem gets invoked after a system call and is configured to log specified system calls with their arguments and return values. However, it does not record kernel file system activity, such as paging from executable images.

The major problem we faced in using the auditing system was that HP-UX records pathnames exactly as specified by the user, and users often specify paths relative to the current working directory instead of with their complete paths. Since some file systems use a file's parent directory to direct file layout, we needed to record the full pathname. We solved this problem by recording the current working directory's pathname for each process and configuring the auditing system to catch all system calls capable of changing the current working directory. These changes required only small changes to the kernel (about 350 of lines of C code) and were wholly contained within the auditing subsystem.

3.2.2 Windows NT Collection Methodology

We collected the Windows NT traces using a tool we developed that traces not only file system activity, but also a wide range of device and process behavior [Lorc00]. We focus here on the aspects of the tracer relevant to tracing file system activity.

We perform most of the file system tracing using the standard mechanism in Windows NT for interposing file system calls: a file system filter driver. A file system filter driver creates a virtual file system device that intercepts all requests to an existing file system device and handles them itself. Our filter device merely records information about the request, passes the request on to the real file system, and arranges to be called again when the request has completed so it can record information about the success or failure of the request. The design of our filter driver borrows much from the Filemon file system monitoring program [Russ97b].

A Windows NT optimization called the *fast path* complicates tracing these file systems. The operating system uses this optimization whenever it believes a request can be handled quickly, for example, with the cache. In this case, it makes a call to a fast-dispatch function provided by the file system instead of passing requests through the standard request path. In order to intercept these calls, we implemented our own fast-dispatch functions to record any calls made this way.

In order to collect data on memory-mapping operations, we needed to interpose Windows NT system calls. This is difficult because Microsoft gives no documented way to do this. Fortunately, a tool called Regmon solves this problem; it finds the system call entry point vector in memory and overwrites certain entry points with our own [Russ97a].

Because we interpose at the file system layer and not at the system call layer, there were some challenges in converting our traces to a format comparable with the UNIX traces. The first problem arises when the file system calls the cache manager to handle a read request, and there is a miss. The cache manager fills the needed cache block by recursively calling the file system. We need to identify the recursive requests because they do not reflect actual read requests and should be elided. We distinguish them by three of their properties: they are initiated by the kernel, they have the no-caching flag set (in order to prevent an infinite loop), and they involve bytes that are being read by another ongoing request. The second problem is that we cannot distinguish a read caused by an explicit read request from one caused by kernel-initiated read-ahead. We distinguish the latter by looking for read requests with the following four properties: they are initiated by the kernel, they have the no-caching flag set, they do not involve bytes currently being read by another request, and they are made to a file handle that was explicitly read earlier. Finally, it is also difficult to determine which read and write requests are due to paging of memory-mapped files. If a request is initiated by the kernel with the no-caching flag set and it does not belong to any of the previous characterizations, we classify it as a paging request.

The file system interface of Windows NT is quite different from that of UNIX. For instance, there is no `stat` system call in Windows NT, but there is a similar system call: `ZwQueryAttributesFile`. For the purpose of comparison, we have mapped the request types seen in Windows NT to their closest analogous system calls in UNIX in this paper.

4 Results

Due to the time-consuming nature of collecting statistics on the entire length of our traces (which are currently over 150GB compressed), we present results in this paper based on subsets of the traces. For the INS and RES traces, we used traces collected from the month of March 1997. For WEB, we used the traces from January 23 to February 16, 1997. Because this trace includes activity not related to the web server, we filtered it to remove non-web-server activity. Because the NT traces begin at different times, we chose a 31-day period for each host. All but one of these periods were within the first quarter of the year 2000; the other trace was taken from October and November of 1999. For the Sprite results, our results differ slightly from those presented by Hartman and Ousterhout [Hart93] because we filter them differently. For example, we do not include non-file, non-directory objects in any results.

None of our results include paging of executables. For the NT workload, executable paging constitutes 15% of all reads and nearly 30% of all writes. Paging activity for the UNIX workloads is unknown.

4.1 Histogram of Key Calls

To provide an overview of our workloads, we first present counts of the most common events traced; these are summarized in Table 1. The results reveal some notable differences among the workloads. For example, the WEB workload reads significantly more data than the other workloads; its read to write ratio is two orders of magnitude higher than any other workload. The NT workload reads and writes more than twice the amount of data per host per day than the INS and RES workloads, despite having significantly fewer users. Also, notable in all workloads is the high number of requests to read file attributes. In particular, calls to `stat` (including `fstat)` comprise 42% of all file-system-related calls in INS, 71% for RES, 10% for WEB, and 26% for NT.

Two common usage patterns could account for the large number of `stat` calls. First, listing a directory often involves checking the attributes of each file in the directory: a `stat` system call is made for each file. Second, a program may call `stat` to check attributes before opening and accessing a file. For example, the `make` program checks the last modification times on source and object files to determine whether to regenerate the object file. We measured the percentage of `stat` calls that follow another `stat` system call to a file from the same directory to be 98% for INS and RES, 67% for WEB, and 97% for NT. The percentage of `stat` calls that are followed within five minutes by an open to the same file is 23% for INS, 3% for RES, 38% for WEB, and only 0.7% for NT.

TABLE 1. Trace Event Summary

	INS	RES	WEB	NT	Sprite
hosts	19	13	1	8	55
users	326	50	7	8	76
days	31	31	24	31	8
data read (MB)	94619	52743	327838	125323	42929
data written (MB)	16804	14105	960	19802	9295
read:write ratio	5.6	3.7	341.5	6.3	4.6
all events (thousands)	317859	112260	112260	145043	4602
fork (thousands)	4275	1742	196	NA	NA
exec (thousands)	2020	779	319	NA	NA
exit (thousands)	2107	867	328	NA	NA
open (thousands)	39879	4972	6459	21583	1190
close (thousands)	40511	5582	6470	21785	1147
read (thousands)	71869	9433	9545	39280	1662
write (thousands)	4650	2216	779	7163	455
mem. map (thousands)	7511	2876	1856	614	NA
stat (thousands)	135886	79839	3078	37035	NA
get attr. (thousands)	1175	826	15	36	NA
set attr. (thousands)	467	160	23	273	NA
chdir (thousands)	1262	348	80	NA	NA
read dir. (thousands)	4009	1631	172	12486	NA
unlink (thousands)	490	182	2	285	106
truncate (thousands)	37	4	0	1981	42
fsync (thousands)	514	420	2	1533	NA
sync (thousands)	3	71	0	NA	NA

This table summarizes the number of events for the time period indicated for each trace. For all workloads, the above calls represent over 99% of all traced calls. The get attribute category includes `getacl`, `fgetacl`, `access`, and `getaccess`. The set attribute category includes `chmod`, `chown`, `utime`, `fchmod`, `fchown`, `setacl`, and `fsetacl`. The number of users is estimated from the number of unique user identifiers seen. This may be an overestimate since some user identifiers are simply administrative. For the NT traces, `exec` and `chdir` calls were not recorded, and process forks and exits were recorded only periodically during the NT traces.

Since this system call is so common, it would be worthwhile to optimize its performance. Since it is most commonly invoked near other `stat` calls in the same directory, storing the attribute data structures together with those from the same directory [McKu84] or within the directory structure [Gang97] may provide better performance than storing each file's attribute information with its data blocks.

4.2 Data Lifetime

In this section, we examine block lifetime, which we define to be the time between a block's creation and its deletion. Knowing the average block lifetime for a workload is important in determining appropriate write delay times and in deciding how long to wait before reorganizing data on disk. Our method of calculating lifetime differs from that used in the Sprite study, and, in some cases, results in significantly longer lifetimes. We find that most blocks live longer than 30 seconds—the standard write-delay used in many file systems. In particular, blocks created in the NT workload tend to be long-lived. Most blocks die by being overwritten, and these blocks are often overwritten many times.

4.2.1 Create-based Method

We calculate lifetime by subtracting a block's creation time from its deletion time. This is different from the *delete-based* method used by [Bake91] in which they track all deleted files and calculate lifetime by subtracting the file's creation time from its deletion time. In our *create-based* method, a trace is divided into two parts. We collect information about blocks created within the first part of the trace. We call the second part of the trace the *end margin*. If a tracked block is deleted during either part of the trace, we calculate its lifetime by subtracting the creation time from the deletion time. If a tracked block is not deleted during the trace, we know the block has lived for at least the end margin.

The main difference between the create-based and delete-based methods is the set of blocks that we use to generate the results. Because the delete-based method bases its data on blocks that are deleted, one cannot generalize from this data the lifetime distribution of newly created blocks. Because that is the quantity which interests us, we use the create-based algorithm for all results in this paper. One drawback of this approach is that it only provides accurate lifetime distributions for lifetimes less than the end margin, which is necessarily less than the trace duration. However, since our traces are long-term, we are able to acquire lifetime data sufficient for our purposes; we use an end margin of one day for all results in this section. Figure 1 shows the difference in results of create-based and delete-based methods on one of the Sprite traces. Due to the difference in sampled files, the delete-based method calculates a shorter lifetime than the create-based method.

If the traces collected reflect random samples of the steady state of creation and deletion, the principal difference between the methods would result from blocks that are created and never deleted. As a result of this difference, the create-based method predicts that disk space used will tend to increase with time—something disk sales confirm.

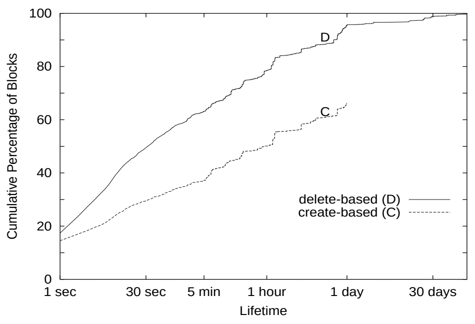

FIGURE 1. Create-based versus Delete-based Lifetime Distributions. This graph shows byte lifetime values calculated using a create-based and a delete-based algorithm. The trace used comprises the two contiguous days represented in the fourth Sprite trace (days 7 and 8); this trace showed the most difference between the two methods of all the Sprite traces. Unlike the results reported in [Bake91], these results include blocks overwritten in files that were not deleted, however this difference has only minor effects on the results.

4.2.2 Block Lifetime

Using the create-based metric for both our traces and the Sprite traces, we calculate block lifetimes using a block size of 512 bytes. Figure 2 shows these results. Block lifetime for a combination of the Sprite traces is included for comparison. Because most activity occurred during the second trace, this trace dominates Sprite's lifetime results. The graph shows a knee in the WEB workload that is mainly due to database working space files and `http` log files. RES has a knee at ten minutes caused primarily by periodic updates to Netscape database files. The Sprite trace has a knee just before five minutes contributed mainly by activity in the second trace. Since the Sprite traces do not include information on filenames, we do not know which files were deleted at that time. Neither INS nor NT has a knee; instead, block lifetimes gradually decrease after one second. Unlike the other workloads, NT shows a bimodal distribution pattern—nearly all blocks either die within a second or live longer than a day. Although only 30% of NT block writes die within a day, 86% of newly created files die within that timespan, so many of the long-lived blocks belong to large files. Some of the largest files resulted from newly installed software. Others were in temporary directories or in the recycle bins on hosts where the bin is not emptied immediately. Of the short-lived blocks, many belong to browser cache and database files, system registry and log files, and files in the recycle bin on hosts where users immediately empty the bin.

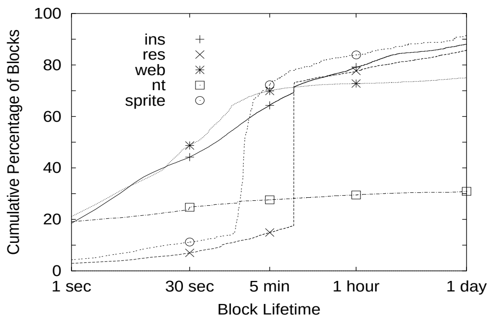

FIGURE 2. Block Lifetime. This graph shows create-based block lifetimes using a block size of 512 bytes. Points demarcate the 30 second, 5 minute, and 1 hour points in each curve. The end margin is set to 1 day for these results.

4.2.3 Lifetime Locality

By recording whether blocks die due to file deletion, truncation, or overwriting, we observe that most blocks die due to overwrites. For INS, 51% of blocks that are created and killed within the trace die due to overwriting; for RES, 91% are overwritten; for WEB, 97% are overwritten; for NT, 86% are overwritten. A closer examination of the data shows a high degree of locality in overwritten files. For INS, 3% of all files created during the trace are responsible for all overwrites. These files are overwritten an average of 15 times each. For RES, 2% of created files are overwritten, with each file overwritten an average of 160 times. For WEB, 5% of created files are overwritten, and the average number of overwrites for these files is over 6,300. For NT, 2% of created files are overwritten; these files are overwritten an average of 251 times each. In general, a relatively small set of files are repeatedly overwritten, causing many of the new writes and deletions.

An important result from this section is that average block lifetime is longer than delete-based lifetime estimates would predict. For some workloads, average block lifetime is significantly longer than the standard file system write delay of 30 seconds. Since it is unreasonable to leave data volatile for a longer period of time, file system designers will need to explore alternatives that will support fast writes for short-lived data. Some possibilities are NVRAM [Bake92] [Hitz94], reliable memory systems [Chen96], backing up data to the memory of another host, or logging data to disk. Most file blocks die in overwrites, and the locality of overwrites offers some predictability that may prove useful to the file system in determining its storage strategy.

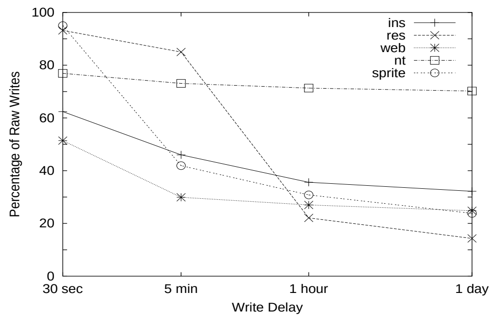

FIGURE 3. Write Bandwidth versus Write Delay. Using a simulated 16MB write buffer and varied write delay, we show the percentage of all writes that would be written to disk. For these results, we ignore calls to `sync` and `fsync`.

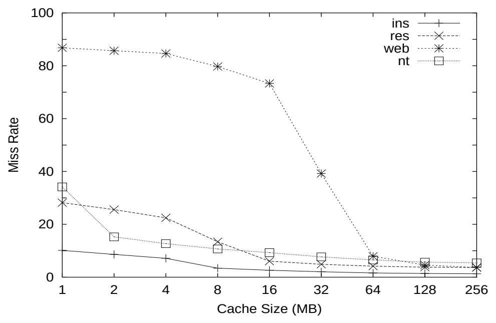

FIGURE 4. Read Bandwidth versus Cache Size. This graph shows the percentage of all block read requests that miss the cache versus cache size. The block size used by the cache simulator is 4KB. The cache was warmed with a day of traces before generating results.

4.3 Effect of Write Delay

Since newly written blocks often live longer than thirty seconds, increasing the write delay period should reduce disk write traffic. However, two factors limit the effectiveness of increasing write delay. First, user requests to `sync` and `fsync` cause data to be written to disk whether or not the write delay period has passed. Second, the operating system may limit the amount of dirty data that may be cached. This limit is generally imposed so that reading a new page into the cache is not slowed by the need to write out the old page first. On systems with NVRAM, the size limit is simply imposed by the NVRAM capacity. In either case, we refer to the space allocated to dirty pages as the write buffer.

In order to measure the effectiveness of increasing write delay, we simulated a write buffer and measured the resultant disk bandwidth while varying the write delay and the capacity of the buffer. Figure 3 shows the results using a 16MB buffer. For these results, we ignore calls to `sync` and `fsync`. As expected, the efficacy of increasing write delay is strongly related to the average block lifetime for each workload. Since RES has many blocks that live less than one hour, a one-hour write delay significantly throttles disk write traffic. On the other hand, the NT workload contains more long-lived blocks, so even write delays of a day have little effect.

To estimate the memory capacity needed to increase write delay, we tested write buffers of size 4MB and 16MB, and an infinitely-sized write buffer. For all workloads, the 16MB buffer closely approximates an infinitely-sized write buffer. In fact, for all workloads except Sprite, the 4MB write buffer also approximates an infinitely-sized write buffer. Large simulations included in the second Sprite trace (the third and fourth of the eight days) are probably responsible for the large write bandwidth. When these traces are omitted, the 4MB write buffer approximates an infinitely-sized buffer for the Sprite workload as well.

The importance of user calls to `sync` and `fsync` to flush data to reliable storage depends on the storage strategy employed. For example, a file system using NVRAM may ignore these calls since the data is already reliably stored. On other systems, the longer the data is kept in the write buffer, the stronger the impact of these calls. In our study, the maximal impact would be to the infinitely-sized write buffer with a write delay period of one day. For INS, calls to flush data increased writes to disk by 8% at this point; for RES, these calls increased write bandwidth by 6%. For NT, write bandwidth increased by 9%, and for WEB there was no increase at all.

In summary, the efficacy of increasing write delay depends on the average block lifetime of the workload. For nearly all workloads, a small write buffer is sufficient even for write delays of up to a day. User calls to flush data to disk have little effect on any workload.

4.4 Cache Efficacy

An important factor in file system performance is how effectively the cache absorbs read requests. In particular, we are interested in how effective caches are at reducing disk seeks and how caching affects the balance between disk reads and writes. In this section, we examine the effect of cache size on read misses. We find that even relatively small caches absorb most read traffic, but there are diminishing returns to using larger caches. We also examine how caching affects the ratio of disk reads to disk writes. In 1992, Rosenblum and Ousterhout claimed that large caches would avert most disk reads, so file system layout should optimize for disk writes [Rose92]. We find that the read to write ratio depends not only on the cache size, but also on the write delay and workload. Finally, we examine how well caching works for memory-mapped files. We find that because a small number of files tend to be memory-mapped by many processes, chances are high that these files will be cached.

4.4.1 Effect of Cache Size

We implemented a cache simulator to test the effectiveness of different cache sizes on read traffic. Both reads and writes enter blocks into the simulator, and blocks are replaced in LRU order. For all results in this section, we modeled a local cache, so each host maintains its own instance of the simulator.

Figure 4 shows the cache miss bandwidth for reads for various cache sizes. For all workloads, the curves have a knee showing the working set size, and there are diminishing benefits to increasing the cache size beyond this point. The WEB workload has the largest working set size; its read bandwidth does not reach the point of diminishing returns until a cache size of 64MB. Some of its poor performance may be due to the LRU replacement policy interacting poorly with the database engine. For the other workloads, even a 1MB cache reduces read bandwidth by 65–90%. For these workloads, there is little benefit to increasing the cache beyond 16MB. The BSD study predicted that in the future larger caches would significantly reduce disk reads. However, several years later, the Sprite study found that despite its large caches, read misses did not decrease as much as expected. Our results show that even very large caches have limited effectiveness in reducing read misses.

Since disk bandwidth is improving faster than disk latency, a critical metric in evaluating cache performance is the number of seeks caused by cache misses. Most file systems attempt to store blocks from the same file consecutively on disk. For example, FFS specifically allocates new file blocks as closely as possible to previous file blocks [McVo91]. In LFS, blocks are laid out in the order they are written [Rose92]. Since most files are written sequentially (as we show in Section 4.6), file blocks tend to be allocated consecutively on disk. If file blocks are laid out on disk consecutively, a rough estimate for the number of seeks incurred is a count of the disk reads to different files. We call this metric *file read misses* and calculate it as follows. Within a stream of cache misses, if a cache miss is to the same file as the previous cache miss, we count no file read miss; otherwise, we increment the number of file read misses by one. We define the *file write miss* metric analogously. Although these are crude metrics, we believe they are more accurate estimates of seeks than block miss counts.

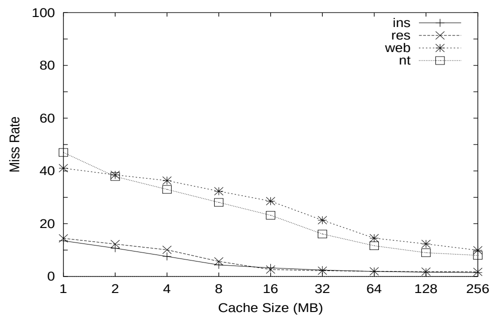

FIGURE 5. File Reads versus Cache Size. The miss rate is the percentage of file read misses out of the raw number of file reads. This graph shows the file miss rate for various cache sizes. The block size used by the cache simulator is 4KB. The cache was warmed with a day of traces before results were collected.

When multiple hosts share a single file system, a strict computation of the file read count requires interleaving the traces for those hosts. Because the INS and RES clusters share file servers for most of their file system activity, we were able to estimate the effect of file server sharing on file reads by running our measurements on these workloads using both a single interleaved trace for all hosts together and separate traces for each host. These two methods show at most a 2% difference in file read counts and no difference at all when the cache size is over 16MB. This may be because file system traffic tends to be bursty[Grib98]—bursts of activity from single streams may cause a series of cache misses near enough to each other in time that there are few intervening cache misses from other processes in the same time period.

In Figure 5, we show the effectiveness of different cache sizes on reducing the number of file read misses, using interleaved traces when applicable. The graph shows that even a 1MB cache is sufficient to more than halve the number of file read misses for all workloads. At the 1MB cache size, the WEB workload has many fewer file read misses than block read misses, which indicates that many block misses are part of larger files.

4.4.2 Read and Write Traffic

File systems lay out data on disk to optimize for reads [McKu84] or writes [Rose92] [Hitz94], depending on which type of traffic is likely to dominate. As we have already shown, the amount of disk write traffic depends largely on the write delay and the amount of read traffic depends on the cache size. In order to compare the amount of read and write traffic, we examine two envi-

TABLE 2. I/O Count

	INS	RES	WEB	NT
Impoverished Environment				
Block Reads	4,417,055	1,943,728	70,658,318	2,820,438
Block Writes	909,120	2,970,596	1,646,023	3,420,874
File Reads	620,752	199,436	2,389,988	330,528
File Writes	524,551	247,960	144,155	341,581
Enriched Environment				
Block Reads	2,114,991	613,077	6,544,037	1,761,339
Block Writes	1,510,163	585,768	1,483,862	3,155,584
File Reads	277,155	70,078	980,918	144,575
File Writes	209,113	101,621	64,246	248,883

In the impoverished environment, read results are based on an 8MB local cache and write results are based on a 16MB write buffer with a 30 second write delay. In the enriched environment, read results are based on a 64MB local cache, and write results are based on a 16MB write buffer with a 1 hour delay. In both environments, the block size is 4KB, and calls to `sync` and `fsync` flush the appropriate blocks to disk whether or not the write delay has elapsed.

ronments. The first environment has 8MB of local cache and a write delay of 30 seconds; we refer to this as the impoverished environment. The second, the enriched environment, has 64MB of local cache and a write delay of 1 hour. Read and write traffic for each environment are shown in Table 2. By looking at the number of blocks read and written, we see that reads dominate writes in all cases for the WEB workload. For the INS workload, the number of read blocks is almost five times the number of write blocks in the impoverished environment but is only about 50% greater in the enriched environment. For the RES workload, writes dominate the impoverished environment. In the enriched environment, there are more block reads than block writes but fewer file reads than writes. This is most likely caused by the large number of small writes made to various log files on the RES hosts. For the NT workload, writes dominate reads in all cases. However, most of the write traffic is caused by a single host. When this host is removed, reads dominate writes for all categories except file operations in the enriched environment.

Whether reads or writes dominate disk traffic varies significantly across workloads and environments. Based on these results, any general file system design must take into consideration the performance impact of both disk reads and disk writes.

4.4.3 Effect of Memory Mapping

Another important factor in cache performance is the effect of memory-mapped files. Over the last few years, memory mapping has become a common method to access files, especially shared libraries. To see the impact of memory mapping on process I/O, we counted the number of processes that memory-map files and the number that perform reads and writes. Table 3 summarizes these results. For all workloads, a greater number of processes memory-map files than perform reads or writes. With such a high number of processes accessing memory-mapped files, people designing or evaluating file systems should not ignore the effect of these files on the I/O system.

TABLE 3. Process I/O

	INS	RES	WEB	NT
Processes that Read	209050 (10%)	103331 (12%)	8236 (9%)	1933 (36%)
Processes that Write	110008 (5%)	80426 (9%)	18505 (19%)	1182 (22%)
Processes that Memory Map	1525704 (72%)	584465 (68%)	37466 (39%)	4609 (85%)

Processes are tracked via `fork` and `exit` system calls. For all workloads, more processes use memory-mapped files than read or write. Because the NT traces do not continuously record all fork and exit information, the NT results are based on a subset of the traces.

TABLE 4. Memory-mapped File Usage

	INS	RES	WEB
Avg. Mapped Files	43.4	17.6	7.4
Max. Mapped Files	91	47	10
Avg. Cache Space	23.2 MB	7.6 MB	2.4 MB
Max. Cache Space	41.2 MB	19.2 MB	3.0 MB
Cache Miss Rate	0.5%	1.5%	1.0%

For this data, each host maintains its own (unlimited size) cache of memory-mapped files, and only processes active on that host can affect the cache.

Because our traces only monitor calls to map and unmap files, we do not have information on how programs access these files. For example, the traces do not indicate which parts of a mapped file the program accesses via memory loads. Although we do not have the precise access patterns, we estimate the effect of memory mapped files on the cache based on process calls to `mmap`, `munmap`, `fork`, and `exit`. Unfortunately, because our traces do not contain a complete record for forks and exits for the NT workload, we cannot perform an accurate estimate for the NT workload. For the UNIX workloads, we estimated the effect of memory-mapped files on the cache by keeping a list of all files that are

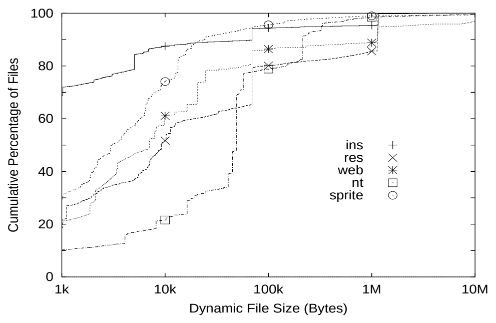

FIGURE 6. Dynamic File Size. We record file size for each accessed file when it is closed. If a file is opened and closed multiple times, we include the file in the graph data multiple times. Points depict sizes of 10KB, 100KB, and 1MB.

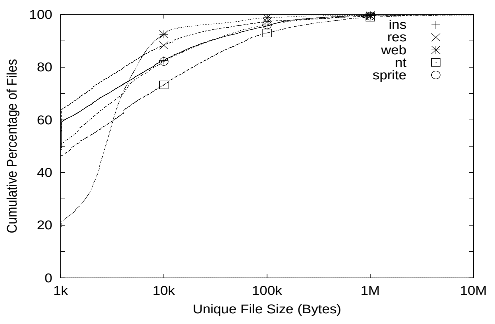

FIGURE 7. Unique File Size. We record file size at the time of file close. If a file is opened and closed multiple times, we only use the last such event in the graph data. Points depict sizes of 10KB, 100KB, and 1MB.

mapped either explicitly through a call to `mmap` or implicitly when a forked process inherits a file descriptor to a mapped file. We remove files from the list when no processes have the file mapped. Considering the number of `mmap` system calls, the average number of mapped files is quite low. The average and maximum number of files is shown in Table 4, along with the average and maximum space that would be required to keep the entire files in memory. We found that the same files tend to be mapped by many processes simultaneously. In fact, if the system kept each file in memory as long as at least one process mapped it, then cache miss rates for requests to map a file would only be about 1%.

4.5 File Size

Knowing the distribution of file sizes is important for designing metadata structures that efficiently support the range of file sizes commonly in use. The Sprite study found that most accessed files were small, but that the size of the largest files had increased since the BSD study. Our results show that this trend has continued.

In Figure 6, we show the file sizes across our workloads. In this graph, file size is determined *dynamically*—that is, file size is recorded for files as they are closed. With this methodology (also used in the Sprite study), files opened and closed multiple times are counted multiply. Like the Sprite study, we find that small files still comprise a large number of file accesses. The percentage of dynamically accessed files that are under 16KB is 88% for INS, 60% for RES, 63% for WEB, 24% for NT, and 86% for Sprite. At the other end of the spectrum, the number of accesses to large files has increased since the Sprite study. The number of files over 100KB accessed in Sprite is 4%, for INS it is 6%, for RES it is 20%, for WEB it is 14%, and for NT it is 21%. The largest file accessed in the Sprite traces is 38MB; the largest files in the other traces are an order of magnitude larger: from 244MB (WEB) to 419MB (INS and NT).

In addition to dynamic file size distribution, we examined unique file size distribution. By this we mean a distribution computed by counting each file that occurs in the trace only once. Of course, this does not include any files that are never accessed, since they are not recorded in the traces. This distribution reflects the range of file sizes stored on disk that are actively accessed. Figure 7 shows the results. Assuming a disk block size of 8KB and an inode structure with twelve direct data pointers, files over 96KB must use indirect pointers. The percentage of files over 96KB is 4% for INS, 3% for RES, 1% for WEB, 7% for NT, and 4% for Sprite.

The WEB workload has many files in the 1–10KB range. A large number of these are image files. Because these images are exported over the Internet, the WEB administrators limit the size of these files to keep access latency small. Except for the NT workload, the unique file size distribution has not become more skewed towards larger files since the time of Sprite. Although the NT traces are two years younger than the UNIX traces, we believe its larger files are due to differences in the operating system and applications rather than the time difference since the six years between the UNIX and Sprite traces show no appreciable effect.

Although the size of the largest files has increased tenfold since the Sprite study, the unique file distribution indicates that, except for the NT workload, the percentage of large files has not increased since the Sprite study. However, the dynamic distribution indicates that large files are accessed a greater percentage of the time. As a result, the number of file accesses that require indirect pointers has increased. Since this trend is likely to continue, it may be worthwhile to redesign the inode structure to more efficiently support access to large files. However, since most files are still small, their data structures must still efficiently handle file sizes for a broad spectrum of sizes. File systems that use extent-based or multiple block sizes [Powe77] [Hitz94] may be more efficient at handling the range of file sizes in use today.

4.6 File Access Patterns

In this section, we examine file access patterns—that is, whether a file is read or written and the order in which its bytes are accessed. Knowing common access patterns is crucial to optimizing file system performance. For example, knowing that most files are read in their entirety, many file systems implement a simple prefetching strategy that prefetches blocks in sequential order.

4.6.1 Run Patterns

We define a *run* as the accesses to a file that occur between its open and close. We classify runs into three categories. We classify a run as *entire* if it reads or writes a file once in order from beginning to end, *sequential* if it accesses the file sequentially but not from beginning to end, and *random* otherwise.

Table 5 compares file access patterns across workloads. Like Sprite and BSD, the majority of runs are reads and only a small percentage of runs contain both reads and writes. Also like the previous studies, most files are read in their entirety and most write runs are either entire or sequential. However, a higher percentage of runs are read-only in the HP-UX workloads than in NT, Sprite, or BSD. Also, our workloads tend to have a larger percentage of random reads than Sprite or BSD (the only exception being that BSD has a higher percentage of random runs than INS).

We examined random read patterns more closely and discovered a correlation between read pattern and file size. In Figure 8, we show the number of bytes transferred in entire, sequential, and random runs versus the size of the file being accessed. The graphs show that files that are less than 20KB are typically read in their entirety. For the Sprite workload, nearly all bytes are transferred in entire runs—even from very large files. However, for our workloads, large files tend to be read randomly. For INS, WEB, and NT, the majority of bytes from files over 100KB are accessed randomly. For RES, both entire runs and random runs are well-represented in bytes read from large files.

Most file systems are designed to provide good performance for sequential access to files. Prefetching strate-

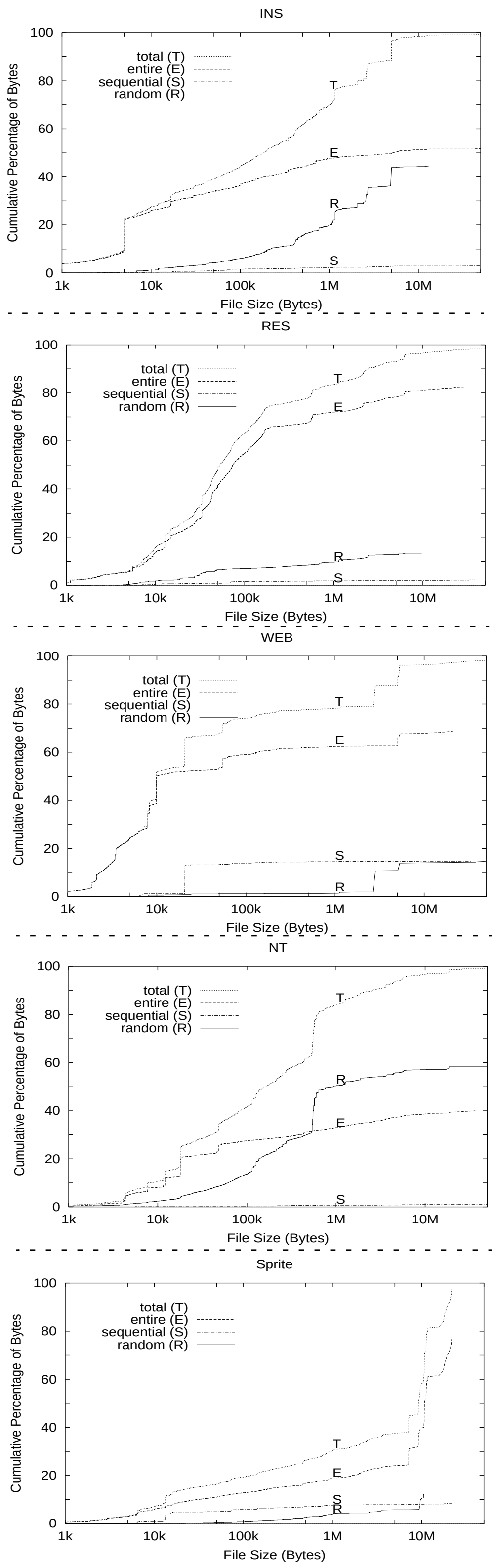

FIGURE 8. File Read Pattern versus File Size. In these graphs, we plot the cumulative percentage of all bytes transferred versus file size for all transferred bytes, those transferred in entire runs, those transferred in sequential runs, and those transferred in random runs.

TABLE 5. File Access Patterns

	INS	RES	WEB	NT	Sprite	BSD
Reads (% total runs)	98.7	91.0	99.7	73.8	83.5	64.5
Entire (% read runs)	86.3	53.0	68.2	64.6	72.5	67.1
Seq. (% read runs)	5.9	23.2	17.5	7.1	25.4	24.0
Rand. (% read runs)	7.8	23.8	14.3	28.3	2.1	8.9
Writes (% total runs)	1.1	2.9	0.0	23.5	15.4	27.5
Entire (% write runs)	84.7	81.0	32.1	41.6	67.0	82.5
Seq. (% write runs)	9.3	16.5	66.1	57.1	28.9	17.2
Rand. (% write runs)	6.0	2.5	1.8	1.3	4.0	0.3
Read-Write (% total runs)	0.2	6.1	0.3	2.7	1.1	7.9
Entire (% read-write runs)	0.1	0.0	0.0	15.9	0.1	NA
Seq. (% read-write runs)	0.2	0.3	0.0	0.3	0.0	NA
Rand. (% read-write runs)	99.6	99.7	100	83.8	99.9	75.1

A run is defined to be the read and write accesses that occur between an open and close pair. BSD results are from [Oust85].

gies often simply prefetch blocks of files that are being accessed sequentially [McVo91] [Sand85]. This provides little benefit to small files since there will not be many blocks to prefetch. If large files tend to be accessed randomly, this prefetching scheme may prove ineffective for large files as well, so more sophisticated prefetching techniques are necessary. Without effective prefetching, the increasing number of randomly read files may result in poor file system response time.

4.6.2 Read and Write Patterns

In Section 4.2, we noted that overwrites have significant locality—that is, the same files tend to get overwritten multiple times. In examining file access patterns, we noticed that read runs also have locality—that is, many files are repeatedly read without being written. To clarify how files are read and written, we tabulated for each file the number of runs that were read-only runs and the number that were write-only runs. (The number of read-write runs is negligible.) For each file, we calculated the percentage of its runs that were read-only. Files that are only read during the trace have 100% read runs, while files that are only written have 0% read runs. We rounded the percentage of read-runs to the nearest 10%; files having fewer than five runs are not included. We then added up the percentage of files that occurred in each percentage category. The results, shown in Figure 9, indicate that files tend to have a bimodal access pattern—they are either read-mostly or write-mostly. Furthermore, the larger the number of runs for a particular file, the stronger the affiliation. Many files tend to be read-mostly. This is evidenced by the large percentage of files that have 100% read runs. A small number of files are write-mostly. This is shown by the slight rise in the graphs at the 0% read-only point. Note that while the percentage of files in this category is small, these files have many runs each. Files that are both read and written have a read-run percentage between 0% and 100%; however, as the number of runs increases, fewer files fall into these middle categories.

5 Conclusions

We collected file system traces from several different environments, consisting of an instructional workload, a research workload, a web workload, and a Windows NT personal computer workload. We used these traces to compare the file system behavior of these systems to each other and to systems studied in past research. Based on this analysis, we draw the following conclusions.

First, different systems show different I/O load. The WEB workload has far more read bandwidth than any other workload but has relatively little write bandwidth. The NT workload has more than twice the read and write bandwidth as the other workloads.

Second, we found that average block lifetime, and even the distribution of block lifetimes, varies significantly across workloads. In the UNIX workloads, most newly created blocks die within an hour. In contrast, in the NT workload, newly created blocks that survive one second are likely to remain alive over a day. However, common to all workloads are that 1) overwrites cause the most significant fraction of deleted blocks, and 2) overwrites show substantial locality. Due to this locality, a small write buffer is sufficient to absorb write traffic for nearly all workloads. What differs from one workload to another is the ideal write delay: some workloads perform well with the standard 30-second write delay while others benefit from a slightly longer delay.

Third, we examined the effect of caching on read traffic. We found that even small caches can sharply decrease disk read traffic. However, our results do not support the claim that disk traffic is dominated by writes when large caches are employed. Whether this claim holds depends not only on the cache size, but also on the workload and write delay.

Fourth, we determined that all modern workloads use memory-mapping to a large extent. We examined how memory-mapping is used in the UNIX workloads and found that a small number of memory-mapped files are shared among many active processes. From this we conclude that if each file were kept in memory as long as it is memory-mapped by any process, the miss rate for file map requests would be extremely low.

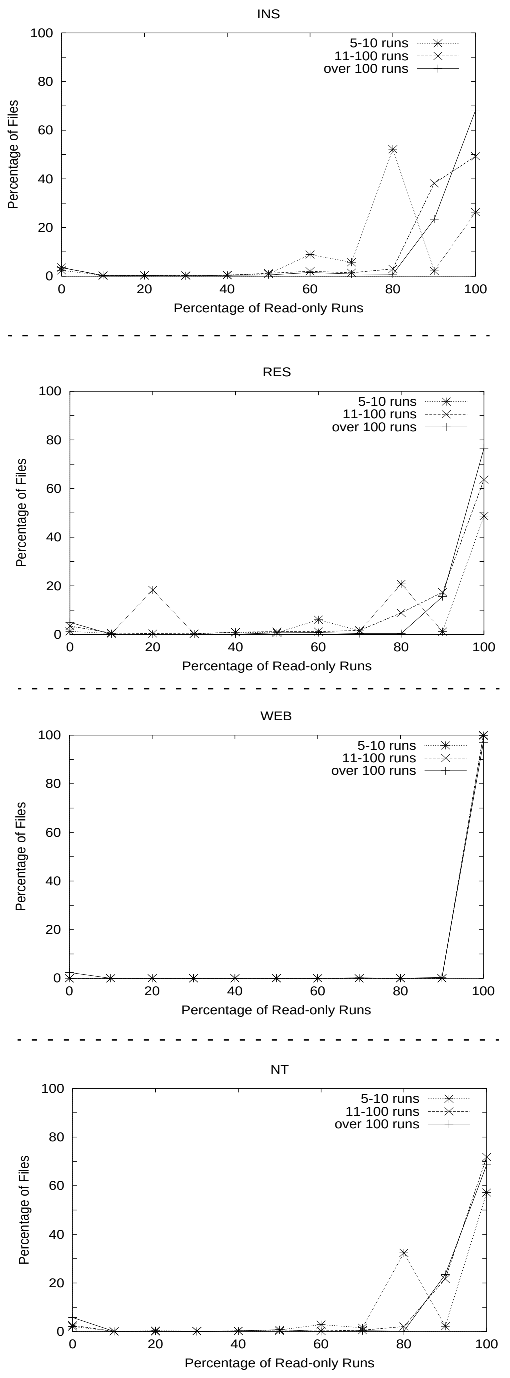

FIGURE 9. Percentage of Runs that are Read-only. Each line represents files categorized by the number of runs seen in the traces, where a run is defined to be all bytes transferred between the file's open and its close. The x-axis shows the percentage of runs that are read-only rounded to the nearest 10 percent. For each line, the percentages across the x-axis add to 100. Because most runs are read-mostly, the percentages are highest at the 100 percent read point, especially for files with many runs. A smaller number of files are write-mostly. These files appear at the 0 percent read runs point on the x-axis.

Fifth, we found that applications are accessing larger files than previously, and the maximum file size has increased in recent years. This is not surprising, as past studies have seen increases in file sizes as years passed. It might seem that increased accesses to large file sizes would lead to greater efficacy for simple readahead prefetching; however, we found that larger files are more likely to be accessed randomly than they used to be, rendering such straightforward prefetching less useful.

Finally, we found that for all workloads, file access patterns are bimodal in that most files tend to be mostly-read or mostly-written. We found this tendency to be especially strong for the files that are accessed most frequently. We expect file systems can make use of this knowledge to predict future file access patterns and optimize layout and access strategies accordingly.

Acknowledgments

Trace collection is a major undertaking accomplished with the aid of many of people. For the UNIX traces, Francisco Lopez provided invaluable assistance as the local HP-UX guru and as the main technical support contact for the instructional cluster; Eric Fraser created the infrastructure for the trace collection host; Kevin Mullally, Jeff Anderson-Lee, Ginger Ogle, Monesh Sharma, and Michael Short contributed to the installation and testing of the trace collection process. For the NT traces, we would like to thank Alan Jay Smith, Jee Pang, John Vert, Felipe Cabrera, Patrick Franklin, Gordon Chaffee, and Windsor Hsu for help in designing and testing the tracer.

We would especially like to thank the anonymous users from the University of California at Berkeley, FP Enterprises, Toshiba Corporation, and the Michigan State Police, who let us run our tracer on their work machines. Without their participation, this work would have been impossible.

For providing valuable feedback on this work, we would like to gratefully acknowledge Jeanna Matthews, Adam Costello, Eric Anderson, Keith Smith, Thomas Kroeger, and Steve Lumetta.

Trace Availability

The UNIX traces used for this paper are publicly available at http://tracehost.cs.berkeley.edu/traces.html.

References

[Bake91] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout, "Measurements of a Distributed File System," *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pp. 198–212, December 1991.

[Bake92] M. Baker and M. Sullivan, "The Recovery Box: Using Fast Recovery to Provide High Availability in the UNIX Environment," *Proceedings of the 1992 Summer USENIX Conference*, pp. 31–41, June 1992.

[Benn91] J. M. Bennett, M. Bauer, and D. Kinchlea, "Characteristics of Files in NFS Environments," *Proceedings of the 1991 Symposium on Small Systems*, pp 33–40, June 1991.

[Blaz92] M. Blaze, "NFS Tracing by Passive Network Monitoring," *Proceedings of the 1992 Winter USENIX Conference*, pp. 333–343, January 1992.

[Bozm91] G. Bozman, H. Ghannad, and E. Weinberger, "A Trace-Driven Study of CMS File References," *IBM Journal of Research and Development*, 35(5–6), pp. 815–828, September–November 1991.

[Chen96] P. Chen, W. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell, "The Rio File Cache: Surviving Operating System Crashes," *Proceedings of the Seventh ASPLOS Conference*, pp. 74–83, October 1996.

[Chia93] C. Chiang and M. Mutka, "Characteristics of User File-Usage Patterns," *Systems and Software*, 23(3), pp. 257–268, December 1993.

[Dahl94] M. Dahlin, C. Mather, R. Wang, T. Anderson, and D. Patterson, "A Quantitative Analysis of Cache Policies for Scalable Network File Systems," *Proceedings of the 1994 Sigmetrics Conference*, pp. 150–160, May 1994.

[Douc99] J. Douceur and W. Bolosky, "A Large-Scale Study of File-System Contents," *Proceedings of the 1999 Sigmetrics Conference*, pp. 59–70, June 1999.

[Floy89] R. Floyd and C. Schlatter Ellis, "Directory Reference Patterns in Hierarchical File Systems," *IEEE Transactions on Knowledge and Data Engineering*, 1(2), pp. 238–247, June 1989.

[Gang97] G. Ganger and M. F. Kaashoek, "Embedded Inodes and Explicit Groupings: Exploiting Disk Bandwidth for Small Files," *Proceedings of the USENIX Annual Technical Conference*, pp. 1–17, January 1997.

[Grib98] S. Gribble, G. Manku, D. Roselli, E. Brewer, T. Gibson, and E. Miller, "Self-Similarity in File Systems," *Proceedings of the 1998 Sigmetrics Conference*, pp. 141–150, June 1998.

[Hart93] J. Hartman and J. Ousterhout, "Corrections to Measurements of a Distributed File System," *Operating Systems Review*, 27(1), pp. 7–10, January 1993.

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

[Lorc00] J. Lorch and A. J. Smith, "Building VTrace, a Tracer for Windows NT," Accepted for publication in *MSDN Magazine*, September–October 2000.

[McKu84] M. McKusick, W. Joy, S. Leffler, and R. Fabry, "A Fast File System for UNIX," *ACM Transactions on Computer Systems*, 2(3), pp. 181–197, August 1984.

[McVo91] L. McVoy and S. Kleiman, "Extent-like Performance from a UNIX File System," *Proceedings of the 1991 Winter USENIX Conference*, pp. 33–44, January 1991.

[Mumm94] L. Mummert and M. Satyanarayanan, "Long-term Distributed File Reference Tracing: Implementation and Experience," *Software—Practice and Experience*, 26(6), pp. 705–736, November 1994.

[Oust85] J. Ousterhout, H. Da Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson, "A Trace-Driven Analysis of the UNIX 4.2 BSD File System," *Proceedings of the Tenth Symposium on Operating Systems Principles*, pp. 15–24, December 1985.

[Powe77] M. Powell, "The DEMOS File System," *Proceedings of the Sixth Symposium on Operating Systems Principles*, pp. 39–40, November 1977.

[Rose98] D. Roselli, "Characteristics of File System Workloads," University of California at Berkeley Technical Report CSD-98-1029, December 1998.

[Rose92] M. Rosenblum and J. Ousterhout, "The Design and Implementation of a Log-Structured File System for UNIX," *ACM Transactions on Computer Systems*, 10(1), pp. 26–52, February 1992.

[Ruem93] C. Ruemmler and J. Wilkes, "UNIX Disk Access Patterns," *Proceedings of 1993 Winter USENIX Conference*, CA, January 1993.

[Russ97a] M. Russinovich and B. Cogswell, "Windows NT System-Call Hooking," *Dr. Dobb's Journal*, 22(1), pp. 42–46, January 1997.

[Russ97b] M. Russinovich and B. Cogswell, "Examining the Windows NT Filesystem," *Dr. Dobb's Journal*, 22(2), pp. 42–50, February 1997.

[Sand85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and Implementation of the Sun Network Filesystem," *Proceedings of the 1985 Summer USENIX Conference*, pp. 119–130, June 1985.

[Saty81] M. Satyanarayanan, "A Study of File Sizes and Functional Lifetimes," *Proceedings of the Eighth Symposium on Operating System Principles*, pp. 96–108, December 1981.

[Sien94] T. Sienknecht, R. Friedrich, J. Martinka, and P. Friedenbach, "The Implications of Distributed Data in a Commercial Environment on the Design of Hierarchical Storage Management," *Performance Evaluation*, 20, pp. 3–25, May 1994.

[Smit81] A. J. Smith, "Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms," *IEEE Transactions on Software Engineering*, SE-7(4), pp. 403–416, July 1981.

[Voge99] W. Vogels, "File System Usage in Windows NT 4.0", *Proceedings of the Seventeenth Symposium on Operating Systems Principles*, pp 93–109, December 1999.

[Zhou99] M. Zhou and A. J. Smith, "Analysis of Personal Computer Workloads," *Proceedings of the Seventh International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 208–217, October 1999.