Caching in the Sprite Network File System

Michael N. Nelson Brent B. Welch John K. Ousterhout

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

Abstract

The Sprite network operating system uses large main-memory disk block caches to achieve high performance in its file system. It provides non-write-through file caching on both client and server machines. A simple cache consistency mechanism permits files to be shared by multiple clients without danger of stale data. In order to allow the file cache to occupy as much memory as possible, the file system of each machine negotiates with the virtual memory system over physical memory usage and changes the size of the file cache dynamically. Benchmark programs indicate that client caches allow diskless Sprite workstations to perform within 0-12% of workstations with disks. In addition, client caching reduces server loading by 50% and network traffic by 75%.

1. Introduction

Caches have been used in many operating systems to improve file system performance. Typically, caching is implemented by retaining in main memory a few of the most recently accessed disk blocks (e.g., UNIX [THOM78]). Repeated accesses to a block in the cache can be handled without involving the disk, which has two advantages. First, caching reduces delays: a block in the cache can usually be returned to a waiting process five to ten times more quickly than one that must be fetched from disk. Second, caching reduces contention for the disk arm, which may be advantageous if several processes are attempting to access files on the same disk. Measurements of timesharing systems indicate that even small caches provide substantial benefits, and that the benefits are increasing as larger physical memories permit larger caches [LEFF84, OUST85].

This paper describes a simple distributed mechanism for caching files among a networked collection of workstations. We have implemented it as part of the Sprite

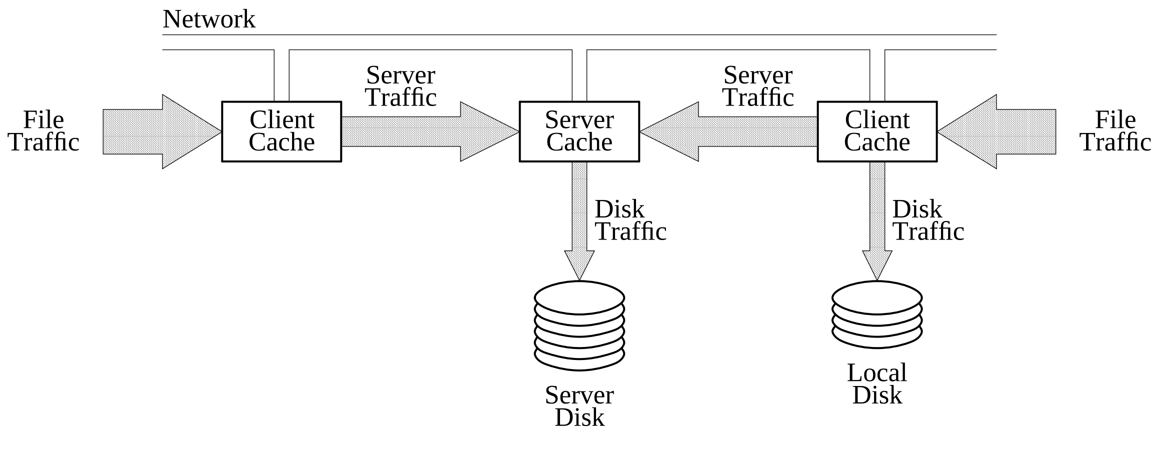

Figure 1. File caches in the Sprite system. When a process makes a file access, it is presented first to the cache of the process's workstation (''file traffic''). If not satisfied there, the request is passed either to the local disk, if the file is stored there (''disk traffic''), or to the server where the file is stored (''server traffic''). Servers also maintain caches in order to reduce their disk traffic.

operating system. In Sprite, file information is cached in the main memories of both servers (workstations with disks), and clients (workstations wishing to access files on non-local disks), as shown in Figure 1. On machines with disks, the caches achieve the same effects described above, namely to reduce disk-related delays and contention. On clients, the caches also reduce the communication delays that would otherwise be required to fetch blocks from servers. In addition, client caches reduce contention for the network and for the server machines. Since server CPUs appear to be the bottleneck in several existing network file systems [SATY85, LAZO86], client caching offers the possibility of greater system scalability as well as increased performance.

There are two unusual aspects to the Sprite caching mechanism. The first is that Sprite guarantees workstations a consistent view of the data in the file system, even when multiple workstations access the same file simultaneously and the file is cached in several places at once. This is done through a simple cache consistency mechanism that flushes portions of caches and disables caching for files undergoing read-write sharing. The result is that file access under Sprite has exactly the same semantics as if all of the processes on all of the workstations were executing on a single timesharing system.

The second unusual feature of the Sprite caches is that they vary in size dynamically. This was a consequence of our desire to provide very large client caches, perhaps occupying most of the clients' memories. Unfortunately, large caches may occasionally conflict with the needs of the virtual memory system, which would like to use as much memory as possible to run user processes. Sprite provides a simple mechanism through which the virtual memory system and file system of each workstation negotiate over the machine's physical memory. As the relative needs of the two systems change, the file cache changes in size.

We used a collection of benchmark programs to measure the performance of the Sprite file system. On average, client caching resulted in a speedup of about 10-40% for programs running on diskless workstations, relative to diskless workstations without client caches. With client caching enabled, diskless workstations completed the benchmarks only 0-12% more slowly than workstations with disks. Client caches reduced the server utilization from about 5-18% per active client to only about 1-9% per active client. Since normal users are rarely active, our measurements suggest that a single server should be able to support at least 50 clients. In comparisons with Sun's Network File System [SAND85] and the Andrew file system [SATY85], Sprite completed a file-intensive benchmark 30-35% faster than the other systems. Sprites server utilization was substantially less than NFS but more than Andrew.

The rest of the paper is organized as follows: Section 2 gives a brief overview of Sprite; Section 3 describes prior work that motivated our cache design; Section 4 presents the basic structure of the Sprite caches; Section 5 describes the consistency protocols and Section 6 discusses the mechanism for varying the cache sizes; Section 7 presents the benchmark results; Section 8 compares Sprite to other network file systems; and Section 9 describes work still to be done in the areas of recovery and allocation.

2. Overview of Sprite

Sprite is a new operating system being implemented at the University of California at Berkeley as part of the development of SPUR, a high-performance multiprocessor workstation [HILL86]. A preliminary version of Sprite is currently running on Sun-2 and Sun-3 workstations, which have about 1-2 MIPS processing power and 4-16 Mbytes of main memory. The system is targeted for workstations like these and newer models likely to become available in the near future, such as SPURs; we expect the future machines to have at least five to ten times the processing power and main memory of our current machines, as well as small degrees of multiprocessing. We hope that Sprite will be suitable for networks of up to a few hundred of these workstations. Because of economic and environmental factors, most workstations will not have local disks; instead, large fast disks will be concentrated on a few server machines.

The interface that Sprite provides to user processes is much like that provided by UNIX [RITC74]. The file system appears as a single shared hierarchy accessible equally by processes on any workstation in the network (see [WELCH86a] for information on how the name space is managed). The user interface to the file system is through UNIX-like system calls such as open, close, read, and write.

Although Sprite appears similar in function to UNIX, we have completely re-implemented the kernel in order to provide better network integration. In particular, Sprite's implementation is based around a simple kernel-to-kernel remote-procedure-call (RPC) facility [WELCH86b], which allows kernels on different workstations to request services of each other using a protocol similar to the one described by Birrell and Nelson [BIRR84]. The Sprite file system uses the RPC mechanism extensively for cache management.

3. Background Work

The main motivation for the Sprite cache design came from a trace-driven analysis of file activity in several time-shared UNIX 4.2 BSD systems, hereinafter referred to as ''the BSD study'' [OUST85]. For those systems the BSD study showed that even small file caches are effective in reducing disk traffic, and that large caches (4-16 megabytes) work even better, cutting disk traffic by as much as 90%. For the kinds of applications measured in the BSD study it appears that increases in memory sizes will soon make it possible to keep entire file working sets in main memory, with disks serving primarily as backup devices. Although the BSD study was based on time-sharing machines rather than networks of personal workstations, we hypothesized that the results would apply in a network environment too, and that the overheads associated with remote file access could be reduced by caching on clients as well as servers (Sections 5.3 and 7 provide simulation and measurement data to support this hypothesis).

An additional motivating factor for us was a concern about server contention. A study of remote file access by Lazowska et al. concluded that the server CPU is the primary bottleneck that limits system scalability [LAZO86]. Independently, the designers of the Andrew file system decided to redesign their system in order to offload the servers [SATY85], and achieved substantial improvements as a result [HOWA87]. These experiences, plus our own informal observations of our computing environment, convinced us that client caching could substantially increase the scalability of the system.

4. Basic Cache Design

This section describes the basic organization of file caches in Sprite, and addresses three issues:

- Where should client caches be kept: main memory or local disk?
- How should caches be structured and addressed?
- What policy should be used for writing blocks back to disk?

The issues of maintaining cache consistency and varying the sizes of caches are discussed separately in the following two sections.

4.1. Caches on Disk or in Main Memory?

In several previous network file systems (e.g. Andrew [SATY85, HOWA87] and Cedar [SCHR85]), clients' file caches were kept on their local disks. For Sprite we chose to cache file data in main memory, for four reasons. First, main-memory caches permit workstations to be diskless, which makes them cheaper and quieter. Second, data can be accessed more quickly from a cache in main memory than a cache on disk. Third, physical memories on client workstations are now large enough to provide high hit ratios (e.g. a 1-Mbyte client cache provides greater than 80% read hits). As memories get larger, main-memory caches will grow to achieve even higher hit ratios. Fourth, the server caches will be in main memory regardless of where client caches are located: by using main-memory caches on clients too, we were able to build a single caching mechanism for use by both servers and clients.

4.2. Cache Structure

The Sprite caches are organized on a block basis using a fixed block size of 4 Kbytes. We made this choice largely for simplicity and are prepared to revise it after we have more experience with the system. Cache blocks are addressed virtually, using a unique file identifier provided by the server and a block number within the file. We used virtual addresses instead of physical disk addresses so that clients could create new blocks in their caches without first contacting a server to find out their physical locations. Virtual addressing also allows blocks in the cache to be located without traversing the file's disk map.

For files accessed remotely, client caches hold only data blocks. Servers also cache file maps and other disk management information. These blocks are addressed in the cache using the blocks' physical disk addresses along with a special ''file identifier'' corresponding to the physical device.

4.3. Writing Policy

The policy used to write dirty blocks back to the server or disk has a critical effect on the system's performance and reliability. The simplest policy is to write data through to disk as soon as it is placed in any cache. The advantage of write-through is its reliability: little information is lost when a client or server crashes. However, this policy requires each write access to wait until the information is written to disk, which results in poor write performance. Since about one-third of all file accesses are writes [OUST85], a caching scheme based on write-through cannot reduce disk or server traffic by more than two-thirds.

An alternate write policy is to delay write-backs: blocks are initially written only to the cache and then written through to the disk or server some time later. This policy has two advantages over write-through. First, since writes are to the cache, write accesses complete much more quickly. Second, data may be deleted before it is written back, in which case it need not be written at all. In the BSD study, 20 to 30 percent of new data was deleted within 30 seconds, and 50 percent was deleted within 5 minutes. Thus, a policy that delays writes several minutes can substantially reduce the traffic to the server or disk. Unfortunately, delayed-write schemes introduce reliability problems, since unwritten data will be lost whenever a server or client crashes.

For Sprite, we chose a delayed-write policy similar to the one used in UNIX: every 30 seconds, all dirty blocks that haven't been modified in the last 30 seconds are written back. A block written on a client will be written to the server's cache in 30-60 seconds, and will be written to disk in 30-60 more seconds. This policy avoids delays when writing files and permits modest reductions in disk/server traffic, while limiting the damage that can occur in a crash. We plan to experiment with longer write-back intervals in the future.

Another alternative is to write data back to the server when the file is closed. This approach is used in the Andrew system and NFS. Unfortunately, the BSD study found that 75 percent of files are open less than 0.5 seconds and 90 percent are open less than 10 seconds. This indicates that a write-on-close policy will not significantly reduce disk or server traffic. In addition, the write-on-close policy requires the closing process to delay while the file is written through, which reduces the performance advantages of delayed writes.

5. Cache Consistency

Allowing clients to cache files introduces a consistency problem: what happens if a client modifies a file that is also cached by other clients? Can subsequent references to the file by other clients return ''stale'' data? Most existing network file systems provide only limited guarantees about consistency. For example, the NFS and Andrew systems guarantee that once a file is closed all data is back on the server and future opens by other clients will cause their caches to be updated with the new version. Under conditions of ''sequential write-sharing'' (a file is shared but is never open simultaneously for reading and writing on different clients), each client will always see the most up-to-date version of the file. However, if a file in NFS or Andrew is open simultaneously on several clients and one of them modifies it, the other clients will not see the changes immediately; users are warned not to attempt this kind of sharing (which we call ‘‘concurrent write-sharing’’).

For Sprite we decided to permit both concurrent and sequential write-sharing. Sprite guarantees that whenever a process reads data from a file, it receives the most recently written data, regardless of when and where the data was last written. We did this in order to make the user view of the file system as clean and simple as possible, and to encourage use of the file system as a shared system-wide store for exchanging information between different processes on different machines. We hope that shared files will be used to simplify the implementation of system services such as print spoolers and mailers. Of course, we still expect that concurrent write-sharing will be infrequent, so the consistency algorithm is optimized for the case where there is no sharing.

It is important to distinguish between consistency and correct synchronization. Sprite’s mechanism provides consistency: each read returns the most up-to-date data. However, the cache consistency mechanism cannot guarantee that concurrent applications perform their reads and writes in a sensible order. If the order matters, applications must synchronize their actions on the file using system calls for file locking or other available communication mechanisms. Cache consistency simply eliminates the network issues and reduces the problem to what it was on time-sharing systems.

Sprite uses the file servers as centralized control points for cache consistency. Each server guarantees cache consistency for all the files on its disks, and clients deal only with the server for a file: there are no direct client-client interactions. The Sprite algorithm depends on the fact that the server is notified whenever one of its files is opened or closed, so it can detect when concurrent write-sharing is about to occur. This approach prohibits performance optimizations (such as name caching) that allow clients to open files without contacting the files' servers. Section 8 discusses the potential performance improvements that name caching could provide.

5.1. Concurrent Write-Sharing

Concurrent write-sharing occurs for a file when it is open on multiple clients and at least one of them has it open for writing. Sprite deals with this situation by disabling client caching for the file, so that all reads and writes for the file go through to the server. When a server detects (during an ''open'' operation) that concurrent write-sharing is about to occur for a file, it takes two actions. First, it notifies the client that has the file open for writing, if any, telling it to write all dirty blocks back to the server. There can be at most one such client. Second, the server notifies all clients that have the file open, telling them that the file is no longer cacheable. This causes the clients to remove all of the file's blocks from their caches. Once these two actions are taken, clients will send all future accesses for that file to the server. The server's kernel serializes the accesses to its cache, producing a result identical to running all the client processes on a single timeshared machine.

Caching is disabled on a file-by-file basis, and only when concurrent write-sharing occurs. A file can be cached simultaneously by many clients as long as none of them is writing the file, and a writing client can cache the file as long as there are no concurrent readers or writers on other workstations. When a file becomes non-cacheable, only those clients with the file open are notified; if other clients have some of the file's data in their caches, they will take consistency actions the next time they open the file, as described below. A non-cacheable file becomes cacheable again once it is no longer undergoing concurrent write sharing; for simplicity, however, Sprite does not not re-enable caching for files that are already open.

5.2. Sequential Write-Sharing

Sequential write-sharing occurs when a file is modified by one client, closed, then opened by some other client. There are two potential problems associated with sequential write-sharing. First, when a client opens a file it may have out-of-date blocks in its cache. To solve this problem, servers keep a version number for each file, which is incremented each time the file is opened for writing. Each client keeps the version numbers of all the files in its cache. When a file is opened, the client compares the server's version number for the file with its own. If they differ, the client flushes the file from its cache. This approach is similar to NFS and the early versions of Andrew.

Server Traffic With Cache Consistency				
Client Cache Size	Blocks Read	Blocks Written	Total	Traffic Ratio
0 Mbyte	445815	172546	618361	100%
0.5 Mbyte	102469	96866	199335	32%
1 Mbyte	84017	96796	180813	29%
2 Mbytes	77445	96796	174241	28%
4 Mbytes	75322	96796	172118	28%
8 Mbytes	75088	96796	171884	28%

Table 1. Client caching simulation results, based on trace data from BSD study. Each user was treated as a different client, with client caching and a 30-second delayed-write policy. The table shows the number of read and write requests made by client caches to the server, for different client cache sizes. The ''Traffic Ratio'' column gives the total server traffic as a percentage of the total file traffic presented to the client caches. Write-sharing is infrequent: of the write traffic, 4041 blocks were written through because of concurrent write-sharing and 6887 blocks were flushed back because of sequential write-sharing.

The second potential problem with sequential write-sharing is that the current data for the file may be in some other client's cache (the last writer need not have flushed dirty blocks back to the server when it closed the file). Servers handle this situation by keeping track of the last writer for each file; this client is the only one that could potentially have dirty blocks in its cache. When a client opens a file the server notifies the last writer (if there is one and if it is a different client than the opening client), and waits for it to write its dirty blocks through to the server. This ensures that the reading client will receive up-to-date information when it requests blocks from the server.

5.3. Simulation Results

We used the trace data from the BSD study to estimate the overheads associated with cache consistency, and also to estimate the overall effectiveness of client caches. The traces were collected over 3-day mid-week intervals on 3 VAX-11/780s running 4.2 BSD UNIX for program development, text processing, and computer-aided design applications; see [OUST85] for more details. The data were used as input to a simulator that treated each timesharing user as a separate client workstation in a network with a single file server. The results are shown in Table 1. Client caching reduced server traffic by

Server Traffic, Ignoring Cache Consistency				
Client Cache Size	Blocks Read	Blocks Written	Total	Traffic Ratio
0 Mbyte	445815	172546	618361	100%
0.5 Mbyte	80754	93663	174417	28%
1 Mbyte	52377	93258	145635	24%
2 Mbytes	41767	93258	135025	22%
4 Mbytes	38165	93258	131423	21%
8 Mbytes	37007	93258	130265	21%

Table 2. Traffic without cache consistency. Similar to Table 1 except that cache consistency issues were ignored completely.

over 70% and resulted in read hit ratios of more than 80%.

Table 2 presents similar data for a simulation where no attempt was made to guarantee cache consistency. A comparison of the bottom-right entries in Tables 1 and 2 shows that about one-fourth of all server traffic in Table 1 is due to cache consistency. Table 2 is not realistic, in the sense that it simulates a situation where incorrect results would have been produced; nontheless, it provides an upper bound on the improvements that might be possible with a more clever cache consistency mechanism.

6. Virtual Memory and the File System

In addition to guaranteeing coherency between the client caches, we wanted to permit each client cache to be as large as possible. For example, applications that don't require much virtual memory should be able to use most of the physical memory as a file cache. However, if the caches were fixed in size (as they are in UNIX), then large caches would leave little physical memory for running user programs, and it would be difficult to run applications with large virtual memory needs. In order to get the best overall performance, Sprite allows each file cache to grow and shrink dynamically in response to changing demands on the machine's virtual memory system and file system. This is accomplished by having the two modules negotiate over physical memory usage.

The file system module and the virtual memory module each manage a separate pool of physical memory pages. Virtual memory keeps its pages in approximate LRU order through a version of the clock algorithm [NELS86]. The file system keeps its cache blocks in perfect LRU order since all block accesses are through the ''read'' and ''write'' system calls. Each system keeps a time-of-last-access for each page or block. Whenever either module needs additional memory (because of a page fault or a miss in the file cache), it compares the age of its oldest page with the age of the oldest page from the other module. If the other module has the oldest page, then it is forced to give up that page; otherwise the module recycles its own oldest page.

The approach just described has two potential problems: double-caching and multi-block pages. Double-caching can occur because virtual memory is a user of the file system: backing storage is implemented using ordinary files, and read-only code is demand-loaded directly from executable files. A naive implementation might cause pages being read from backing files to end up in both the file cache and the virtual-memory page pool; pages being eliminated from the virtual-memory page pool might simply get moved to the file cache, where they would have to age for another 30 seconds before being sent to the server. To avoid these inefficiencies, the virtual memory system bypasses the local file cache when reading and writing backing files. A similar problem occurs when demand-loading code from its executable file. In this case, the pages may already be in the file cache (e.g. because the program was just recompiled). If so, the page is copied to the virtual memory page pool and the block in the file cache is given an ‘‘infinite’’ age so that it will be replaced before anything else in memory.

Although virtual memory bypasses its local file cache when reading and writing backing files, the backing files *will* be cached on servers. This makes servers’ memories into an extended main memory for their clients.

The second problem with the negotiation between virtual memory and the file system occurs when virtual memory pages are large enough to hold several file blocks. Is the LRU time of a page in the file cache the age of the oldest block in the page, the age of the youngest block in the page, or the average age of the blocks in the page? Once it is determined which page to give back to virtual memory, what should be done with the other blocks in the page if they have been recently accessed? For our Sun-3 implementation of Sprite, which has 8-Kbyte pages but 4-Kbyte file blocks, we used a simple solution: the age of a page is the age of the youngest block in the page, and when a page is relinquished all blocks in the page are removed. We are currently investigating the effect of this policy on file system performance.

We also considered more centralized approaches to trading off physical memory between the virtual memory page pool and the file cache. One possibility would be to access all information through the virtual memory system. To access a file, it would first be mapped into a process's virtual address space and then read and written just like virtual memory, as in Apollo's DOMAIN system [LEACH83]. This approach would eliminate the file cache entirely; the standard page replacement mechanisms would automatically balance physical memory usage between file and program information. We rejected this approach for several reasons, the most important of which is that it would have forced us to use a more complicated cache consistency scheme for concurrent write-sharing. A mapped-file approach requires a file's pages to be cached in a workstation's memory before they can be accessed, so we would not have been able to implement cache consistency by refusing to cache shared files.

Another possible approach would have been to implement a centralized physical memory manager, from which both the virtual memory system and the file system would make page requests. The centralized manager would compute page ages and make all replacement decisions. We rejected this approach because the most logical way to compute page ages is different for virtual memory than for files. The only thing the two modules have in common is the notion of page age and LRU replacement. These shared notions are retained in our distributed mechanism, while leaving each module free to age its own pages in the most convenient way. Our approach also permits us to adjust the relative aging rates for virtual memory and file pages, if that should become desirable. Our initial experiences with the system suggest that virtual memory pages should receive preferential treatment, particularly in times of memory contention: the sequential nature of file accesses means that a low file hit ratio has a much smaller impact on system performance than a low virtual-memory hit ratio.

7. Benchmarks

This section describes a series of benchmarks we ran to measure the performance of the Sprite file system. Our goal was to measure the benefits provided by client caches in reducing delays and contention:

- How much more quickly can file-intensive programs execute with client caches than without?
- How much do client caches reduce the load placed on server CPUs?
- How much do client caches reduce the network load?
- How many clients can one server or network support?
- How will the benefits of client caches change as CPU speeds and memory size increase?

All of our measurements were made on configurations of Sun-3 workstations (about 2 MIPS processing power). Clients were Sun-3/75's and Sun-3/160's with at least 8 Mbytes of memory, and the server was a Sun-3/180 with 12 Mbytes of memory and a

File Lookup Operations			
Operation	Local Disk	Diskless	
		Elapsed Time	Server CPU Time
Open/Close	2.17ms	8.11ms	3.75ms
Failed Open	1.48ms	4.13ms	2.01ms
Get Attributes	1.28ms	4.47ms	2.12ms

Table 3. Cost of three common file name lookup operations in Sprite. Each of these operations requires contacting the server of the named file. Times are milliseconds per operation on a pathname with a single component. The first row is the cost of opening and closing a file, the second row is the cost of opening a file that does not exist, and the third row is the cost of getting the attributes of a file (‘‘stat’’).

400-Mbyte Fujitsu Eagle disk.

7.1. Micro-benchmarks

We wrote several simple benchmarks to measure the low-level performance of the Sprite file system. The first set of benchmarks measured the time required for local and remote invocation of three common naming operations. See Table 3. The remote versions took 3-4 times as long as the local versions; about half of the elapsed time for the remote operations was spent executing in the server’s CPU. The second set of benchmarks measured the raw read and write performance of the Sprite file system by reading or writing a single large file sequentially. Before running the programs, we rigged the system so that all the accesses would be satisfied in a particular place (e.g. the client’s cache). Table 4 shows the I/O speeds achieved to and from caches and disks in different

Read & Write Throughput, Kbytes/second				
	Local Cache	Server Cache	Local Disk	Server Disk
Read	3269	475	224	212
Write	2893	380	197	176

Table 4. Maximum rates at which programs can read and write file data in various places, using large files accessed sequentially.

locations.

Table 4 contains two important results. First, a client can access bytes in its own cache 6-8 times faster than those in the server's cache. This means that, in the best case, client caching could permit an application program to run as much as 6-8 times faster than it could without client caching. The second important result is that a client can read and write the server's cache at about the same speed as a local disk. In our current implementation the server cache is twice as fast as a local disk, but this is because Sprite's disk layout policy only allows one block to be read or written per disk revolution. We expect eventually to achieve throughput to local disk at least as good as 4.3BSD, or about 2-3 times the rates listed in Table 4; under these conditions the local disk will have about the same throughput as the server's cache. In the future, as CPUs get much faster but disks don't, the server's cache should become much faster than a local disk, up to the limits of network bandwidth. Even for paging traffic, this suggests that a large server cache may provide better performance than a local disk.

7.2. Macro-benchmarks

The micro-benchmarks give an upper limit on the costs of remote file access and the possible benefits of client caching. To see how much these costs and benefits affect real applications, we ported several well-known programs from UNIX to Sprite and measured them under varying conditions. We ran each benchmark three times for each data point measured and took the average of the three runs. Table 5 describes the benchmark programs.

Program	Description	I/O (Kbytes/sec)	
		Read	Write
Andrew	Copy a directory hierarchy containing 70 files and 200 Kbytes of data; examine the status of every file in the new subtree; read every byte of the files; compile and link the files. Developed by M. Satyanarayanan for benchmarking the Andrew file system; see [HOWA87] for details.	54.9	34.4
Fs-make	Use the ‘‘make’’ program to recompile the Sprite file system: 33 source files, 33,800 lines of C source code.	56.6	28.9
Simulator	Simulate set-associative cache memory using 3375-Kbyte address trace.	23.0	0.0
Sort	Sort a 1-Mbyte file.	47.0	90.2
Diff	Compare 2 identical 1-Mbyte files.	252.4	0
Nroff	Format the text of this paper.	16.1	18.1

Table 5. Macro-benchmarks. The I/O columns give the average rates at which file data were read and written by the benchmark when run on Sun-3’s with local disks and warm caches; they measure the benchmark’s I/O intensity.

Benchmark	Local Disk, with Cache		Diskless, Server Cache Only		Diskless, Client & Server Caches	
	Cold	Warm	Cold	Warm	Cold	Warm
Andrew	261 105%	249 100%	373 150%	363 146%	291 117%	280 112%
Fs-make	660 102%	649 100%	855 132%	843 130%	698 108%	685 106%
Simulator	161 109%	147 100%	168 114%	153 104%	167 114%	147 100%
Sort	65 107%	61 100%	74 121%	72 118%	66 108%	61 100%
Diff	22 165%	8 100%	27 225%	12 147%	27 223%	8 100%
Nroff	53 103%	51 100%	57 112%	56 109%	53 105%	52 102%

Table 6. Execution times with and without local disks and caching, measured on Sun-3's. The top number for each run is total elapsed time in seconds. The bottom number is normalized relative to the warm-start time with a local disk. ''Cold'' means that all caches, both on server and client, were empty at the beginning of the run. ''Warm'' means that the program was run once to load the caches, then timed on a second run. In the ''Diskless, Server Cache Only'' case, the client cache was disabled but the server cache was still enabled. In all other cases, caches were enabled on all machines. All caches were allowed to vary in size using the VM-FS negotiation scheme.

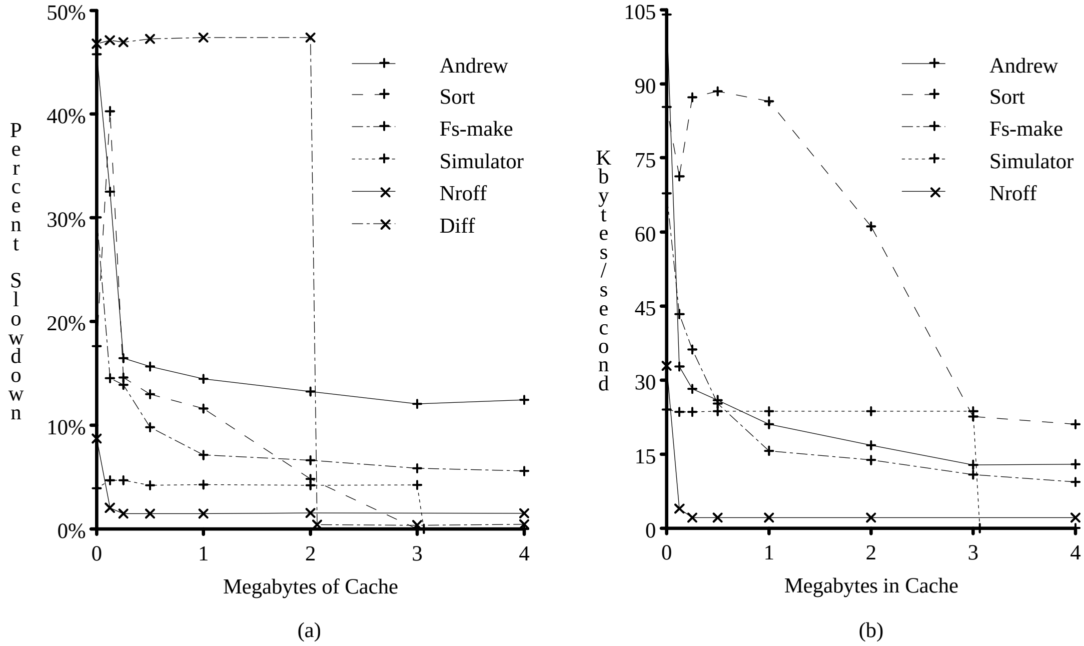

Figure 2. Client degradation and network traffic (diskless Sun-3's with client caches, warm start) as a function of maximum client cache size. For each point, the maximum size of the client cache was limited to a particular value. The ''degradation'' shown in (a) is relative to the time required to execute the benchmark with a local disk and warm cache. The network traffic shown in (b) includes bytes transmitted in packet headers and control packets, as well as file data. The diff benchmark did not fit on graph (b); for all cache sizes less than 2 Mbytes it has an I/O rate of 185 Kbytes/second and for all larger cache sizes it has an I/O rate of only 0.5 Kbytes/second.

7.2.1. Application Speedups

Table 6 lists the total elapsed time to execute each of the macro-benchmarks with local or remote disks and with client caches enabled or disabled. Without client caching, diskless machines were generally about 10-50% slower than those with disks. With client caching enabled and a warm start (caches already loaded by a previous run of the program), the difference between diskless machines and those with disks was very small; in the worst case, it was only about 12%. Figure 2(a) shows how the performance varied with the size of the client cache.

We expect the advantages of client caching to improve over time, for two reasons. First, increasing memory sizes will make larger and larger caches feasible, which will increase their effectiveness. Second, processor speeds are increasing faster than network or disk speeds; without caches, workstations will end up spending more and more of their time waiting for the network or disk.

7.2.2. Server Load

One of the most beneficial effects of client caching is its reduction in the load placed on server CPUs. Figure 3 shows the server CPU utilization with and without client caching. In general, a diskless client without a client cache utilized about 5-20% of the server's CPU. With client caching, the server utilization dropped by a factor of two or more, to 2-9%.

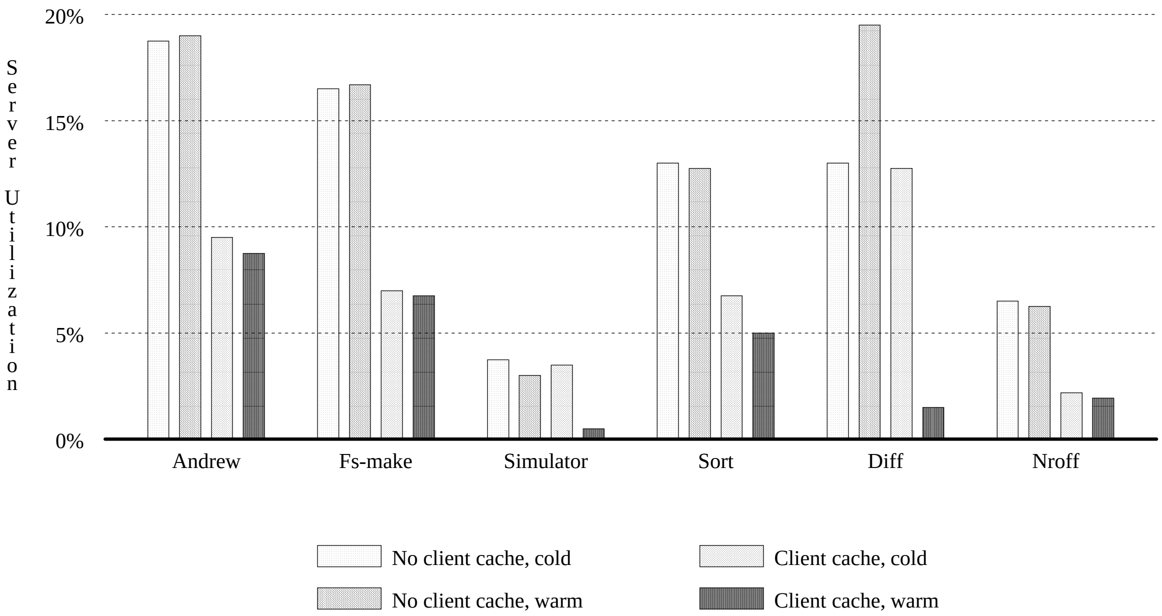

Figure 3. Client caching reduces server loading by a factor of 2-5 (measured on Sun-3's with variable-size client caches).

7.2.3. Network Utilization

In their analysis of diskless file access, based on Sun-2 workstations, Lazowska *et al.* concluded that network loading was not yet a major factor in network file systems [LAZO86]. However, as CPU speeds increase the network bandwidth is becoming more and more of an issue. Figure 2(b) plots network traffic as a function of cache size for our benchmarks running on Sun-3's. Without client caching the benchmarks averaged 6.5% utilization of the 10-Mbit/second Ethernet. The most intensive application, diff, used 15% of the network bandwidth for a single client. Machines at least five times faster than Sun-3's will be widely available within a few years (e.g., the SPUR workstations under development at Berkeley, or the recently-announced Sun-4); a single one of these machines would utilize 30-50% of the Ethernet bandwidth running the benchmarks without client caching. Without client caches, application performance may become limited by network transmission delays, and the number of workstations on a single Ethernet may be limited by the bandwidth available on the network.

Fortunately, Figure 2(b) shows that client caching reduces network utilization by about a factor of four, to an average of about 1.5% for the benchmarks. This suggests that 10-Mbit Ethernets will be adequate for the current generation of CPUs and perhaps one or two more generations to follow. After that, higher-performance networks will become essential.

7.2.4. Contention

In order to measure the effects of loading on the performance of the Sprite file system, we ran several versions of the most server-intensive benchmark, Andrew, simultaneously on different clients. Each client used a different copy of the input and output files,

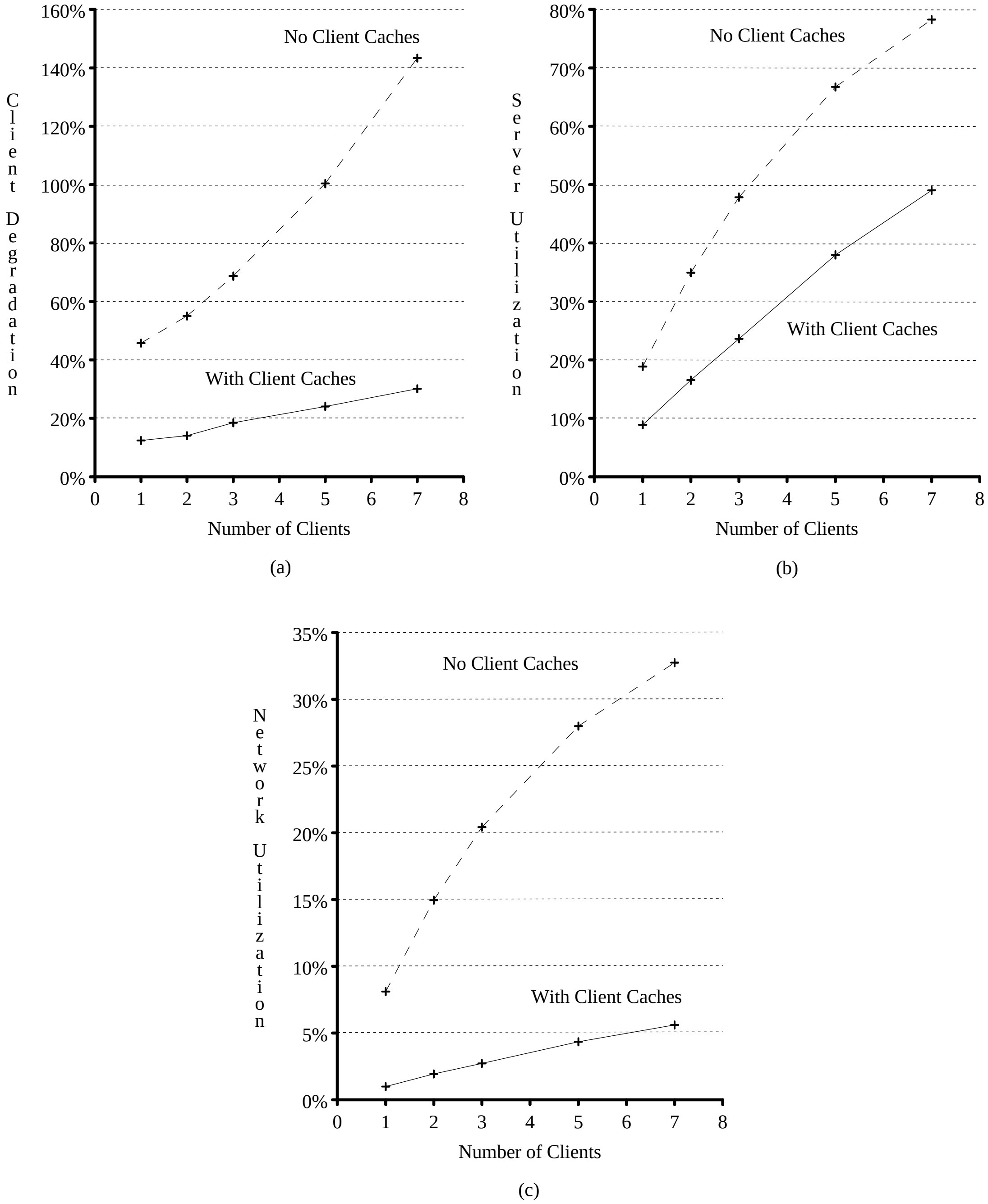

Figure 4. Effect of multiple diskless clients running the Andrew benchmark simultaneously on different files in a Sun-3 configuration with variable-size client caches. (a) shows additional time required by each diskless client to complete the benchmark, relative to a single client running with local disk. (b) shows server CPU utilization. (c) shows percent network utilization.

so there was no cache consistency overhead. Figure 4 shows the effects of contention on the client speed, on the server's CPU, and on the network. Without client caches, there was significant performance degradation when more than a few clients were active at once. With seven clients and no client caching, the clients were executing two-and-a-half times more slowly, the server was nearly 80% utilized, and the network was over 30% utilized. With client caching and seven active clients, each ran at a speed within 30% of what it could have achieved with a local disk; server utilization in this situation was about 50% and network utilization was only 5%.

The measurements of Figure 4 suggest that client caches allow a single Sun-3 server to support at least ten Sun-3 clients simultaneously running the Andrew benchmark. However, typical users spend only a small fraction of their time running such intensive programs. We estimate that one instance of the Andrew benchmark corresponds to about 5-20 active users, so that one Sun-3 Sprite file server should be able to support at least 50 Sun-3 users. This estimate is based on the BSD study, which reported average file I/O rates per active user of .5-1.8 Kbytes/second. We estimate that the average total I/O for an active Sun-3 workstation user will be about 8-10 times higher than this, or about 4-18 Kbytes/second, because the BSD study did not include paging traffic and was based on slower machines used in a timesharing mode (we estimate that each of these factors accounts for about a factor of two). Since the average I/O rate for the Andrew benchmark was 90 Kbytes/second, it corresponds to about 5-20 users. This estimate is consistent with independent estimates made by Howard *et al.*, who estimated that one instance of the Andrew benchmark corresponds to five average users [HOWA87], and by Lazowska *et al.*, who estimated about 4 Kbytes/second of I/O per user on slower Sun-2 workstations [LAZO86]).

The server capacity should not change much with increasing CPU speeds, as long as both client and server CPU speeds increase at about the same rate. In a system with servers that are more powerful than clients, the server capacity should be even higher than this.

8. Comparison to Other Systems

The Sprite file system is only one of many network file systems that have been implemented in the last several years. Table 7 compares Sprite to six other well-known systems for five different parameters. Most of these parameters have already been discussed in previous sections; this section focuses on the cache consistency issues and compares Sprite's performance with NFS and Andrew.

System	Cache Location	Cache Size	Writing Policy	Consistency Guarantees	Cache Validation
NFS [SAND85]	Memory	Fixed	On close or 30 sec. delay	Sequential	Ask server on open
RFS [BACH87]	Memory	Fixed	Write-through	Sequential, Concurrent	Ask server on open
Andrew [HOWA87]	Disk	Fixed	On close	Sequential	Server calls client when modified
Locus [POPEK85]	Memory	Fixed	On close	Sequential, Concurrent	Ask server on open
Apollo [LEACH83]	Memory	Variable	Delayed or on unlock	Sequential	Ask server when lock
CFS [SCHR85]	Disk	Variable	On SModel	Not applicable	Not applicable
Sprite	Memory	Variable	30 sec. delay	Sequential, Concurrent	Ask server on open

Table 7. Comparison of file systems. All of the systems but Apollo, Cedar and Sprite are variants of the UNIX operating system. The Apollo system delineates active use of a file by lock and unlock instead of open and close. The Cedar File System (CFS) only caches immutable files and provides a different type of cache consistency than the other systems. SModel is a software tool that is used to move cached files that have been changed back to their file server.

Of the systems in Table 7, only two systems besides Sprite support both sequential and concurrent write-sharing. The RFS system handles concurrent write-sharing in a fashion similar to Sprite, by disabling caching. However, RFS is based on write-through and disables caching on the first write to a file, rather than during the open. Locus supports concurrent write-sharing without disabling caching. Instead, it uses a token-passing scheme where each client must have a read or write token in order to access a file; the kernels pass the tokens around and flush caches in a way that guarantees consistency. The Apollo system doesn't support concurrent write-sharing, but provides lock and unlock primitives that concurrent readers and writers can use to serialize their accesses; the system flushes caches in a way that guarantees consistency if the locking primitives are used correctly.

All of the systems in Table 7 except Andrew check with a file's server when the file is opened or locked, in order to ensure that the client's cached copy is up-to-date. The Andrew system used the check-on-open approach initially, but was later redesigned to have the servers keep track of which clients have cached which files and notify the clients when the files are modified. This allowed clients to open files without contacting

Benchmark	Degradation		Server Utilization		Network Utilization	
	Original	Handle Locally	Original	Handle Locally	Original	Handle Locally
Andrew	12.5%	6.2%	8.9%	5.6%	1.04%	.69%
Fs-make	5.6%	1.0%	6.7%	4.1%	.75%	.45%

Table 8. Estimated improvements possible from client-level name caching and server callback. The estimates were made by counting invocations of Open and Get Attributes operations in the benchmarks and recalculating degradations and utilizations under the assumption that all of these operations could be executed by clients without any network traffic or server involvement.

the servers and resulted in a substantial reduction in server loading. For Sprite we decided to process all opens and file naming operations on the servers in order to avoid the complexity of maintaining file name caches on clients.

Although we are generally satisfied with Sprite's performance and scalability, we were curious how much improvement would be possible if we implemented client-level name caching with an Andrew-like callback mechanism. Table 8 contains simple upper-bound estimates. The table suggests that client-visible performance would only improve by a few percent (even now, clients run almost as fast with remote disks as with local ones), but server utilization and network utilization would be reduced by as much as one-third. This could potentially allow a single server or network to support up to

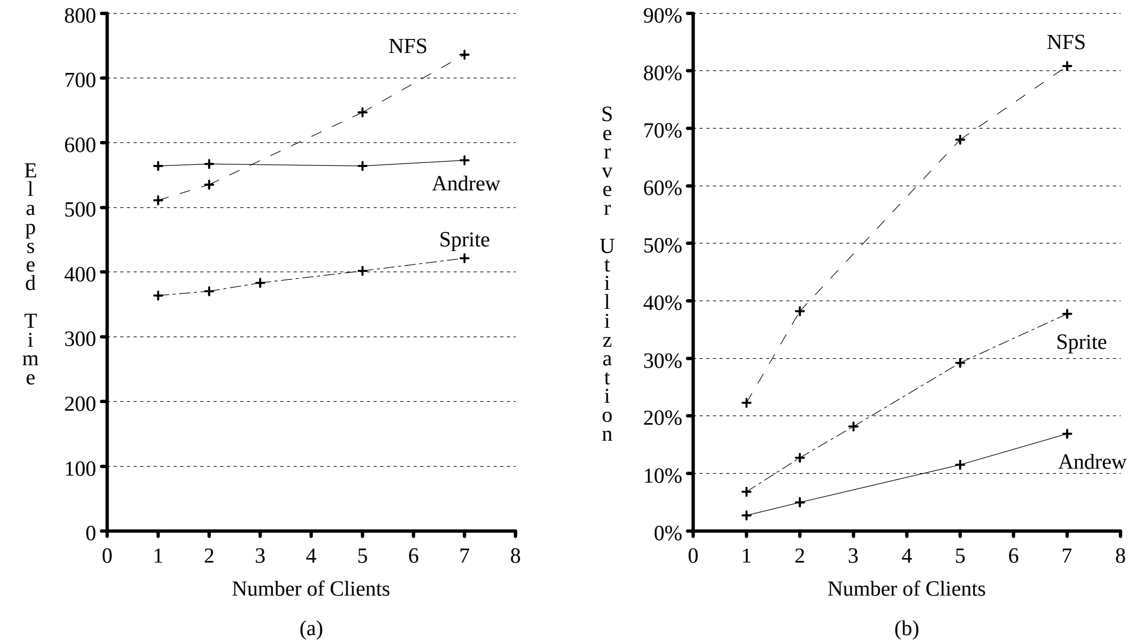

Figure 5. Performance of the Andrew benchmark on three different file systems; Sprite, Andrew, and NFS. (a) shows the absolute running time of the benchmark as a function of the number of clients executing the benchmark simultaneously, and (b) shows the server CPU utilization as a function of number of clients. The Andrew and NFS numbers were taken from [HOWA87] and are based Sun-3/50 clients. The Sprite numbers were taken from Table 6 and Figure 4 and re-normalized for Sun-3/50 clients.

50% more clients than in the current implementation. Our estimate for improvement in Sprite is much smaller than the measured improvement in Andrew when they switched to callback. We suspect that this is because the Andrew servers are implemented as user-level processes, which made the system more portable but also made remote operations much more expensive than in Sprite's kernel-level implementation. If the Andrew servers had been implemented in the kernel, we suspect that there would have been less motivation to switch to a callback approach.

Figure 5 compares Sprite to the Andrew and NFS filesystems using the Andrew benchmark. The measurements for the NFS and Andrew file systems were obtained from [HOWA87]. Unfortunately, the measurements in [HOWA87] were taken using Sun-3/50 clients, whereas we had only Sun-3/75 clients available for the Sprite measurements; the Sun-3/75 is about 30% faster than the Sun-3/50. In order to make the systems comparable, we normalized for Sun-3/50 clients: the Sprite elapsed times from Table 6 and Figure 4 were multiplied by 1.3, and the server utilizations from Figure 4 were divided by 1.3 (the servers were the same for the Sprite measurements as for the Andrew and NFS measurements; slowing down the Sprite clients would cause the server to do the same amount of work over a longer time period, for lower average utilization). Another difference between our measurements and the ones in [HOWA87] is that the NFS and Andrew measurements were made using local disks for program binaries, paging, and temporary files; for Sprite, all of this information was accessed remotely from the server.

Figure 5 shows that for a single client Sprite is about 30% faster than NFS and about 35% faster than Andrew. The systems are sufficiently different that it is hard to pinpoint a single reason for Sprite's better performance; however, we suspect that

Sprite’s high-performance kernel-to-kernel RPC mechanism (vs. more general-purpose but slower mechanisms used in NFS and Andrew), Sprite’s delayed writes, and Sprite’s kernel implementation (vs. Andrew’s user-level implementation) are major factors. As the number of concurrent clients increased, the NFS server quickly saturated. The Andrew system showed the greatest scalability: each client accounted for only about 2.4% server CPU utilization, vs. 5.4% in Sprite and over 20% in NFS. We attribute Andrew’s low server CPU utilization to its use of callbacks. Figure 5 reinforces our claim that a Sprite file server should be able to support at least 50 clients: our experience with NFS is that a single server can support 10-20 clients, and Sprite’s server utilization is only one-fourth that of NFS.

9. Future Work

There are two issues concerning client caching that we have not yet resolved in the Sprite implementation: crash recovery and disk overflow. The current system is fragile due to the amount of state kept in the main memory of each server. If a server crashes, then all the information in its memory is lost, including dirty blocks in its cache and information about open files. As a result, all client processes using files from the server usually have to be killed. In contrast, the servers in Sun’s NFS are stateless. This results in less efficient operation (since all important information must be written through to disk), but it means that clients can recover from server crashes: the processes are put to sleep until the server reboots, then they continue with no ill effects.

We are currently exploring ways to provide better recovery from server crashes in Sprite. One possibility is to use write-through in the servers’ caches. Table 4 shows that a client can write to a server’s disk at 176 Kbytes/sec, yet with client caching even the most intensive benchmark generated data for the server at less than 20 Kbytes/sec (see Figure 2(b)). Thus it appears that it might be possible to make server caches write-through without significant performance degradation. This would guarantee that no file data would be lost on server crashes. Client caches would still use a delayed-write policy, so the extra overhead of writing through to the server cache would only be incurred by the background processes that clean client caches. In addition, clients should be able to provide servers with enough information to re-open files after a server crash. We hope that this approach will enable clients to continue transparently after server crashes.

The second unresolved issue has to do with ‘‘disk-full’’ conditions. In the current implementation, a client does not notify the server when it allocates new blocks for files. This means that when the client eventually writes the new block to the server (as much as 30 seconds later), there may not be any disk space available for the block. In UNIX, a process is notified at the time of the ‘‘write’’ system call if the disk is full. We plan to provide similar behavior in Sprite with a simple quota system in which each client is given a number of blocks from which it can allocate disk space. If the client uses up its quota, it requests more blocks from the server. When the amount of free disk space is too small to give quotas to clients, clients will have to submit explicit disk allocation requests to the server whenever they create new blocks.

10. Conclusions

Sprite’s file system demonstrates the viability of large caches for providing high-performance access to shared file data. Large caches on clients allow diskless client workstations to attain performance comparable to workstations with disks. This performance is attained while utilizing only a small portion of servers’ CPU cycles. The caches can be kept consistent using a simple algorithm because write-sharing is rare. By varying the cache sizes dynamically, Sprite permits the file caches to become as large as possible without impacting virtual memory performance.

The high performance attainable with client caches casts doubts on the need for local disks on client workstations. For users considering the purchase of a local disk, our advice is to spend the same amount of money on additional memory instead. We believe that this would improve the performance of the workstation more than the addition of a local disk: it would not only improve file system performance by allowing a larger cache, but it would also improve virtual memory performance.

11. Acknowledgments

We are grateful to M. Satyanarayanan for providing us with the Andrew benchmark and his measurements of that benchmark on Andrew and NFS. Andrew Cherenson, Fred Douglis, Garth Gibson, Mark Hill, Randy Katz, and Dave Patterson provided numerous helpful comments that improved the presentation of the paper. This work was supported in part by the Defense Advanced Research Projects Agency under contract N00039-85-C-0269, in part by the National Science Foundation under grant ECS-8351961, and in part by General Motors Corporation.

12. References

[BACH87]
Bach, M.J., Luppi, M.W., Melamed, A.S., and Yueh, K. ‘‘A Remote-File Cache for RFS.’’ *Proceedings of the USENIX Summer 1987 Conference*, June 1987, pp. 275-280.

[BIRR84]
Birrell, A.D., Nelson, B.J. ‘‘Implementing Remote Procedure Calls.’’ *ACM Transactions on Computer Systems*, Vol. 2, No. 1, February 1984, pp. 39-59.

[HILL86]
Hill, M.D., et al. ‘‘Design Decisions in SPUR.’’ *IEEE Computer*, Vol. 19, No. 11, November 1986, pp. 8-22.

[HOWA87]
Howard, J.H., et al. ‘‘Scale and Performance in a Distributed File System.’’ *ACM Transactions on Computer Systems*, to appear.

[KLEI86]
Kleiman, S.R. ‘‘Vnodes: An Architecture for Multiple File System Types in Sun UNIX.’’ *Proceedings of the USENIX 1986 Summer Conference*, June 1986, pp. 238-247.

[LAZO86]
Lazowska, E.D., Zahorjan, J., Cheriton, D., and Zwaenepoel, W. ‘‘File Access Performance of Diskless Workstations.’’ *ACM Transactions on Computer Systems*, Vol. 4, No. 3, August 1986, pp. 238-268.

[LEACH83]
Leach, P.J., et al. ‘‘The Architecture of an Integrated Local Network.’’ *IEEE Journal on Selected Areas in Communications*, Vol. SAC-1, No. 5, November 1983, pp. 842-857.

[LEFF84]
Leffler, S., Karels, M., and McKusick, M.K. ‘‘Measuring and Improving the Performance of 4.2BSD’’, *Proceedings of the USENIX 1984 Summer Conference*, June 1984, pp. 237-252.

[NELS86]
Nelson, M. ‘‘Virtual Memory for the Sprite Operating System.’’ Technical Report UCB/CSD 86/301, Computer Science Division (EECS), University of California, Berkeley, 1986.

[OUST85]
Ousterhout, J.K. et al. ‘‘A Trace-Driven Analysis of the 4.2 BSD UNIX File System.’’ *Proceedings of the 10th Symposium on Operating Systems Principles*, December 1985, pp. 15-24.

[POPEK85]
G.J., Walker, B.J, editors. ‘‘The LOCUS Distributed System Architecture.’’ The MIT Press, Cambridge, Mass., 1985.

[RITC74]
Ritchie, D.M., and Thompson, K. ‘‘The UNIX Time-Sharing System.’’ *Communications of the ACM*, Vol. 17, No. 7, July 1974, pp. 365-375.

[SAND85]
Sandberg, R. et al. ‘‘Design and Implementation of the Sun Network Filesystem.’’ *Proceedings of the USENIX 1985 Summer Conference*, June 1985, pp. 119-130.

[SATY85]
Satyanarayanan, M. et al. ‘‘The ITC Distributed File System: Principles and Design.’’ *Proceedings of the 10th Symposium on Operating Systems Principles*, December 1985, pp. 35-50.

[SCHR85]
Schroeder, M.D., Gifford, D.K. and Needham, R.M. ‘‘A Caching File System for a Programmer’s Workstation.’’ *Proceedings of the 10th Symposium on Operating Systems Principles*, December 1985, pp. 25-34.

[THOM78]

Thompson, K. ''UNIX Time-Sharing System: UNIX Implementation.'' *Bell System Technical Journal*, Vol. 57, No. 6, July-August 1978, pp. 1931-1946.

[WELCH86a]

Welch, B., Ousterhout, J. ''Prefix Tables: A Simple Mechanism for Locating Files in a Distributed Filesystem.'' *Proceedings of the 6th International Conference on Distributed Computing Systems*, May 1986, pp. 184-189.

[WELCH86b]

Welch, B. ''The Sprite Remote Procedure Call System.'' Technical Report UCB/CSD 86/302, Computer Science Division (EECS), University of California, Berkeley, 1986.