# 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, California 94720

## Extended Abstract

This paper describes a simple distributed mechanism for caching files among a networked collection of workstations. We have implemented it as part of Sprite, a new operating system being implemented at the University of California at Berkeley. 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; 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.

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). On machines with disks, the caches 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.

The Sprite caches are organized on a block basis using a fixed block size of 4 Kbytes. 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.

Dirty blocks are written back to the server or disk using 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. This policy was chosen over write-through and write-back-on-close because it avoids delays when writing and closing files and permits modest reductions in disk/server traffic. Although it does not have the high reliability of write-through, it limits the amount of file data that can be lost in a crash.

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. To simplify the implementation of cache consistency, we considered two separate cases. The first case is *sequential write-sharing*, where one workstation modifies a file and another workstation reads it later, but the file is never open on both workstations at the same time. We expect this form of write-sharing to be the most common one. The second case is *concurrent write-sharing*, where one workstation modifies a file while it is open on another workstation. Our solution to this situation is more expensive, but we do not expect it to occur very often.

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.

Sprite handles sequential write-sharing using version numbers. When a client opens a file, the server returns the current version number for the file, which the client compares to the version number associated with its cached blocks for the file. If they are different, the file must have been modified recently on some other workstation, so the client discards all of the cached blocks for the file and reloads its cache from the server when the blocks are needed. The delayed-write policy used by Sprite means that the server doesn't always have the current data for a file (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; when a client other than the last writer opens the file, the server forces the last writer to write all its dirty blocks back to the server's cache. This guarantees that the server has up-to-date information for a file whenever a client needs it.

For concurrent write-sharing, where the file is open on two or more workstations and at least one of them is writing the file, Sprite disables client caching for that file. When the server receives an open request that will result in concurrent write-sharing, it flushes dirty blocks back from the current writer (if any), and notifies all of the clients with the file open that they should not cache the file anymore. Cache disabling is done on a file-by-file basis, and only when concurrent write-sharing occurs. A file may be cached simultaneously by several active readers.

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. 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.

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.

We also compared the performance of Sprite to both the Andrew file system [SATY85] and Sun's Network File System (NFS) [SAND85]. We did this by executing the Andrew file system benchmark [HOWA87] concurrently on multiple Sprite clients and comparing our results to those presented in [HOWA87] for NFS and Andrew. For a single client, Sprite is about 30% faster than NFS and about 35% faster than Andrew. 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.

## References

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

[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.

[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.

[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.