1. Trace Collection

He is a master storyteller, the only one I ever heard who could tell a whole story with only two grammatical subjects.

"Them sons-of-bitches," he said, opening with his first subject, "was Mennonites and wouldn't fight in the last war—said they wasn't afraid to work or die for their country but wouldn't kill anybody, so somebody, maybe for this somebody's idea of a joke, had them sent to the Smokejumpers. It turned out them sons-of-bitches was farm boys and, what's more, didn't believe in using machines no way— working was just for their hands and their horses, and them sons-of-bitches took them shovels and saws and Pulaskis and put a hump in their backs and never straightened up until morning when they had a fire-line around the whole damn fire. Them sons-of-bitches was the world's champion firefighters."

His second grammatical subject he saved for the end. "The rest of us bastards," he said, "was dead by midnight."

—*Norman Maclean,* Young Men and Fire

1.1 Introduction

Our goal is to evaluate disk layout policies. Because layout matures over weeks and months, we need long-term file system and disk usage data. There are three primary methods for

generating file system requests: synthetic generation, benchmarks, and traces. Synthetic generation is primarily useful when the characteristics of the workload are already understood. One of the goals of our study is to discover long-term characteristics to allow synthetic models to be constructed. Likewise, benchmarks are useful only if one knows in advance that the chosen benchmarks are representative of the actual workload and even then most benchmarks are too short for our purposes. We could synthetically generate long-term workloads by repeatedly running short benchmarks, but this is unlikely to produce realistic results since it does not model changes in input over time. In order to understand the characteristics of long-term file system traffic, we must first examine data from actual usage.

However, evaluation by replaying traces has its own set of difficulties. Traces are necessarily bound to the environment from which they are collected, and because computing environments change rapidly, the traces must be recent. In addition, most available traces are too short to evaluate long-term file system behavior. For these reasons, we began collecting our own traces in the summer of 1996. We currently have traces varying in length from one month to one year from three different environments.

In this chapter, we describe the procedure we used to collect traces from three different environments. After our traces were collected, a more recent set of traces was collected from a different environment by Jacob Lorch. For many of the studies conducted in this work, we use these traces to supplement our own. We describe these traces at the end of this chapter.

1.2 Related Efforts in Trace Collection

Traces are frequently used to evaluate file system innovations, yet relatively few traces are publicly available. This is due to both privacy concerns and the amount of effort required to collect traces. 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.

There are several levels at which traces can be taken ranging from static disk snapshots to dynamic kernel tracing. In general, all trace collection efforts have to choose between completeness and complexity. In the following sections, we describe different collection methods used and the benefits and drawbacks associated with each.

The simplest traces consist of static snapshots of all files on disk. This method can be implemented easily by running a `find` command to collect system metadata at one or several frozen instants in time. Studies of this type are useful for studying distributions of file attributes commonly stored in metadata, such as the directory structure and file name, size, last access time, and last modification time [DB99, SFMF94, CM93, BBK91, Sat81, Smi81]. The disadvantages to static traces are that they are not sufficient to study cache behavior, detailed file access patterns, block level activity, or short-term behavior.

In comparison with static snapshots, traces that record the file system as it runs contain more detailed information on file system activity but are more difficult to collect. The

simplest of these dynamic tracing techniques are non-invasive, such as snooping NFS traffic off the network [Bla92, DMW+94]. While these traces capture dynamic file system activity, they are incomplete because they miss requests handled on the local hosts, such as requests that hit in the cache. Also, artifacts of the network file system being measured can affect these types of traces. For example, NFS [SGK+85] generates file system requests to implement its cache coherence protocol.

Similarly, by interposing trace collection at the disk interface level, Ruemmler et al. collected traces outside the file system [RW93]. These traces were taken from three HP-UX servers for approximately 4.5 months and include only disk operations, not the higher level file system calls. These traces are unique in tracing actual disk access patterns. They clearly show the amount of disk accesses caused by metadata operations and virtual memory operations. However, because the traces are dependent on the particular file system running on the traced machines, they are not useful for evaluating alternative file system designs above the disk level.

Capturing complete file system activity at the file system level yields more detailed information about file system usage. However, 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. Third, the overhead of collecting fine-grained traces must be kept low so that overall system performance is not significantly degraded. Finally, because these modifications are usually non-portable, the same work must be repeated for every platform traced. 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 [FE89] do not include file read or write requests. Mummert et al. focused on results relevant to disconnected file system operation [MS94]. Zhou and Smith collected traces on personal computers for research in low-power computing [ZS99].

In 1985, Ousterhout et al. collected dynamic traces from three servers running BSD UNIX for slightly over three days [OCH+85]. Because the traces contained detailed access patterns, this study was able to introduce a number of metrics to characterize file system workloads. These metrics include run length, burstiness, lifetime of newly written bytes, and file access sequentiality. 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 [BHK+91]. 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. These sets of traces include all read and write activity although the exact times of the reads and writes is unknown. Metadata operations are not included. 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 [BGW91]. This study confirmed that the results from the Sprite study applied to non-academic sites. In 1998, these studies were repeated on our own workloads [Ros98], and, in 1999, they were again repeated on three sets of two-week traces taken from 45 hosts running Windows NT [Vog99]. 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. Although the level of detail in these sets of traces is sufficient for characterizing short-term file system behavior, the traces are not long enough to study long-term file system characteristics.

A combination of features distinguish our traces from the above work. First, our traces are long term. Because file system factors such as disk fragmentation mature over time [MJLF84] [SS96], long-term traces are necessary to evaluate disk layout policies. Second, our traces include three different workloads and include detailed traces of a database web server. Third, individual reads and writes are recorded so that detailed information on access patterns is available. Fourth, since metadata operations have a significant impact on disk requests [RW93], our traces include metadata operations. In addition, our traces include full pathnames for file references and `exec` system calls are traced so that it is possible to investigate applications that cause file system activity. Finally, because our traces were recently collected, we can compare our results with those of previous studies to illustrate the change in patterns in file system activity over time.

1.3 Trace Collection

Our purpose in collecting traces was to study the effect of different layout policies over time. To ensure that our traces could fulfill this purpose, we set these goals for our traces:

1. The traces should be long-term: weeks to months in duration.

2. The traces should be independent of the file system used on the traced hosts and

should include all information necessary for relevant layout policy decisions.

3. Trace collection should have minimal impact on users.

4. The trace collection mechanism should be scalable to a large number of hosts.

5. The traces should be taken from a variety of environments.

We explored several alternative methods for trace collection and ultimately chose the auditing system as the closest fit to these goals. In the following sections, we first present an overview of the auditing system and then discuss how we designed the system to meet each of our goals.

1.3.1 Methodology: The Auditing System

There are several levels at which traces can be taken: in the kernel, between the application and the kernel, and on the network between client and server machines.

The simplest method is to monitor network traffic between clients and servers. However, because file system requests sent over the network have already been processed by the client's file system, these traces are both incomplete and contain artifacts of the clients' file system.

Tracing between the application and kernel involves snooping all application requests to the kernel. This method can be easily implemented on some operating systems, for example, those that support a `/proc` file system. Unfortunately, the machines available to us for

tracing ran the HP-UX 9.05 operating, which has no such support. Another approach would be to compile a trace collection library with all applications to be traced. We ruled out this approach since it would inconvenience users and would miss any applications that were not re-compiled with the trace collection library.

For these reasons, we chose to collect traces at the kernel level despite the complexity of this approach. To minimize kernel changes, we used the auditing subsystem to record file system events. Many operating systems include an auditing subsystem for security purposes. The auditing subsystem gets invoked after a system call is issued and before the call executed. It can be configured to log specified system calls with their arguments and return values. This is ideal for tracing since it catches the logical level of requests using already existent kernel functionality. Unfortunately, it does not record kernel file system activity, such as the paging of executables. The major problem we faced in using the auditing system was that the HP-UX 9.05 version records pathnames exactly as specified by the user. Users often specify paths relative to their current working directory rather than the complete path. Since some disk layout policies use a file's parent directory, we needed to record the full pathname. We solved this problem by recording the current working directory 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.

The benefits of using the auditing system as a tracing tool are twofold; it provides a method of collecting a complete set of file system calls with low overhead and requires only small

changes to the kernel. In fact, for systems that implement the auditing system in such a way that full pathnames are recorded for security monitoring purposes, any type of system call could be traced without modifying the kernel at all.

Advantages of collecting the traces on the clients rather on the server are that local cache hits are included and no burden is added to the server which is often already a bottleneck in networked file systems. But recording on the client requires the tracing code to be installed on many machines and complicates processing since all the individual traces must be merged. In our environment, no machines that acted as file servers were traced. However, unless all clients are traced, it is possible to miss accesses to files from non-traced hosts. Although we did not collect traces on all clients, in the clusters traced, most users tend to use the same host (or set of hosts) regularly, so it is likely that most activity for a given user of the cluster is collected.

Since the traces are taken at the system call level, file system requests internal to the kernel are not recorded. As a result, pathname lookup and reads and writes to executables and memory-mapped regions are not included. Measurements by Ruemmler and Wilkes show disk paging to swap partitions to be 0.4–1.8% of all disk traffic on hosts well-endowed with memory and 16.5% on low-memory hosts [RW93]. However the impact of reading executables and memory-mapped files is unknown.

Finally, although our original intention in recording complete pathnames was to record the parent directory, having this information proved invaluable to understanding the traces and the applications that cause specific file traffic patterns.

time
host id
process id
user id
system call number
length of all arguments
argument 1
...
argument n

Table 1.1: **Trace Record Format.** All trace records contain a header followed by any arguments. The number and type of arguments is determined by the system call type. A complete list of system calls traced and their arguments is shown in Table 1.7 at the end of this chapter. For the unprocessed traces, the arguments are the same as those specified by the system call. The system call number and argument length fields are each two bytes long. All other fields are four bytes long.

The format of the trace records is shown in Table 1.3.1. Each record contains a header followed by arguments. The header is a fixed length and contains six fields. The first field contains the time the record was generated measured in seconds since January 1, 1970. The following fields identify the host, user, and process that generated the record. The next field contains the system call type. The system call type determines the number and type of any arguments that follow the header. The last header field contains the length in bytes of all the arguments for the record. In the raw traces, the arguments are those that are provided by the system call. During postprocessing (described in the next section), some of the arguments are changed. For example, file descriptor arguments are replaced by identifiers for the files they reference. The complete list of traced calls and their arguments is enumerated in Table 1.7 at the end of this chapter.

1.3.2 Long-term Tracing

In order to make long-term trace collection feasible, we had to engineer the trace collection process so that it could run continuously for several months without interruption.

On each traced host, we installed a modified kernel that wrote the auditing system records to a local file. To ensure tracing continued to run over an extended period of time while the hosts were constant use, we installed scripts on these hosts to manage the trace collection process. These scripts checked for problems during tracing, restarted tracing after reboots, and automatically migrated completed trace files to a collection host.

1.3.3 File System Independence

Because we traced at the system call level, our traces capture requests at a logical rather than physical level. For example, each read request includes the exact byte range accessed but not whether the request was cached or where the data blocks were located on disk.

We tried to include enough information in the traces so that they could be used with both existing layout policies and any novel policies that we might wish to explore. To this end, we included all file system requests, including metadata operations, and the complete pathnames for all relevant requests.

1.3.4 Impact on Client Resources

In order for trace collection to be tolerated by the users, it must have minimal impact on the performance and resource usage of the machine being traced. We measured the overhead of our tracing system on a compilation workload. Originally, the tracing code added 11% overhead compared with the same benchmark without tracing. However, by buffering trace records in the kernel and writing them out in large batches, we were able to reduce the overhead to 1%.

On each host being traced, the trace log files were changed and compressed every hour by a `cron` script. The compressed traces used an average of 3.2MB of local disk space per day. The trace files were migrated off the clients nightly over an Ethernet to a collection host. Hosts sent their trace files at staggered one minute intervals so that the network and receiving host were not overwhelmed. Assuming an Ethernet's effective bandwidth to be 5Mbps, sending 3.2MB per minute uses less than 10% of the network capacity.

In terms of storage, the compressed traces require on average 3.2MB per host per day. Therefore, tracing 150 hosts for 6 months requires less than 100GB of disk space.

Hosts migrate their trace files to the trace collection server at a rate of one host per minute. Since only a few networks are shared among all the hosts and these are heavily used during the day, trace migration is done during the night. The least busy period for the traced machines occurred between 4am and 7am. Since this period contains 180 minutes, we could scale up to 180 hosts using this method.

1.3.5 Environment

In order to meet our goal of tracing a variety of environments, we installed tracing on three separate groups of Hewlett-Packard series 700 workstations running HP-UX 9.05. The first group consisted of twenty machines located in laboratories for undergraduate classes. These machines were used for editing, compiling, and running programs for class projects, as well as for email, document processing and web browsing. 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. These hosts were used for a wide variety of tasks including document processing, program development, graphically displaying research results, email, and web browsing. The third set of traces was collected 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 receives approximately 2300 accesses per day. We refer to this workload as the WEB workload. The INS machines mount home directories and common binaries from a non-traced Hewlett-Packard workstation. All other machines mount home directories and common binaries from an untraced file server over an Ethernet. 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.

Each host had 64MB of memory. The file cache size on these hosts at the time of trace collection is unknown. However, since HP-UX 10 allows a maximum of half of memory to be used by the file cache, we can assume that the file cache was maximally 32MB.

1.4 Postprocessing

To simplify replay, the trace files are postprocessed into a format more easily managed by trace analysis programs. The main tasks of the postprocessor are to assign unique identifiers to files, match file descriptors to file identifiers, and fix a number of problems in the raw traces. By handling these tasks in the postprocessor, we were able to greatly simplify programs that analyze the trace data.

One difficulty with the raw traces is that information tends to be distributed over several trace records that may or may not appear near each other in the traces. For example, many system calls have file descriptors as arguments. In order to match the file descriptor to the actual file to which it refers, one must find the `open` or `creat` record that generated the file descriptor. Tracking file descriptors is complicated by system calls that copy or change file descriptors and by inheritance of file descriptors by child processes. Many of the tasks of the postprocessor involve compiling this distributed information and recording it with all relevant records.

The postprocessor also fills in incomplete information in the trace records generated by the auditing system, described later. Most errors in this category affect not only the use

of the records for tracing but would also adversely affect any system that uses the traces for the security purposes for which the auditing system was designed.

1.4.1 Challenges

Because our traces were collected over a period of many months, we could not wait for trace collection to complete before we started analyzing the trace data. Otherwise, not only would valuable research time be wasted, but problems in the trace collection process could go undetected. Because the postprocessor is a complex set of programs, it took many iterations to debug. Obvious problems were easily detected and fixed by test programs. However, with such a large quantity of traces, even highly improbable events eventually occur. These cases are considerably more difficult to identify and correct since many of them only occur after postprocessing several months worth of data. In order to analyze recently collected traces and allow adequate time to fix problems with the postprocessor, the postprocessing routines have to be able to complete at a much greater rate than the traces are collected. As a result, the postprocessing routines have to be extremely efficient.

To speed postprocessing, we ported the postprocessor to a cluster of workstations. We divided the postprocessing routines into two major phases: a parallel phase and a sequential phase. The parallel phase was performed on the individual client traces before they were merged together. To maximize parallelism, as much work as possible is done in this phase. The traces are then merged together by timestamp and the sequential phase begins. The only operations that must be done sequentially are the assignment of file identifiers, *fileids*,

to nonlocal file system operations and the mapping of file descriptors to these fileids. Obviously, it is the sequential phase that limits the scalability of postprocessing. The sequential phase processes one day of traces for 30 hosts in about 30 minutes. This phase scales linearly with the number of hosts, so it could process one day of traces from 150 hosts in about 2.5 hours.

Although distributing the processing improves performance, it also creates problems, notably synchronization between machines. The configuration files passed between machines are mounted on networked file systems. Because the NFS protocol used does not guarantee consistency across different machines, these files could be read differently on different machines. Although this problem rarely occurs, because of the large volume of data processed, it typically does occur at some unpredictable point before postprocessing has completed. To avoid this problem, each program has to verify the validity of its configuration files before reading them.

Similarly, file accesses during postprocessing that time-out due to network congestion or the tardiness of the automounter could cause the postprocessing scripts to fail. Although these errors are extremely rare, the large input size combined with the high load on the processing machines combines to make the probability of an error occurring during postprocessing more likely than not. Therefore, in order to make progress, the postprocessing routines must periodically checkpoint their results.

The tasks performed by the postprocessor require a great deal of memory. The machine designated for the sequential portion has 1GB of memory. In order to run the sequential

portion of postprocessing in under 1GB of memory, every byte in every data structure must be carefully allocated.

In summary, the postprocessor can be thought of as being *programming complete*. Because of the immense number of operations and high load, obscure errors are not only possible but likely—requiring the postprocessor to be highly robust. It has to use little memory and have fast algorithms and requires efficient I/O libraries. It has to implement its own synchronization and fault tolerance. However, the postprocessor succeeded in its goal to simplify the trace analysis programs. Of the scores of such programs used to generate the results in this report, none was as difficult to write as the postprocessor.

1.4.2 Tasks

Some of the tasks of the postprocessor are described in more detail in this section.

Corrupt Records

The first task of the postprocessor is to fix errors in the raw traces. One of the most insidious problems is handling corrupted records. Corrupted records are caused when the logging of an audit record is interrupted by another system call or when the auditing system is switched on or off. To eliminate these corrupted records, the postprocessor checks the range of values of several header and argument fields. Any record containing a field with an invalid value is removed. We detect that less than 0.01% of the trace records are corrupted. While this rate is acceptably low, a single error can cause the rest of the trace file to become

unreadable. To prevent this cascading effect, the postprocessor scans the following trace input one byte at a time until it finds the next valid header.

Clock Adjustments

To appropriately handle access to shared files, the trace records are sorted by global time. The time value in each record header is set from the system clock on each machine. Ideally, this value would be non-decreasing throughout the traces, however, since administrators and system daemons can reset the clocks on their machines, this is not always the case. Although clock adjustments were infrequent in all traces, occasionally several clock resets would occur in succession on the INS cluster. The INS cluster uses a network daemon to synchronize the clocks of all machines in the cluster. The network daemon records the synchronized time in a file. Periodically, another daemon reads this value and updates the system clock. If the network daemon dies or is unable to contact the cluster time server, the time stored in the file is not updated and the other daemon continuously resets the system clock to the stale time value recorded in the file. The times in trace records logged during these events jump back repeatedly, however, the records are in the correct chronological order on each machine. For this case, the actual time is likely to be within a second of the time of the last record. The postprocessor changes the time in these records to be the same as the time in the last record because it produces a better estimate of the actual time and prevents time from regressing. However, since the actual time may be later than the previous record's time and because some measurements are sensitive to the number of events that occur within a second, times corrected in this manner are marked

by negating the time value. Programs that require accurate counts of events per second can ignore time spans that contain records with negative time values.

Tracing Artifacts

Another task of the postprocessor is to remove trace records caused by the trace collection process. Writing individual trace records to the log file occurs in the kernel and so is not traced. However, several processes that manage the trace log files run at user level. These processes check that there is enough disk space for the trace records, switch the trace log files, and send completed trace files to a collection server. Each of these processes is started by the `cron` daemon. The postprocessor recognizes these processes by the file name used for execution. It marks the process identifier and removes all records with this process identifier until the process exits. Child processes forked by a marked process are also marked and excised from the traces. After postprocessing, only the fork and exit records remain for each script. The exit records are left to simplify trace processing routines that track forked processes. The fork calls could be removed by buffering trace records until the process executes a program, however, since an unbounded number of records from other processes can intervene between a process's fork and its execution call, removing these records would significantly complicate the postprocessor. We have measured the residual records to be only 0.04% of the postprocessed traces on an average machine, so we believe that leaving these records in the traces is acceptable.

Pathnames

As part of our trace collection goals, we require complete pathnames for all files in the trace records. As previously mentioned, we track the current working directory of all processes in the kernel's auditing system so that relative pathnames can be translated into full pathnames as the traces are collected. However, two system calls can disrupt this process. The `chroot` system call changes the root directory for the calling process and all of its children. To simplify the trace recording code in the kernel, pathname updates caused by `chroot` calls are performed by the postprocessor. The other system call that disrupts the kernel's knowledge of the current working directory is the `fchdir` call. This call is equivalent to the call to change directories (`chdir`) except that it takes a file descriptor as an argument rather than a pathname. Because the pathname referenced by the file descriptor could not be found in the kernel without making changes outside of the auditing system, this task was deferred to the postprocessor. The postprocessor tracks all `fchdir` calls, matches the file descriptor to the appropriate complete directory path, and updates the pathnames of the calling process and all of its children.

Unique File Identifiers

The postprocessor assigns a unique file identifier to each file accessed in the traces. The postprocessor identifies unique files based on their file system, inode number, and pathname.

The simplest way to identify unique files is by using the file's pathname. However, there

are several problems with this approach. First, the same pathname on different hosts may refer to different files. For example, the filename `/tmp/foo` refers to a different file on each host. Further, shared files can be referred to by different names depending on their mount points. Second, symbolic and hard links may refer to the same file by different pathnames. Because the traced file systems contain a large number of symbolic links, this would introduce a non-negligible number of errors. Third, because the pathnames are large, tracking all files by the full pathname would significantly increase the memory footprint of the postprocessor.

Another approach to tracking unique files is to use the file's inode number. Because each file system has its own set of inode numbers, the postprocessor must first determine the file system for the file.

Each pathname is assigned a file system number (or *device number*) by consulting a static table of mount paths. This is the least automated and therefore most fragile part of post-processing; it requires checking the mount tables of each client by hand and incorporating the mounted systems into the postprocessing code. Files accessed through symbolic links that span file systems are incorrectly recorded as if the data were stored on the file system of the symbolic link rather than the file system where the data actually reside. We measured the number of such symbolic links on several of our file systems and found them to be less than 1% of all files, so we do not believe this inaccuracy significantly affects results generated from the traces.

Once the device has been determined, a file identifier is assigned using the file system's

inode numbers to distinguish files. A problem with using inode numbers to uniquely identify files is that, in the HP-UX auditing system, trace records for files created by the system calls `creat` or `open` contain the inode number of the file's parent directory rather than that of the file itself. For files created via the `creat` system call, we assign a unique identifier to the file and record the file's pathname. The next time a record with this pathname is processed, the postprocessor updates its database with the correct inode number to unique file identifier. The situation is more complex for `open` calls with the mode argument set to `CREATE`. These files may or may not exist. If the file already exists, then the record is a lookup open and the inode number is correct for this file. However, if the file does not already exist, the file is created and the inode number is that of its parent directory. For the assignment of the file identifier, the postprocessor uses several heuristics to distinguish these cases. For example, if the inode number already exists and is known to refer a directory, the opened file must be a create. If the inode number already exists and is known to refer to a file, the opened file must be a lookup. If the postprocessor cannot determine whether the `open` is a create or a lookup, it is assumed to be a create. If incorrect, this assumption causes only minimal inaccuracies in the traces. For example, if the file was accessed previously in the traces, then previous accesses will appear to be to a different file. However, since these previous accesses did not provide the postprocessor with sufficient information to determine whether the file is a file or directory, these previous accesses could only have been produced by a small set of system calls that only access the file's metadata. These calls are: `symlink, rename, chmod, chown, utime, access, stat, lstat, getacl, setacl, getaccess, lstat`. If we had reversed our policy and assumed

the open were a lookup, then all following operations on the file would be incorrectly assumed to be to the file's parent directory. Further, if the same error were made for other files in the same directory, multiple files and their parent directory would appear to be to a single file.

Mapping File Descriptors to File Identifiers

Many file system calls use a process-specific file descriptor as an argument rather than the filename. The postprocessor tracks all mappings from file descriptor to file identifier and replaces file descriptor arguments with the appropriate device-fileid pair.

After this translation, the system calls `dup`, `dup2`, and `fcntl` are removed from the traces since they only map file descriptors onto other file descriptors and do not contain file access information.

If the postprocessor cannot match a file descriptor to a fileid, the fileid is set to the sentinel value `UNKNOWN`. Many unknown files are caused by `fstat` calls to files opened by system processes that start before the auditing system when the host is booted; because an `fstat` call only contains a file descriptor, if the file's open is missed, subsequent calls cannot be matched to the file to which the descriptor refers. The number of records referring to unknown files is 3% for INS, 10% for RES, and 0.1% for WEB. For RES, 97% of records containing unknown files are `fstat` calls. For all workloads, the number of reads and writes to unknown files is less than 1%.

Offset

Unprocessed read and write records have as arguments only the file descriptor and number of bytes accessed. The file offset is stored with the file descriptor data structure in the kernel. The postprocessor tracks the file offset for each open file descriptor and adds it to the read and write records. Afterwards, the `lseek` call, which does not access data but simply changes the file position, is removed from the traces.

Closes

For each open file on a host, the kernel maintains an open file data structure. A number of different file descriptors may refer to this data structure. When the final reference has been removed, the data structure is released.

Releasing references to file descriptors occurs either explicitly, through the `close` system call, or implicitly, for example, when a process exits or a file descriptor overwrites another file descriptor.

The postprocessor tracks all references to open files and both implicit and explicit closes. It adds an argument to the `close` system call indicating whether the close releases the final file reference for the host or process. If the final reference to a file is closed implicitly, an implicit close record is added to the trace.

Privacy

In order to preserve privacy, the user identifier is altered by a one-way mapping function as the traces are processed. However, user identifiers zero through ten are not altered since these accounts are administrative in nature rather than personal and may provide insight into the trace analysis. Because pathnames themselves may compromise privacy, once a file is given a unique identifier, its pathname is removed from the traces and stored in a separate file.

1.5 Windows NT Traces

In addition to our own traces, we use traces collected by Jacob Lorch [LS00]. These traces were collected from eight desktop machines running Windows NT 4.0. Two of these machines are 450MHz Pentium IIIs, two are 200MHz Pentium Pros, and the other four are Pentium IIs ranging from 266–400MHz. Five of them have 128MB of main memory, while the others have 64, 96, and 256MB. 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, e-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, e-mail 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.

1.6 Windows NT Collection Methodology

Lorch collected the Windows NT traces using a tool he developed that traces not only file system activity, but also a wide range of device and process behavior [LS00]. We focus here on the aspects of the tracer relevant to file system activity.

Lorch performs 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. This 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.

A Windows NT optimization called the *fastpath* 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, Lorch implemented his own fast-dispatch functions to record any calls made this way.

There were some challenges in converting these traces to a format comparable with the UNIX traces because they are taken at the file system level rather than the system call level. 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 want to elide the recursive requests because they do not reflect actual read requests. 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 separating a reads and writes caused by explicit requests from those caused by kernel activity. We distinguish kernel-initiated read-ahead 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. 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.

Finally, 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.

1.7 Summary

For all large systems, the issue of scale not only affects the effort involved but also the fundamental design. Every piece of the trace collection and processing routines required additional support code to ensure progress in spite of errors.

system call	call number	arguments in processes traces
exit	1	none
fork	2	pid flag
read	3	dev fid offset bytes
write	4	dev fid offset bytes
open	5	dev fid fd ftype fstype owner size nlinks ctime mtime atime mode
close	6	dev fid mode
creat	8	dev fid fd ftype fstype owner size nlinks ctime mtime atime mode
link	9	dev fid
unlink	10	dev fid
execv	11	dev fid
chdir	12	dev fid
chmod	15	dev fid
chown	16	dev fid
lseek	19	removed from processed traces
smount	21	dev fid
umount	22	dev fid
utime	30	dev fid
access	33	dev fid
sync	36	none
stat	38	dev fid
lstat	40	dev fid
dup	41	removed from processed traces
reboot	55	none
symlink	56	dev fid
rdlink	58	dev fid
execve	59	dev fid
chroot	61	dev fid
fcntl	62	removed from processed traces
vfork	66	pid flag
mmap	71	dev fid
munmap	73	dev fid
dup2	90	removed from processed traces
fstat	92	dev fid
fsync	95	dev fid
readv	120	dev fid offset bytes
writev	121	dev fid offset bytes
fchown	123	dev fid
fchmod	124	dev fid
rename	128	dev fid
trunc	129	dev fid newsize
ftrunc	130	dev fid newsize
mkdir	136	dev fid
rmdir	137	dev fid
lockf	155	dev fid function size
lsync	178	none
getdirentries	195	dev fid
vfsmount	198	dev fid
getacl	235	dev fid
fgetacl	236	dev fid
setacl	237	dev fid
fsetacl	238	dev fid
getaccess	249	dev fid
fsctl	250	removed from processed traces
tsync	267	dev fid
fchdir	272	dev fid

Table 1.2: **System Calls Traced** This table shows all traced calls and their arguments after postprocessing.

1.8 Notes on Arguments

Most calls in Table 1.2 have the arguments *dev* and *fid*, where dev is the device number of the file system and fid is a unique file identifier on that system. A given device-fileid pair is unique throughout the trace.

The `fork` and `vfork` calls have two arguments: a process identifier and a flag which is nonzero if this is the child. If this is the child, the first argument is the parent's process id. If this is the parent, the first argument is the child's process id.

The `open` and `creat` calls were modified to record additional information. They have ten arguments in addition to the device and fileid. The *fd* field is the file descriptor returned by the call, the *ftype* is the file type (regular, directory, symbolic link, or empty directory), the *fstype* is the type of file system (local or NFS), owner is the user identifier of the file's owner, size is the file size in bytes, *nlinks* is the number of hard links, *ctime*, *mtime*, and *atime* refer to the last inode change time, modify time and access time respectively. The mode refers to the open/create mode.

The mode field on the `close` call indicates whether this is the final close for this file.

Bibliography

[BBK91] J. M. Bennett, M. Bauer, and D. Kinchlea. Characteristics of Files in NFS Environments. In *Proceedings of the 1991 Symposium on Small Systems*, pages 3–40, June 1991.

[BGW91] G. Bozman, H. Ghannad, and E. Weinberger. A Trace-Driven Study of CMS File References. *IBM Journal of Research and Development*, 35(5–6):815–828, September–November 1991.

[BHK+91] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout. Measurements of a Distributed File System. In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pages 198–212, December 1991.

[Bla92] M. Blaze. NFS Tracing by Passive Network Monitoring. In *Proceedings of the 1992 Winter USENIX Conference*, pages 333–343, January 1992.

[CM93] C. Chiang and M. Mutka. Characteristics of User File-Usage Patterns. *Systems and Software*, 23(3):257–268, December 1993.

[DB99] J. Douceur and W. Bolosky. A Large-Scale Study of File-System Contents. In *Proceedings of the 1999 Sigmetrics Conference*, pages 59–70, June 1999.

[DMW+94] M. Dahlin, C. Mather, R. Wang, T. Anderson, and D. Patterson. A Quantitative Analysis of Cache Policies for Scalable Network File Systems. In *Proceedings of the 1994 Sigmetrics Conference*, pages 150–160, May 1994.

[FE89] R. Floyd and C. Schlatter Ellis. Directory Reference Patterns in Hierarchical File Systems. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):238–247, June 1989.

[LS00] J. Lorch and A. J. Smith. Building VTrace, a Tracer for Windows nt. *Accepted for publication in MSDN Magazine*, September–October 2000.

[MJLF84] M. McKusick, W. Joy, S. Leffler, and R. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[MS94] L. Mummert and M. Satyanarayanan. Long-term Distributed File Reference Tracing: Implementation and Experience. *Software-Practice and Experience*, 26(6):705–736, November 1994.

[OCH+85] J. Ousterhout, H. Da Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson. A Trace-Driven Analysis of the UNIX 4.2 bsd File System. In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pages 15–24, December 1985.

[Ros98] D. Roselli. Characteristics of File System Workloads. Technical Report CSD-98-1029, University of California at Berkeley, December 1998.

[RW93] C. Ruemmler and J. Wilkes. UNIX Disk Access Patterns. In *Proceedings of 1993 Winter USENIX Conference*, January 1993.

[Sat81] M. Satyanarayanan. A Study of File Sizes and Functional Lifetimes. In *Proceedings of the Eighth Symposium on Operating System Principles*, pages 96–108, December 1981.

[SFMF94] 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*, 20:3–25, May 1994.

[SGK+85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the 1985 Summer USENIX Conference*, pages 119–130, June 1985.

[Smi81] A. J. Smith. Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms. *IEEE Transactions on Software Engineering*, SE–7(4):403–416, July 1981.

[SS96] K. Smith and M. Seltzer. A Comparison of FFS Disk Allocation Polices. In *Proceedings of the 1996 USENIX Technical Conference*, pages 15–26, January 1996.

[Vog99] W. Vogels. File System Usage in Windows NT 4.0. In *Proceedings of the Seventeenth Symposium on Operating Systems Principles*, pages 3–109, December 1999.

[ZS99] M. Zhou and A. J. Smith. Analysis of Personal Computer Workloads. In *Proceedings of the Seventh International Symposium on Modeling Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 208–217, October 1999.