A Trace-Driven Analysis of Name and Attribute Caching in a Distributed System

Ken W. Shirriff
John K. Ousterhout

Computer Science Division
Electrical Engineering and Computer Sciences
University of California at Berkeley
Berkeley, CA 94720

Abstract

This paper presents the results of simulating file name and attribute caching on client machines in a distributed file system. The simulation used trace data gathered on a network of about 40 workstations. Caching was found to be advantageous: a cache on each client containing just 10 directories had a 91% hit rate on name lookups. Entry-based name caches (holding individual directory entries) had poorer performance for several reasons, resulting in a maximum hit rate of about 83%. File attribute caching obtained a 90% hit rate with a cache on each machine of the attributes for 30 files. The simulations show that maintaining cache consistency between machines is not a significant problem; only 1 in 400 name component lookups required invalidation of a remotely cached entry. Process migration to remote machines had little effect on caching. Caching was less successful in heavily shared and modified directories such as `/tmp`, but there weren't enough references to `/tmp` overall to affect the results significantly. We estimate that adding name and attribute caching to the Sprite operating system could reduce server load by 36% and the number of network packets by 30%.

1. Introduction

Operating systems spend much of their time performing path name lookups to convert symbolic path names to file identifiers. In order to reduce the cost of name lookups, many systems have implemented name caching schemes. For instance, Leffler et al [LKM84] measured performance on a single-machine system running 4.2BSD Unix and found path name translation to be the single most expensive function performed by the kernel, requiring 19% of kernel CPU cycles. When name caching was added to 4.3BSD Unix, it reduced name translation costs by 35%.

Name lookup is an even larger problem in distributed systems, where a client machine may have to contact a file server across the network to perform the name lookup. Most network file systems cache naming information on client workstations as well as on servers [HBM89, HKM88, WPE83]. This allows clients to perform most name lookups without contacting the server, which improves the speed of lookups by as much as an order of magnitude. In addition, client-level name caching reduces the load on the server and the network. If client-level name caches are combined with caches of file data, they may even allow a client machine to continue operating when the file server is unavailable [KiS91].

Unfortunately, there has been little data published on the measured performance of name caches in distributed file systems. Floyd et al. [Flo86, FlE89] and Sheltzer et al. [SLP86] performed trace-driven studies of name cache performance, and both concluded that relatively small name caches produce relatively high hit ratios. However, Floyd studied a single time-shared system, and Sheltzer studied a small collection of networked time-shared machines where many accesses were to local files. Distributed systems with large numbers of diskless workstations have a number of characteristics that might interfere with name caching:

- In a distributed environment, the name caches on different machines must be kept consistent. This will result in extra network messages and cost that is not needed on a single time-shared system.
- In a distributed environment, the most important overhead is the communication time involved in server requests; the actual operations on the server often take less time than the basic network communication.

- Name caching schemes typically require a separate server request for each component that is not in the client's name cache, and typical path names contain several components. In contrast, a system without name caching can pass the entire path name to the server in a single operation (i.e. there can never be more than one server request per lookup). This means that an individual lookup operation could take substantially longer with a client-level name cache than without one.
- A name cache is usually accompanied by a separate cache of file attributes such as permissions, file size, etc. The entries in the attribute cache are typically managed separately from entries in the name cache, resulting in additional server requests.
- Most implementations of name caching use a whole-directory approach, meaning they cache entire directories. However, this may not work well with load-sharing techniques where a single user spawns processes on many machines simultaneously. If those processes work in a single directory then there may be a substantial amount of overhead to keep the cached directory consistent on the multiple machines.
- Highly-shared directories such as the UNIX `/tmp` directory may also add to the overhead of maintaining name cache consistency.

Because of these concerns, we performed a trace-driven analysis of name caching in a distributed environment. We collected traces of name and attribute usage in a network of about 35 diskless workstations and 5 file servers. We then wrote simulators to analyze and compare the effectiveness of several different methods of name and attribute caching. Our approach differs from previous work primarily in that we use diskless workstations as the source of trace data and we examine effects such as load-sharing that were not present in previous studies.

Our study confirms previous studies that caching names and attributes is highly effective (see Table 1). We found that a high hit rate can be obtained with a relatively small cache. For instance, caching 20 whole directories on each client (about 20 kilobytes of storage) resulted in a 97% hit rate for pathname component lookups. An entry-based name cache, which caches individual directory entires, had poorer performance, having a hit rate of 81%. The attribute cache had a 91% hit rate with a cache of the attributes for 40 files on each client.

We found minimal problems with maintaining cache consistency across multiple machines. To our surprise, we found that process migration does not have a significant effect on name and attribute caching, even though migrated processes account for an average of 19% of lookups. (Process migration is a mechanism in Sprite used to move processes to idle machines for parallel execution [DoO91].) There was almost no difference in cache performance between simulations with process migration and without migration. Consistency overhead was small; only a small amount of network traffic was required to keep the caches consistent across multiple machines, as shown by the low invalidation rate in Table 1. This is because very few operations required a remotely cached entry to be invalidated, and most invalidations only invalidated one other machine's copy.

The remainder of the paper is structured as follows: Section 2 describes the trace data we collected. Section 3 presents the results of the cache simulations. Section 4 consists of a discussion of the results and our conclusions.

Cache type	Hit rate	Remote invalidation rate
Whole-directory name cache (20 directories cached)	.97	0.0022
Entry-based name cache (40 directory entries cached)	.81	0.0004
Attribute cache (40 attributes cached)	.91	0.0005

Table 1: Summary of results. This shows the hit rate and the invalidation rate with client caching of directories and attributes, for a reasonably sized cache. The hit rate is the fraction of cache accesses that are found in the cache. The remote invalidation rate is the average number of cache entries on remote machines that must be invalidated, per cache access. We performed eight traces; these results are averages.

2. Collection of data

2.1. The Sprite system

We performed our name and attribute cache tracing on Sprite, a network-based operating system [OCD88]. Sprite provides a Unix-like environment in a network of about 40 workstations. Files are stored on one of several file servers and may be cached on the clients, with full consistency maintained among the cached copies. One important aspect of Sprite with respect to this study is that Sprite encourages sharing, both of files and of processors. We wished to examine the effect of this sharing on name and attribute caching.

Sprite provides a process migration facility, which allows processes to be moved across the network to idle machines [DoO91]. This permits users to take advantage of the processing power of several machines at once. There are currently two main uses of process migration in Sprite: parallel compilation and large simulations. We suspected that process migration and name caching were incompatible; contention among shared directories would cause name caching to perform poorly, we thought. As will be shown in Section 3.6, these concerns were unfounded.

In order to judge the applicability of our results to other systems, it is important to understand our computational environment and workload. Our measurements were taken on a Sprite system with about 50 users using Suns and DECstations. The users were distributed among several different academic research groups and engaged in various office/engineering tasks. Significant applications included electronic communication, typesetting, editing, software development and compilation, VLSI circuit design, graphics, and simulations.

2.2. The trace data

We collected eight one-day traces of activity on the Sprite system. These traces consisted of log records of file system activity, collected on our file servers. More information on the trace data is available in [BHK91]. Table 2 gives an overview of the trace data we used for this study.

There were three types of trace records used in this study. The most important was the lookup record, which logged a path name lookup. Each lookup record contained the file identifier of each examined component of the path name. The record also included the client machine requesting the lookup, migration information (if the request was from a migrated process), the operation responsible for the lookup, and whether or not the lookup succeeded. The second type of trace record traced opens and closes and was used to keep track of what files were open. The third type of record traced operations to get and set file attributes (e.g., `fstat`, `fchmod`).

Some interesting statistics on the traces are available from Table 2. On average, there were 16 name lookups per second. The number of name component accesses was a factor of 3.2 higher; this resulted from the multiple name component accesses required for each name lookup. An average of 19% of the lookups resulted from migrated processes, although this was much higher in some traces (the last trace had 48% migrated lookups). We also found that there were very few operations that resulted in modifications of names or attributes.

Table 3 shows statistics about the kernel calls that result in name lookups. Note that `open` and `stat` operations account for most of the lookup operations. This is fortunate since these operations benefit most from successful name caching. The other operations modify the file system, and thus will likely contact the file server regardless of the name lookup. Table 3 shows that a significant fraction of path name lookups terminate with an invalid name (i.e., ‘‘file not found’’ error). (Besides typographical errors, one major source of invalid names is search paths, which search through multiple directories for commands or include files.) About 14% of the lookup operations in Table 3 resulted from lookups being repeated on multiple file servers, due to a characteristic of Sprite file server operations called *redirects*. Since the file system is partitioned across several file servers, name lookups occasionally pass from the part of name space stored on one server to another server (usually due to a symbolic link). In this case, a redirect occurs and the client must submit the remaining part of the lookup to the new server.

We also measured the distribution of directory sizes in order to estimate the storage requirements for the whole-directory cache. Figure 1 shows the static distribution of directory sizes, obtained from a scan of the file system after one trace had completed. The distribution of directory sizes is important in estimating how much memory is required to cache directories. Since the average directory requires about 1 kilobyte of storage, the memory requirements for directory caching are quite modest, with a 20 directory cache taking around 20 kilobytes, if it

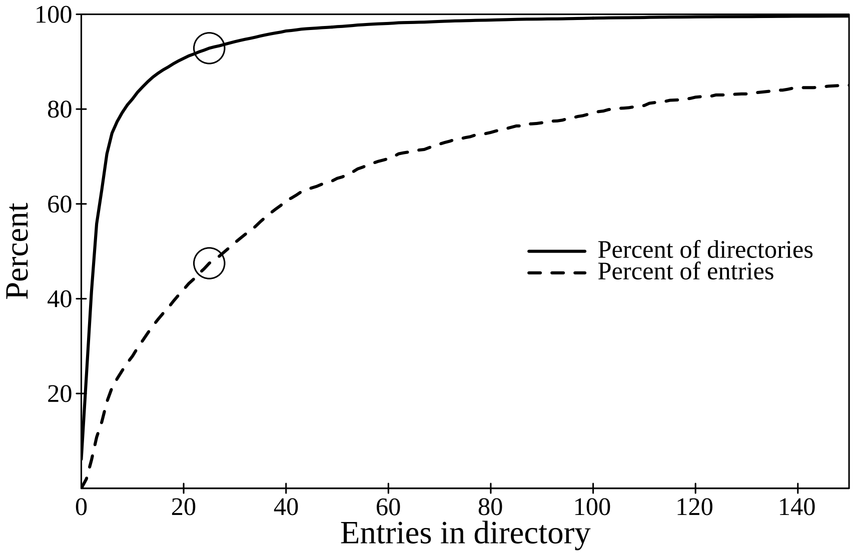

Figure 1: Static size distribution of directories. This graph shows the static distribution of directory sizes, calculated over all directories in the file system. The directory size is the number of entries in the directory (excluding ‘‘.’’ and ‘‘..’’). The upper curve shows the cumulative percentage of directories of each size. The lower curve shows the directory size distribution weighted by the number of entries in the directory. This shows what cumulative percentage of files and subdirectories are in directories of the specified size. (For example, the circled points show that about 93% of all directories had fewer than 25 entries, but these directories held under 50% of all directory entries.) Our measurements also showed that the average number of entries per directory was 8.9 and the average size of a directory was 1.1 1-Kbyte blocks.

---

holds average-sized directories. (Admittedly, the cached directories could be much larger than average. However, an examination of some common directories shows them to be only a few kilobytes.) The size of a directory cache is noteworthy in comparison to file system data caches in Sprite, which may hold several megabytes of data. Our measurements correlate well with those in [FlE89], which found a 10 directory cache was equivalent to about 14 kilobytes.

3. Simulations of name and attribute caching

3.1. About the simulator

We constructed a simulator that used the trace data to estimate the effectiveness of various caching schemes for file names and attributes. The client caches were assumed to have a least-recently-used (LRU) replacement policy: a cache of *n* directories holds the *n* most recently accessed directories. The simulator functioned by taking the trace data, determining the resulting low-level name operations, and simulating the effects of these operations on the client caches. Each trace record corresponded to several low-level operations; these operations were: look up a name in a directory, look up an attribute, modify a name, modify an attribute, remove a name and attribute, create a name and attribute, and read a directory. Each low-level operation caused an access to some of the cache LRU lists. When a cache entry was accessed, it was moved to the front of the appropriate machine’s LRU list. When an entry was modified, it was invalidated from the caches of all other machines.

We used several techniques to keep the simulation to a reasonable size and run time. The simulator used a stack-based model [Hil87] in order to simulate multiple cache sizes in one simulation run. To keep the simulation state from growing excessively, we pruned the LRU lists at regular intervals. The simulator scanned all the LRU lists, discarding idle entries. (We defined an entry as idle if it had not been used in the past 10 minutes and it was more than 20 entries down on the LRU list.) Measurements on smaller trace files showed that this pruning of LRU lists had little effect on the simulation results.

3.2. Name caching simulations

We simulated two types of name cache: a whole-directory cache and an entry-based name cache. For the whole-directory cache, each machine cached a fixed number of directories. The cache used the directory identifier as a key and returned the entire directory. With an entry-based cache, machines cached individual directory entries instead of whole directories. That is, the cache used the parent directory identifier and a symbolic component name as a key and returned the file identifier of the component.

There are several potential advantages of an entry-based cache over a whole-directory cache. One advantage of the entry-based cache is that cache performance may be better in a directory with a high update rate (such as `/tmp`). With a whole-directory cache, any change to any entry in the directory will result in the entire directory being invalidated from the cache on other machines. However, in an entry-based cache, only the modified entry will be invalidated; all other cached entries will remain valid. Another advantage to an entry-based cache over a whole-directory cache is that only the directory entries being used need to be cached. This may result in a higher hit rate for an entry-based cache than for a similarly sized whole-directory cache.

The entry-based name cache also has several disadvantages compared to the whole-directory cache. A major disadvantage is that it does not distinguish between cache misses and nonexistent entries. As shown in Table 3, a significant fraction of lookups try to access a nonexistent file or directory. With the entry-based name cache, when a path name component is not found in the cache the server must be contacted to determine if the component does not exist or if it is just not present in the cache. (One could cache invalid names as well as valid names. However, depending on the reference patterns of invalid names, this might not be effective. It would also add complexity to maintaining consistency, since an invalid name must be removed from the cache in the event that the corresponding file is later created.) A second disadvantage of the entry-based cache is that the cache doesn't help the performance of whole-directory reads. Some operations, such as listing a directory's contents, require reading the entire directory. These operations can benefit from a whole-directory cache, but not from an entry-based cache. Finally, an entry-based cache won't benefit from locality of directory accesses as much as the whole-directory cache will. If there are many references to different entries in a directory, a entry-based cache will have a miss for each entry. On the other hand, a whole-directory cache would load the directory once and subsequent references to entries would be hits.

We assumed that the name and attribute caches were kept strongly consistent. That is, the caches never were allowed to contain stale data. We simulated a callback mechanism similar to the one used in Andrew [HKM88] to maintain consistency. For the callback mechanism, the server keeps a record of what data is cached on each client. When another client modifies data, it must inform the server, which then calls back all clients with cached copies of that data. The clients then invalidate their stale data.

Several other cache consistency mechanisms are possible. For instance, consistency can be loosened, allowing clients to have inconsistent cached data. In the Echo system, on the upper levels (close to the root) of the file system, clients invalidate cached name data after several hours. Until the data is invalidated, clients can access inconsistent data. As another alternative, some NFS implementations [SGK85] uses a probabilistic scheme, in which cached names and attributes are considered invalid after a certain length of time. This time varies between 3 and 60 seconds, and is selected based on prior reference patterns of the file.

In our cache simulations, each name component that was found in the client's cache was counted as a cache hit. If the component was not found, it was counted as a miss. We divided misses into several categories (based on the categories used in [HeP90]):

- Compulsory misses are misses that would occur in an arbitrarily large cache: the first access to each directory or entry causes a compulsory miss.
- Capacity misses are name references that are misses due to the size of the cache; they would have been hits in a suitably large cache.
- Consistency misses consist of names that were in the cache, but had to be invalidated due to modification on other machines.

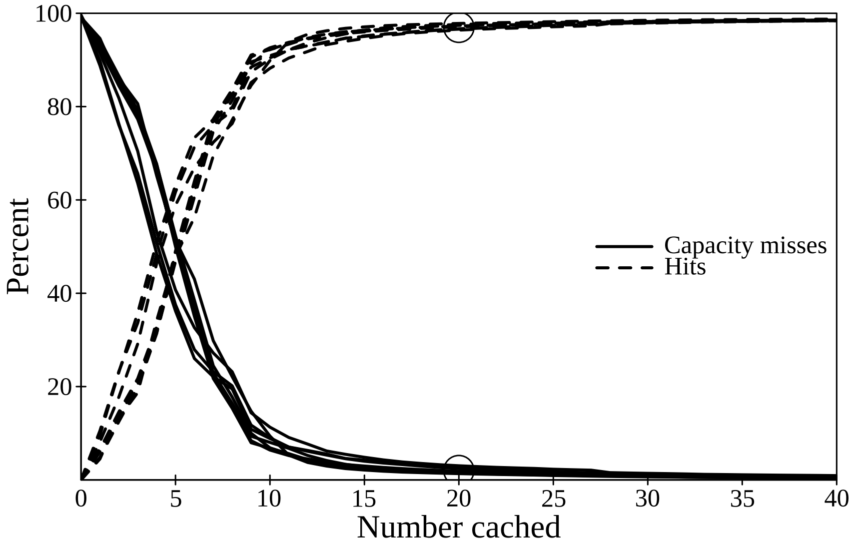

Figure 2: Whole-directory name cache performance. This graph shows the hit rate and capacity miss rate for accesses to the directory cache. There are separate lines for each of the eight traces. The X axis shows the number of directories cached on each client. The Y axis shows the percent of cache references that were hits or capacity misses. For instance, the circled points show that with 20 cached directories, the hit rate was about 97% and the capacity miss rate was about 2%. Capacity misses are misses that occur due to the cache size. There are two other types of misses which aren't graphed: compulsory misses and consistency misses. Compulsory misses occur the first time a directory is referenced, and cause the directory to be loaded into the cache. Consistency misses result from references that would have hit in the cache, except the entries were invalidated because they were modified on another machine. The compulsory miss rate for the different traces was between 0.6% and 0.9%. The consistency miss rate was between 0% and 0.3%. Note that the compulsory miss rate is constant for all cache sizes, while the consistency miss rate varies.

- Invalid-file misses are references to nonexistent files or directories. These count as misses in the entry-based cache, since, as explained above the entry-based cache does not distinguish between entries that are not cached and entries that don't exist. These references are not necessarily misses in the whole-directory name cache or attribute cache measurements.

3.3. Whole-directory name cache results

According to the measurements shown in Figure 2, whole-directory name caching is highly effective. A cache of 10 directories had a hit rate of 91%. Caching 20 directories increased the hit rate to about 97%. About 0.7% of the misses were compulsory misses, resulting from entries that were never referenced before. We found a low rate of consistency misses (about 0.2%), which shows that it is very rare to have contention due to modifications to a shared directory. We expect this is because users tend to work in different directories, and don't usually modify shared directories (with a few exceptions, such as `/tmp`, described in Section 3.8).

One question we had was how the component hit rate compared to the hit rate for entire paths. (An entire path is counted as a hit if every component in the path is in the cache.) If cache misses were uniformly distributed, the hit rate for entire paths would be much lower than the component hit rate, since an *n*-component path would have a hit rate equal to the component hit rate raised to the *n*th power. However, Figure 3 shows that the hit rate for entire paths is higher than would be predicted from the component hit rate, and in fact is close to the component hit rate. One explanation is that the component hit rate is significantly higher for short paths than for long paths. This biases the entire path hit rate to be better than would be expected, since long path names are more likely to have multiple component misses that only account for a single path miss.

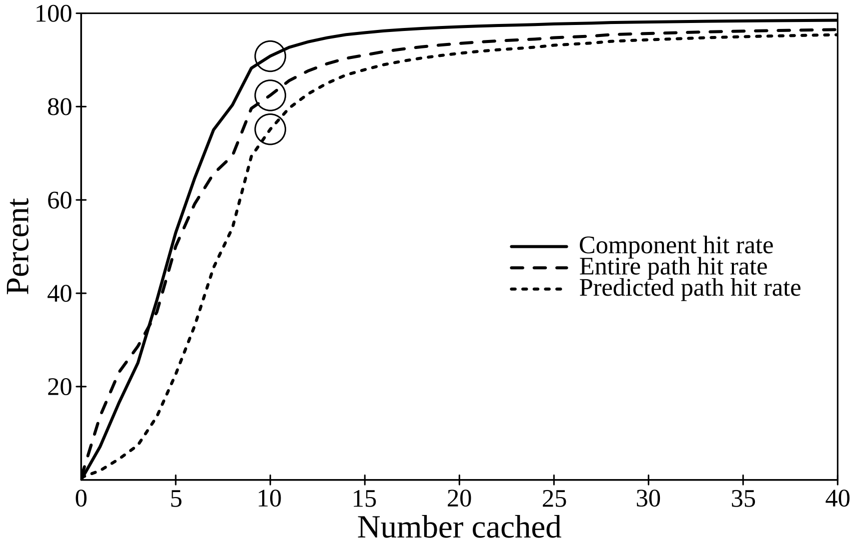

Figure 3: Entire path hit rate. This graph shows the measured name component hit rate, the measured entire path hit rate, and the entire path hit rate predicted from the component hit rate. The entire path hit rate is the fraction of paths that have every path component in the cache. The predicted hit rate is derived from the assumption that the component misses are uniformly distributed. This graph shows that the entire path hit rate was significantly better than predicted, especially for small cache sizes. For instance, the circled points show that for a 10 element cache, the hit rate for each component was 91%. Assuming this hit rate applies equally to all components, the average path would have a 75% chance of being entirely in the cache. However, the average path actually had 82% chance of being entirely in the cache. This graph shows averages over all eight traces.

Our name cache performance results are close to those of other papers, even though our computing environment is different. For instance, Floyd et al. [FlE89] found an 85% hit rate with a 10 directory cache, and a 95% hit rate on a 30 directory cache. These hit rates are close to ours, even though their measurements were on a single, multi-user machine. Sheltzer et al. [SLP86] found a 15-directory cache reduced the whole-path miss ratio from 71% to 11%. (Even without caching, many of the path name lookups could be completed locally because each machine in the 15-site VAX network stored part of the file system.) The hit ratio with 40 cached directories ranged from 87% to 96%.

3.4. Entry-based name cache results

The second type of cache we simulated was an entry-based name cache, which caches individual directory entries as opposed to whole directories. Cache results for the entry-based name cache are given in Figure 4. Note that the entry-based cache has poorer performance than the whole-directory cache. One reason is that each item in the whole-directory cache corresponds to several directory entries in the entry-based cache. (Figure 1 shows about 8.9 entries per directory.) However, even after scaling the cache sizes by this value, the entry-based name cache still has poorer performance than the whole-directory cache. There are several reasons for this. The entry-based cache has a much higher compulsory miss rate, because each referenced entry in a directory requires a separate cache miss to be loaded into the cache. On the other hand, the first whole-directory cache miss loads the entire directory into the cache. There appears to be substantial locality of access within a directory, so the whole-directory approach provides a significant performance advantage.

A second reason for the poorer performance of the entry-based cache is that it can't handle invalid names, since the entry-based cache can't distinguish between a name that isn't in the cache and a name that doesn't exist in the file system. These references are described in Figure 4 as ‘‘invalid-file misses’’.

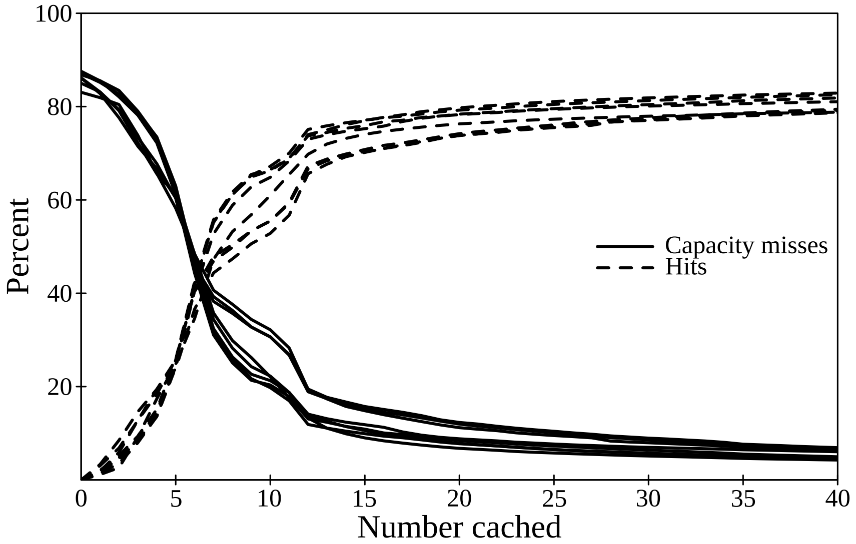

Figure 4: Entry-based name cache performance. This shows the hit rate for accesses to the component name cache, where the directory holds individual components. This graph has a separate line for each of the eight traces. A reference to a nonexistent path name component is called an invalid-file miss; this reference will result in a miss in the entry cache. The compulsory miss rate was 7.6% ± 2%; the invalid-file miss rate was 6.3% ± 3%; the consistency miss rate was negligible. Note that the compulsory and invalid-file miss rates do not depend on the cache size.

The entry-based name cache has another disadvantage besides its lower hit rate: read operations on directories can be satisfied by the whole-directory cache. (The contents of a directory are directly read by commands such as `ls`.) We measured the rate of these operations and found that read operations on directories are very common. On average, directories are opened for reading about 3000 times per hour. If the whole-directory cache stores the directory data in a suitable format (by storing the raw directory data, as opposed to a hash table of the entries, for instance), the whole-directory cache can provide the data for these read requests. Since an entry-based cache only holds parts of a directory, it can't satisfy directory reads.

3.5. Attribute cache results

We simulated an entry-based attribute cache, in which each client caches the attributes for a number of files. The entry-based attribute cache used the file (or directory) identifier as the key and returned the attributes (such as permissions, owner, size, and modify time). Table 4 provides a summary of the measurements of operations using and modifying attributes.

One problem with caching Unix-style attributes is that they include the access time attribute, indicating the last time the file was accessed; and the modify time attribute, indicating the last time the file was modified. For an open file, these attributes may change with each access to the file. Because of this, remote caching of Unix-style attributes will be expensive on files that are open on other workstations. Each read of the file will change the access time of the file. Each write to the file will change the access and modify times, as well as, usually, the size. To cache these attributes correctly would require the cached attributes to be updated on every read and write operation. Because of the high consistency cost this would entail, we assumed in this study that attributes of open files were not cached on remote machines. (This could be done by invalidating cached attributes for a file if the file is opened or by not guaranteeing consistency while the file is open.) We also assumed that the access time attribute was not used (that is, we didn't invalidate cached attributes each time another machine accessed the file). Another possibility for maintaining consistency of attributes, used in Andrew, is to propagate a file's attributes only when the file is closed. Under this model, consistency is loosened, since other remote machines may have the old attributes cached while the file is open and the attributes are changing. In any case, Table 4 shows that accesses to the attributes of

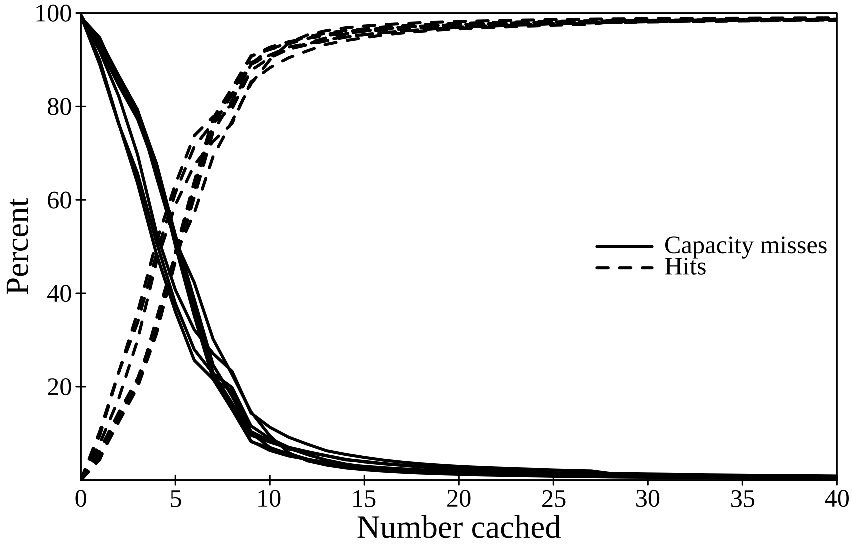

Figure 6: Whole-directory name cache performance for migrated processes. This graph shows the performance of name cache references for migrated processes (i.e. processes executed on a remote machine). This graph is analogous to Figure 2, but restricted to lookups from migrated processes. To generate this graph, the simulator used all name cache references, but only references from migrated processes are graphed. The compulsory miss rate was 0.7% ± 0.2%; the consistency miss rate was 0% to 0.2%.

---

an open file are very rare.

The results for attribute caches are generally similar to those for the name caches. Figure 5 shows the performance of the attribute caches. A cache of just 10 attributes had an average 76% hit rate, while a cache of 20 attributes raised the hit rate to 88%. The attribute cache had a relatively high compulsory miss rate of about 6%, since there are many distinct attributes used, and a miss is necessary to load each attribute into the cache. There were almost no consistency misses in the attribute caches.

It is not surprising that the attribute cache performance is similar to the name cache performance considering the close relationship between the name cache and the attribute cache: each name lookup requires an access of the corresponding attributes to check permissions, and most attribute operations require a name lookup to determine the file. However, since the attribute cache miss rates are much higher than the whole-directory name cache miss rates, the attribute cache may be the limiting factor in overall performance of the name and attribute caches.

3.6. The effects of process migration

Because we think some form of load sharing is likely to be an important part of distributed operating systems, we were concerned about the effects of process migration on client name caching. Since Table 2 shows that migrated processes accounted for an average of 19% of name lookups, these processes could have a significant effect on overall cache results. We had several reasons to suspect that name caching might perform poorly in the presence of process migration. A typical application of process migration, such as parallel compilation, involves several processes sharing a small collection of files and directories and modifying files in a shared directory. This group of migrated processes is likely to have good name and file reference locality. Migrating the processes to multiple machines eliminates the benefits of this locality. Also, since these processes may be modifying shared directories, we expected heavy consistency traffic for the shared directories.

To determine the effect of process migration on cache behavior, we made two kinds of measurements. First, we ran simulations in which we attempted to eliminate the effects of process migration. Second, we examined migrated processes separately to see if they had different characteristics from ordinary processes.

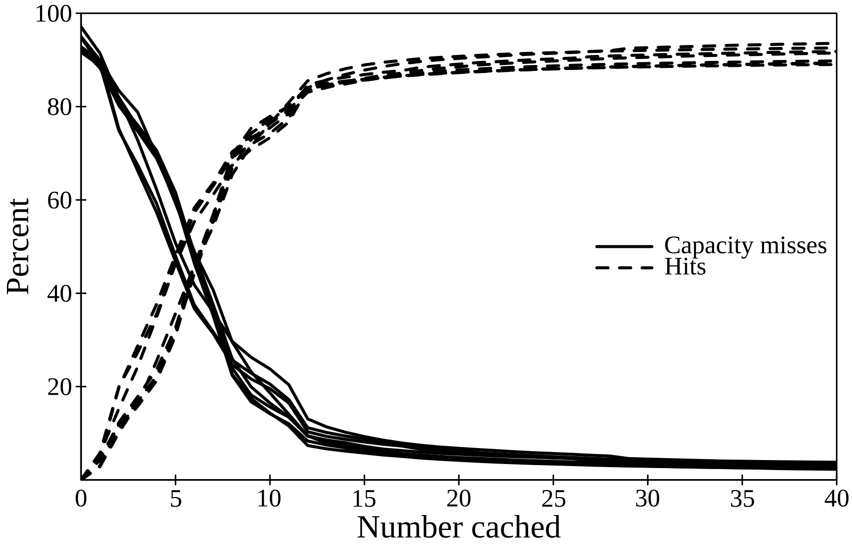

Figure 7: Attribute cache performance for migrated processes. This graph shows the performance of attribute cache references due to migrated processes. This graph is analogous to Figure 5, but restricted to migrated processes. The compulsory miss rate was 6.1 ± 2%. The consistency miss rate was under 0.08%.

We estimated the effects of eliminating process migration by treating all name and attribute requests as if they came from the home machine (the source of the migrated process) instead of the migrated machine (the machine on which the migrated processes were actually running). The results of these simulations were almost indistinguishable from the results with process migration (Figures 2, 4, and 5), so the results are not graphed. We found that if migrated processes ran on the home machine, the overall hit rate would be about 0.2% higher, and the capacity and consistency miss rate would be lower. Thus, process migration makes name cache performance worse, but the difference is very slight. Process migration had a similar effect on attribute cache performance.

For a closer look at process migration we examined migrated processes alone. Figure 6 shows the whole-directory name cache performance, considering only references from migrated processes, and Figure 7 shows the attribute cache performance. In these measurements, the name cache simulation used all the name requests to update the cache, but only the lookups from migrated processes were used to compute the hit and miss rates. Comparing these graphs to Figure 2 and Figure 5 shows that the behavior of migrated processes is very similar to that of regular processes.

3.7. The costs of cache consistency

An important aspect of distributed caching is the amount of overhead required to maintain consistency of the caches. Using a callback scheme, when a client modifies cached data, the server must call back all other clients with a cached copy so the clients can invalidate the stale data. We refer to these server callbacks as remote invalidations. Figure 8 show the average number of remote invalidations per cache access, for a whole-directory cache, entry-based directory cache, and attribute cache. Note that the invalidation rate was very low. For an average 20 entry whole-directory cache, there are about 2 invalidations of a remote machine per thousand cache accesses. For an average 40 entry entry-based directory cache, the rate is much lower: 0.3 remote invalidations per thousand accesses. This is not surprising, since individual directory entries are not modified very often, and it is even rarer for these entries to be shared by multiple machines. For attributes, an average 40 entry cache had 0.9 remote invalidations per thousand accesses. Since these invalidation rates are all very low, the cost of remote invalidations will probably not be an important consideration in cache design.

We looked at the effect of process migration on the remote invalidation rates by treating all requests as if they came from the home machine instead of the migrated machine. As expected, we found that fewer remote

Fraction of name accesses to `/tmp`	0.6% ± 0.5%
Name cache	
Hit rate	87% ± 4%
Compulsory miss rate	0.2% ± 0.1%
Capacity miss rate	3.4% ± 1.5%
Consistency miss rate	9.7% ± 3.6%
Entry-based name cache	
Hit rate	32% ± 8%
Compulsory miss rate	25% ± 7%
Invalid miss rate	24% ± 12%
Capacity miss rate	18% ± 25%
Consistency miss rate	0%
Attribute cache	
Hit rate	83% ± 4%
Compulsory miss rate	14% ± 3%
Capacity miss rate	3.4% ± 1.3%
Consistency miss rate	0%

Table 6: Statistics on `/tmp` accesses. The hit and miss rates are all given for a cache of 20 entries. The name cache results are for the `/tmp` directory. The entry-based name cache and attribute cache results are for entries in the `/tmp` directory. Measurements of `/tmp` were recorded for the first six traces; the figures presented are the average and standard deviation across these traces.

invalidations are required if we eliminate process migration in this way. We found that for a 20 element name cache, the number of remote invalidations would be about 40% lower. For a 40 element attribute cache, the number of remote invalidations would be an average of 17% lower without migration.

One other aspect of consistency overhead that we examined was how many remote machines were invalidated when a potential invalidation occurred. The results are shown in Table 5. The results show that for whole-directory and entry-based name caches, about 88% and 92% (respectively) of the time that an entry was modified, no remote machine was invalidated and only the locally cached copy was updated. For the remainder of the time, usually only one remote machine needs to be invalidated. For the attribute cache, a remote machine needed to be invalidated about 21% of the time.

These consistency overhead measurements are similar to those found on Locus by Sheltzer et al. [SLP86]. Sheltzer found that with a 60-directory cache, only 0.051% of the references required cache invalidation. This invalidation rate is about a fifth of the rate we measured on our system. The probable cause is that since we have 40 machines, compared to 15 in [SLP86], we have a higher chance of requiring invalidation. Sheltzer also found 7% of invalidations resulted in more than a single invalidation, compared to our rate of 11% of whole-directory name cache modifications affecting more than the locally cached copy.

3.8. Other results

Another concern we had with whole-directory name caching was that heavily shared and modified directories would result in a high invalidation rate. In particular, Sprite has a single `/tmp` directory shared among all machines, so we expected there would be a high rate of contention and invalidation for this directory. Table 6 shows that this is true; the whole-directory name cache has a 9.7% consistency miss rate. However, the table also shows that the fraction of accesses to `/tmp` was very low, so the contention in `/tmp` had minimal influence on overall name cache performance. Table 6 also shows that since there was essentially no sharing of files in `/tmp`, the entry-based name cache and attribute cache didn't have any consistency problems. However, these caches had a high compulsory miss rate.

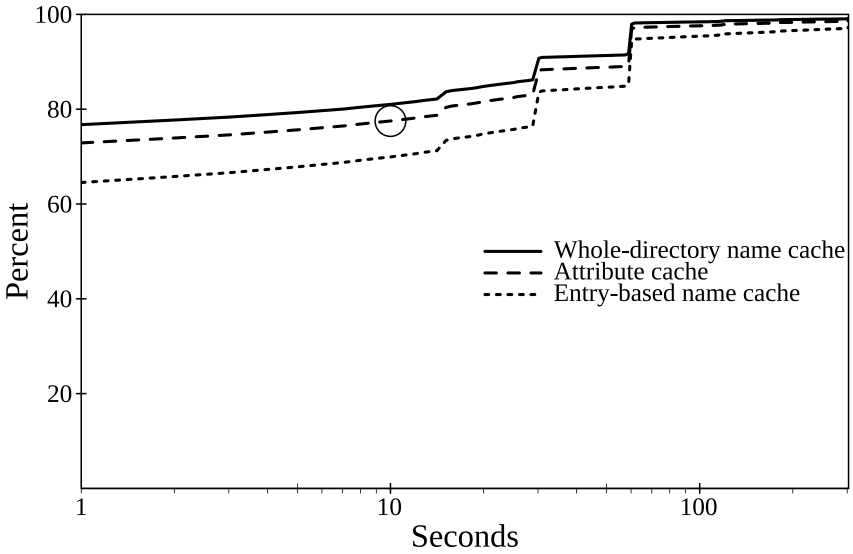

Figure 9: Idle time of cached entries when accessed. This cumulative graph shows (on a logarithmic scale) the time between accesses to cached attributes, averaged over all references that hit the cache. For instance, the circled point shows that 78% of hits in the attribute cache were to entries previously referenced less than 10 seconds ago. There are jumps at 15, 30, and 60 seconds due to programs that access files periodically. Whole-directory name cache entries had the shortest inter-reference times, followed by attributes. Entry-based name cache entries had the longest inter-reference times. The graph shows that the majority of all references to an entry were within a second of the previous reference. This graph shows the average across all eight traces. Individual traces varied about 10% from the average at the left and about 2% from the average at the right.

The entry-based name cache had very poor performance on accesses to `/tmp`; we believe that the typical use of `/tmp` accounts for this. A standard sequence of operations for using a temporary file is to generate a new filename in `/tmp`, stat the filename to ensure that it is unused, open the file, use the file, and remove the file. The stat and open will result in misses if the file does not exist. The remove will result in a hit if the filename is still in the cache after being opened. However, if there are many operations before the remove, the entry may have left the cache, resulting in a miss. The result of this sequence of events is two misses and a hit, or three misses if the remove reference results in a miss.

Since some name cache designs, such as NFS, use a timeout scheme to maintain consistency, an important question is how long should entries be kept in the cache. Figure 9 shows the inter-reference time for the whole-directory name cache, entry-based name cache, and attribute-based name cache. This graph shows the time since the last reference, for references to items in the cache. For all three cache types, the majority of references to cached entries happened no more than a second after the previous reference. This corresponds well with the single-machine results in [FlE89], which found that half of the inter-reference times were under 1/4 second. Cache entries no older than a minute accounted for over 90% of the cache hits. Note the effect of programs that run at regular intervals, such as `cron` and `xbiff`: there are jumps in the reference curves at 15, 30, and 60 seconds. Figure 9 also shows that timing out cache entries can result in a significant loss of cache performance. For instance, consider a whole-directory cache with a 95% hit rate. Figure 9 shows that 9% of the cache entries used are older than 30 seconds. Thus, invalidating entries after 30 seconds would reduce the hit rate to 86%, which almost triples the miss rate.

One final question is the effect name and attribute caching will have on the network and file server load. We did kernel name lookup timing measurements, which show that about 20% of the time the file server spent in the kernel was spent handling name lookups. (This corresponds well with the single-machine measurements in [LKM84], which found the kernel spends 19% of its time performing name lookups.) We estimate that another 20% of the time was spent handling file opens. Given a relatively small name and attribute cache on each client, we could eliminate 90% of name lookups and file opens. Combining these figures, we estimate that total file server kernel load could be reduced by 36%. Caching would also reduce network traffic. A previous study of Sprite network traffic [KhL90] found that about 1/3 of Sprite remote procedure call (RPC) packets were for `open`, `stat`, and `fstat` operations. If we assume a 90% reduction in these packets due to name and attribute caching, we conclude that caching could reduce the number of RPC network packets by 30%. (However, since most of the bytes transferred across the network result from reads and writes, the decrease in RPC network bytes from name and attribute caching is not as significant.) Based on these rough calculations, we expect name caching would result in a significant decrease in server load and network traffic. Earlier measurements of Sprite's performance in [Nel88], estimated that server utilization and network utilization in Sprite could be reduced by a factor of 2 with local name caching. However, since that estimate was an upper bound based on performance measurements on a set of benchmarks, we believe the figures here to be more realistic.

4. Conclusions

We have presented the results of simulating name and attribute caches on clients in a distributed operating system. This simulation used trace data we collected on a network of about 40 workstations running the Sprite operating system.

The simulations showed several significant results. High hit rates can be obtained even with small client caches. Very little memory is required on each client for these caches; a cache of 20 directories will likely require only 20 to 40 kilobytes per machine. We found that caching just 10 directories resulted in a name hit rate of 91%. The entry-based cache was less successful than the whole-directory cache, mainly because the entry-based cache requires more misses than the whole-directory cache in order to fill it, and also because the entry-based cache cannot handle nonexistent filenames. Attribute caching obtained a hit rate of 88% by caching 20 attributes on each client.

There are minimal problems with maintaining cache consistency. The invalidation rate for remotely cached data is very low, indicating very little network traffic is required to maintain cache consistency. The consistency miss rate, due to modifications on shared directories, is correspondingly low.

Process migration does not have a significant impact on cache behavior. We found that there was hardly any difference in performance between migrated and non-migrated processes. Sharing due to migrated processes is responsible for a large fraction of consistency invalidations.

Based on our caching data and measurements of server load, we estimate that client name and attribute caching could reduce server load by 36% and could reduce the number of remote procedure call network packets by 30%.

5. Acknowledgements

We would like to thank Mary Baker, Fred Douglis, John Hartman, Jim Mott-Smith, Mike Kupfer, and Mendel Rosenblum for their helpful comments on this paper. This research was supported by an IBM Graduate Fellowship, NASA and the Defense Advanced Research Projects Agency under Contract No. NAG2-591, and by the National Science Foundation under Grant No. CCR-8900029.

6. References

[BHK91] M. Baker, J. Hartman, M. Kupfer, K. Shirriff and J. Ousterhout, Measurements of a Distributed File System, *Proceedings of the 13th Symposium on Operating System Principles*, Oct. 1991, 198-212.

[DoO91] F. Douglis and J. Ousterhout, Transparent Process Migration: Design Alternatives and the Sprite Implementation, *Software—Practice & Experience 21*, 8 (Aug. 1991), 757-785.

[Flo86] R. Floyd, Directory Reference Patterns in a UNIX environment, Technical Report 179, Computer Science Department, The University of Rochester, Rochester, NY, Aug. 1986.

[FlE89] R. Floyd and C. Ellis, Directory Reference Patterns in Hierarchical File Systems, *IEEE Transactions on Knowledge and Data Engineering 1*, 2 (June 1989), 238-247.

[HeP90] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[Hil87] M. Hill, Aspects of Cache Memory and Instruction Buffer Performance, Report No. UCB/CSD 87/381, PhD Thesis, Computer Science Division, UC Berkeley, Berkeley, CA, Nov. 1987.

[HBM89] A. Hisgen, A. Birrell, T. Mann, M. Schroeder and G. Swart, Availability and Consistency Tradeoffs in the Echo Distributed File System, *Proceedings of the Second Workshop on Workstation Operating Systems (WWOS-II)*, Sep. 1989, 49-54.

[HKM88] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham and M. West, Scale and Performance in a Distributed File System, *ACM Transactions on Computer Systems 6*, 1 (Feb. 1988), 51-81.

[KhL90] D. Khorramabadi and C. Lowery, Analysis of Network Traffic in the Sprite Remote Procedure Call System, Computer Science 262 Project Report, Computer Science Division, UC Berkeley, Berkeley, CA, May 1990.

[KiS91] J. Kistler and M. Satyanarayanan, Disconnected Operation in the Coda File System, *Proceedings of the 13th Symposium on Operating System Principles*, Oct. 1991, 213-225.

[LKM84] S. Leffler, M. Karels and M. McKusick, Measuring and Improving the Performance of 4.2BSD, *Proceedings of the 1984 USENIX Summer Conference*, June 1984, 237-252.

[Nel88] M. Nelson, Physical Memory Management in a Network Operating System, Report No. UCB/CSD 88/471, PhD Thesis, Computer Science Division, UC Berkeley, Berkeley, CA, Nov. 1988.

[OCD88] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson and B. Welch, The Sprite Network Operating System, *IEEE Computer 21*, 2 (Feb. 1988), 23-36.

[SGK85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh and B. Lyon, Design and Implementation of the Sun Network Filesystem, *Proceedings of the 1985 USENIX Summer Conference*, June 1985, 119-130.

[SLP86] A. Sheltzer, R. Lindell and G. Popek, Name Service Locality and Cache Design in a Distributed Operating System, *Proceedings of the 6th International Conference on Distributed Computing Systems*, May 1986, 515-522.

[WPE83] B. Walker, G. Popek, R. English, C. Kline and G. Thiel, The LOCUS Distributed Operating System, *Operating Systems Review 17*, 5 (Oct. 1983), 49-70.

Ken Shirriff is a Ph.D. candidate in the Department of Electrical Engineering and Computer Sciences at the University of California at Berkeley. He is currently a member of the Sprite network operating system project. His interests include operating systems and computer architecture. He received a B.Math. degree from the University of Waterloo in 1987 and a M.S. in computer science from UC Berkeley in 1990. Ken Shirriff can be reached at `shirriff@sprite.berkeley.edu`.

John K. Ousterhout is a Professor in the Department of Electrical Engineering and Computer Sciences at the University of California at Berkeley. His interests include operating systems, distributed systems, user interfaces, and computer-aided design. He is currently leading the development of Sprite, a network operating system for high-performance workstations, and Tcl/Tk, a programming system for graphical user interfaces. In the past, he and his students developed several widely-used programs for computer-aided design, including Magic, Caesar, and Crystal. Ousterhout is a recipient of the ACM Grace Murray Hopper Award, the National Science Foundation Presidential Young Investigator Award, the National Academy of Sciences Award for Initiatives in Research, the IEEE Browder J. Thompson Award, and the UCB Distinguished Teaching Award. He received a B.S. degree in Physics from Yale University in 1975 and a Ph.D. in Computer Science from Carnegie Mellon University in 1980.

Trace	1	2	3	4	5	6	7	8
Date	1/24/91	1/25/91	5/10/91	5/11/91	5/14/91	5/15/91	6/26/91	6/27/91
Trace duration (hours)	24	23.8	24	24	24	24	24	24
Different users	44	48	47	33	48	50	46	36
Users of migration	6	6	11	8	7	11	9	9
Lookups	1235794	1466012	998926	838399	1252421	1668730	987767	2256783
Migrated lookups	131262	120022	122528	94210	28968	187599	232488	1081116
Name accesses	4295413	4914596	2638298	2198843	4406272	5753149	2750424	7549488
Modifications	56354	68547	59765	42703	65638	81159	43596	54006
Attribute accesses	5078731	5944852	3148932	2601814	5166216	6726354	3153794	8452390
Modifications	34661	73237	44956	35190	43222	54365	30574	41392

Table 2: Statistics on the trace data collected. ‘‘Different users’’ is the number of different users who used the system during the trace. ‘‘Users of migration’’ is the number of different users who used process migration during the trace. ‘‘Lookups’’ is the number of path name lookup operations during the trace. ‘‘Migrated lookups’’ is the number of these lookups that resulted from migrated processes. ‘‘Name accesses’’ is the number of name component lookups; this is larger than the number of lookups because each path name lookup may result in multiple component lookups. ‘‘Modifications’’ is the number of component lookups that resulted in modification of a name component. ‘‘Attribute accesses’’ is the number of file or directory attributes that were accessed.

Operation	Percent of total	Percent successful	Percent not found
Stat	53.5 ± 8	75 ± 9	9 ± 5
Open	42.0 ± 8	54 ± 12	32 ± 12
Unlink	3.1 ± 0.7	70 ± 3	6 ± 2
SetAttr	0.7 ± 0.8	89 ± 9	3 ± 2
Link	0.4 ± 0.2	89 ± 4	0.2 ± 0.2
Link(2)	0.3 ± 0.08	97 ± 2	0
Rmdir	0.03 ± 0.04	70 ± 30	0.3 ± 0.6
Mkdir	0.02 ± 0.01	63 ± 12	3 ± 8
Totals	100	66 ± 11	19 ± 10

Table 3: Operation breakdown. This table shows the breakdown of name lookup operations and the results of the operations. The first column of this table lists the system calls that are responsible for pathname lookups. The ‘‘Percent of total’’ column shows the breakdown of name lookup operations. The ‘‘Percent successful’’ column shows for each operation type, the percent of lookups that completed successfully. The ‘‘Percent not found’’ column shows for each operation the percentage of lookups that failed because one of the path name components did not exist. (There are other sources of failure, such as lack of proper permissions, which are not shown.) The numbers are given as an average over the eight traces, followed by the standard deviation. The ‘‘SetAttr’’ row includes functions such as `chmod`, `utimes`, and `chown`, which change a file’s attributes. The ‘‘Link’’ and ‘‘Link(2)’’ rows count the two separate pathname lookups required for the hard link operation. The ‘‘Total’’ row shows the percent of all operations that completed successfully and the percent that failed because of a missing component.

Category	Average Number
Stat	710000
SetAttr	9900
Fstat	79000
FsetAttr	1700
Permission accesses	4300000
Stats of read-open files	1500
Stats of write-open files	2700

Table 4: Operations affecting attributes. This table gives the number of operations per trace, averaged over the eight traces. ‘‘Stat’’ counts the number of `stat` operations, which obtain file attributes from a pathname. The label ‘‘SetAttr’’ combines operations such as `chown` and `chmod` that set file attributes given a path. ‘‘Fstat’’ counts `fstat` operations, which obtain file attributes using a token for an open file rather than a textual path name. ‘‘FsetAttr’’ combines operations such as `fchown` and `fchmod` that set file attributes of an open file. ‘‘Permission accesses’’ is the number of path name component lookups that required examining a file or directory’s permission attributes. ‘‘Stats of read-open files’’ indicates the number of attribute accesses that were performed on a file while some process had the file open for reading. ‘‘Stats of write-open files’’ indicates the number of attribute accesses that were performed on a file while it was open for writing.

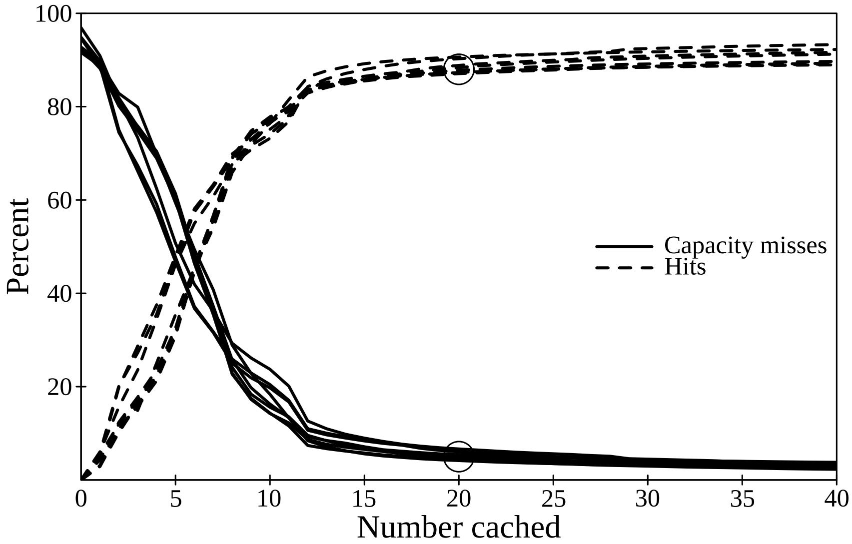

Figure 5: Attribute cache performance. This shows the hit rate for accesses to the attribute cache. The X axis shows the size of the cache in entries. The Y axis shows the percent of accesses that resulted in cache hits or capacity misses. For instance, the circled points show that with 20 cached attributes, the hit rate was about 88% and the capacity miss rate was about 5%. Accesses are divided up as hits, capacity misses, consistency misses, and compulsory misses. The compulsory miss rate was 6.2% ± 1.8%. Consistency misses were under 0.07%.

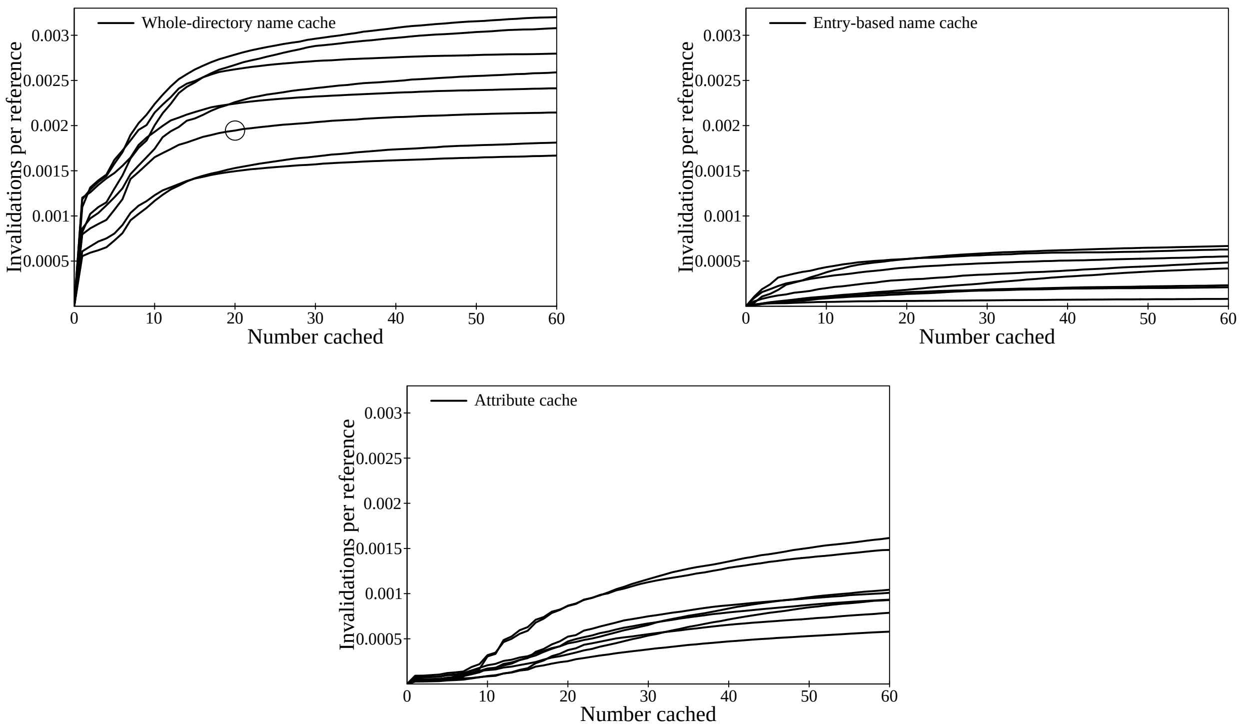

Figure 8: Number of remote invalidations per reference vs. cache size. These graphs show how cache size influences the number of invalidations required. The upper left graph shows results for the whole-directory name cache, the upper right graph shows the entry-based name cache, and the lower graph shows the attribute cache. These graphs have a separate line for each trace. For instance, the circled point shows that in one trace, a 20 element name cache required 2 invalidations of remotely cached directories per thousand cache references.

Cache type	Number of invalidations 0	1	≥2
Whole-directory name cache	88% ± 5%	9% ± 3%	2% ± 2%
Entry-based name cache	92% ± 6%	7% ± 5%	0.5% ± .4%
Attribute cache	84% ± 3%	16% ± 3%	0.4% ± 0.1%

Table 5: Number of invalidations. This graph shows how many remote machines were invalidated when names or attributes were modified. This table shows that usually only the local copy was updated. For a minority of modifications, a remotely cached copy had to be invalidated. It was rare for a modification to require invalidation of more than one remotely cached copy. Averages and standard deviations are over the eight traces.