Physical Memory Management in a Network Operating System

Michael Newell Nelson

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

CHAPTER 1

Introduction

The work presented in this dissertation was motivated by two recent changes in technology: networks and large memories. The introduction of networks has led to a move away from centralized timesharing operating systems towards network operating systems. In these network operating systems each user has a personal high-performance workstation and communicates with other users across a network. Data that was once stored on a single set of disks in the timesharing systems is now distributed amongst the disks of several workstations. In fact, many of the workstations do not have any disk at all; the data for these diskless workstations is stored across the network on the disks of other workstations.

The move towards network operating systems poses two problems: how to provide users with high performance and how to allow users to easily share data. Performance is a problem in network environments because each access of data may require both a network access and a disk access. Network accesses will be required if the data that is being accessed is stored on another workstation's disk; both diskless workstations and workstations that are sharing data may have to perform many network accesses. The performance problem can be solved by using the large memories which have recently become available. The memories can be used to cache recently accessed file data and thereby eliminate many network and disk accesses.

The problem with using large memories as caches of file data is that it may make file sharing difficult. In order for users that are sharing files to get consistent results they will need to see a consistent view of the file data; if one user writes new data to a file, then subsequent reads of the file should return the most recently written data, not some old stale data. In timesharing systems, guaranteeing that each user sees a consistent view of files is easy because the data is only stored in one place; all reads and writes of file data happen to one central place so each user is guaranteed to see the same view of the file. However, in a network operating system that caches data, the data for a particular file may potentially be distributed around the network in many workstations' memories.

This thesis describes the design, implementation, and performance of several techniques that use large physical memories to provide sharing and high-performance in a network operating system. The method that I used to perform this research was to design, build and measure the Sprite file system caching mechanism and the Sprite virtual memory system as part of the Sprite operating system [OCD88]. In addition to measuring the mechanisms used daily in Sprite, I also measured a variety of alternative mechanisms; these measurements provide the first quantitative comparisons between many of the popular memory-management techniques.

One major contribution of this dissertation is an exploration of the tradeoffs in designing and implementing a distributed file data caching mechanism. I will show that by effectively utilizing large physical memories as caches of file data, workstations can achieve high performance even without using a local disk; this high performance can be achieved while providing all workstations with a consistent view of file system data and

without overloading networks or server† machines. In addition I will demonstrate the importance of the writing policy: the policy that determines when dirty data is written back to the server or the disk. I will show that writing policies have a major impact on performance.

Another contribution of this dissertation is in the area of the interaction between the file system and the virtual memory system. I will present a simple mechanism that allows the file system cache to vary in size in response to the needs of the virtual memory system and the file system. This variable-size cache mechanism provides better performance than a fixed-size file system cache of any size.

The last contribution of this dissertation is an analysis of the tradeoffs in one particular area of virtual memory management: fast process creation. When a new process is created, the process is given a copy of its parent's address space. As users begin to take advantage of large memories, the size of processes may increase, which will increase the cost of copying an address space. A common method of improving the performance of process creation is by using copy-on-write: pages in the address space are initially shared by the parent and child; a page is not actually copied until one of the processes attempts to modify it. In this dissertation I will describe a simple copy-on-write mechanism that has been implemented as part of Sprite. I will show that in practice this and other copy-on-write mechanisms may actually give worse performance than the simpler copy-on-process-creation schemes.

---

† Throughout this dissertation the term *server* will be used when referring to workstations that have local disks and the term *client* will refer to workstations that wish to access data stored on non-local disks (i.e. server machine's disks).

The rest of this chapter is divided into three sections. The first section credits the other Sprite developers who helped me perform part of this research. The next section provides an overview of the Sprite operating system, which I used to perform my research. Finally, the last section presents an overview of the dissertation.

1.1. I versus We

The research presented in this thesis was done through the development and measurement of the Sprite operating system. Sprite, which I will describe in the next section, was not a one-person project; it involved 4 other people. All of the work that I will present in this dissertation I did on my own except for the design of parts of the file system. The file system was a joint project between myself and Brent Welch, where I concentrated on the caching issues and Brent on the naming issues. In order to give proper credit to the work of others, when I describe the design of the file system in Chapter 3 and when I give the Sprite overview in the next section, I will use ‘‘we’’ instead of ‘‘I’’. In the rest of the dissertation where I describe work that I did on my own I will use ‘‘I’’.

1.2. Overview of Sprite

Sprite [OCD88] is a new operating system implemented at the University of California at Berkeley as part of the development of SPUR [Hil86], a high-performance multiprocessor workstation. 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.

The interface that Sprite provides to user processes is much like that provided by UNIX [RiT74]. The file system appears as a single shared hierarchy accessible equally by processes on any workstation in the network (see [WeO86] 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 [Wel86], which allows kernels on different workstations to request services of each other using a protocol similar to the one described by Birrell and Nelson [BiN84]. The Sprite file system uses the RPC mechanism extensively for cache management.

1.3. Thesis Overview

This dissertation covers three areas: file caching, virtual memory, and the interaction between the two. The first part of the dissertation (Chapters 2 through 5) covers issues in file caching. Chapter 2 introduces the problems in file caching and discusses previous work in this area. This includes a discussion of an important set of trace-driven analyses that measured file activity in several timeshared UNIX 4.2 BSD systems [Ous85]. These simulations yielded two important results which motivated the Sprite

caching design. First, they demonstrated the potential performance improvements possible through caching; they found that even small caches can greatly improve performance. Second, they demonstrated that the policy that is used to manage dirty data may have a big impact on performance. The best policy is to delay write-backs, so that data is initially written to the cache and then written through to the disk or server some time later.

Chapter 3 presents the design of the Sprite file system. The three goals that were the driving force behind the Sprite design were high-performance, consistency and simplicity. Like many other systems, Sprite attains high-performance by using caches on both client and server workstations. However, in order to achieve the highest performance possible the Sprite file system delays the writing of file data to the server and to disk. Under the Sprite writing policy, clients and servers do not write back file data until up to 30 seconds after the data is created. This delayed-write policy allows higher performance but also introduces extra consistency and recoverability problems which do not occur in other systems.

In spite of the complexities brought about because of the delayed-write policies, Sprite guarantees that workstations see a consistent view of 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 goal of this research was not just to build a distributed file system but also to provide quantitative measurements of the tradeoffs in cache design. Chapter 4 presents the results of running a collection of benchmark programs against Sprite and measuring the performance. On average, client caching resulted in a speedup of about 10-20% for programs running on diskless workstations, relative to diskless workstations without client caches. With client caching enabled, diskless workstations completed the benchmarks only 0-8% more slowly than workstations with disks. Client caches reduced the server utilization from about 5-27% per active client to only about 1-12% per active client. Since normal users are rarely active, my measurements suggest that a single server should be able to support at least 30 clients. In comparisons with Sun's Network File System [San85] and the Andrew file system [Sat85], Sprite completed a file-intensive benchmark 30-35% faster than the other systems. Sprite's server utilization was three times less than NFS but three times higher than Andrew.

In addition to determining the effect of client caching, I was also interested in exploring the reliability/performance tradeoff: what effect does making data storage more reliable have on performance? The writing policy has a big impact on the level of reliability. Chapter 5 gives the result of running benchmark programs with 9 different writing polices on the client and 4 on the server. The results of the benchmarks indicate that in order to achieve good performance, either the client or the server must use a delayed-write policy; the absolute best performance is when they both use delayed-write policies. More restrictive policies such as write-through can cause serious performance degradation: if write-through is used on the server and on the client then benchmark programs execute from 25-100% more slowly than if the server uses a delayed-

write policy.

The results from running benchmarks on Sprite show that large file system caches provide the best performance. However, large caches may conflict with the needs of the virtual memory system, which would like to use as much memory as possible to run user processes. Chapter 6 describes a simple mechanism through which the virtual memory system and the file system of each workstation negotiate over the machine's physical memory. This simple mechanism allows the file system cache to change in size as the relative needs of the virtual memory system and the file system change.

The Sprite negotiation mechanism requires that memory be traded between the virtual memory system and the file system. What effect does this trading have on system performance? Is there a case where the trading is so intense that a small fixed-size cache would be best? Chapter 6 presents the results from a complex benchmark that causes large shifts of memory between the virtual memory and file systems. It shows that the variable-size cache is never worse than any fixed-size cache. In the best case, when a large cache is needed, the variable-size mechanism works very well. In the worst case, when large amounts of trading are required, its performance is the same as that of a fixed-size cache.

One of the features of the Sprite variable-size cache mechanism is that it allows file- and virtual-memory data to be treated differently. For example, the virtual-memory system can be given an advantage over the file system when the two are negotiating over the use of physical memory. The later part of Chapter 6 provides measurements of the impact of penalizing the file system on the performance of two file- and

virtual-memory intensive benchmarks. It shows that penalizing the file system gives better interactive response than without a penalty while not degrading overall performance.

Most of this dissertation focuses on the file system caching mechanism and the interaction between the file system and the virtual memory system. However, I was also interested in looking at one particular virtual memory problem: copy-on-write mechanisms for fast process creation. Chapter 7 presents a simple copy-on-write mechanism that I implemented as part of Sprite. The mechanism is a combination of copy-on-write (COW) and copy-on-reference (COR). The COW-COR mechanism can potentially improve fork performance over copy-on-fork schemes from 10 to 100 times. However, in normal use, most of the pages have to be copied anyway; the overhead of handling additional page faults results in worse overall performance than copy-on-fork. A pure copy-on-write scheme would eliminate 10% of the page copies required under COW-COR, but may have worse overall performance than COW-COR on machines with virtually-addressed caches, due to additional cache-flushing overhead. Even highly optimized implementations can provide at best a 30% improvement in average fork performance.

The final chapter of this dissertation, Chapter 8, offers some conclusions.

CHAPTER 2

File Data Caching

2.1. Introduction

File system caches have been used for many years on timesharing systems to reduce the number of disk accesses. More recently they have begun to be used in distributed file systems where there are caches on both servers and clients (see Figure 2-1); the caches on server workstations are used to reduce disk traffic and the caches on clients are used to reduce network traffic and server loading. This chapter examines the previous work done in file system caching and the issues that must be addressed in

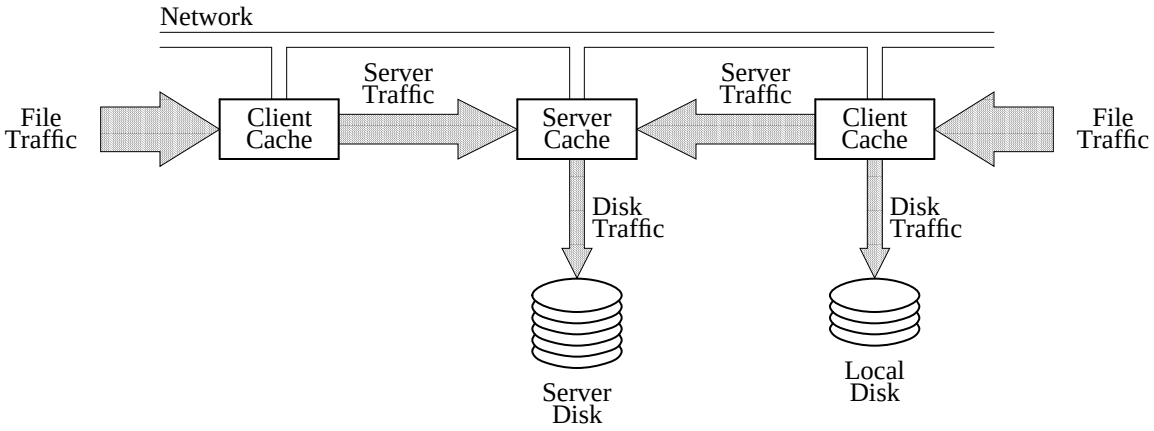

Figure 2-1. File caches in a distributed file system. When a process makes a file access, it is presented first to the cache of the process's workstation (''file traffic''). If not satisfied there, the request is passed either to the local disk, if the file is stored there (''disk traffic''), or to the server where the file is stored (''server traffic''). Servers also maintain caches in order to reduce their disk traffic.

order to build an efficient distributed caching mechanism.

2.2. Server Caches

The purpose of a server cache is to improve client performance by reducing disk accesses: data can be accessed from physical memory many times faster than from disk. The most important metric in measuring the effectiveness of a server cache is the *traffic ratio*: the ratio of physical disk accesses to logical accesses. Both reads and writes contribute to the traffic ratio. Reads will require a disk access if the data being read is not resident in the cache and writes will require a disk access if the modified data is written to disk. How the write traffic impacts the traffic ratio depends on the writing policy (see Section 2.4).

Although server caches have been implemented in several systems, the effectiveness of server caches in these systems has not been analyzed in any detail. However, there have been several attempts to predict the effectiveness of server caches by extrapolating from traces of timesharing systems. A cache on a file server that services multiple clients should have behavior similar to that of a cache on a timesharing system with multiple users; in both cases the cache is a centralized resource that is shared by many users, where each client workstation represents a single user.

One study of server caching was a trace-driven analysis of file activity in several timeshared UNIX 4.2 BSD systems [Ous85]. This study provided the main motivation for the Sprite cache design and I will refer to it extensively throughout this chapter. The systems studied by Ousterhout *et al.* were used for program development, text formatting, and computer-aided design. The study determined that for the traced systems

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 percent. The actual improvement that can be gained from caching depends on the writing policy, which will be explained below.

A study very similar to Ousterhout's study was done by Kent at Purdue [Ken86]. He also did a trace-driven analysis of file activity in a timeshared UNIX 4.2 BSD system, and his results were nearly identical to Ousterhout's results.

One other study of disk caching was done by Smith, who used trace data from IBM mainframes [Smi85]. Smith reported reductions in disk traffic similar to those reported in Ousterhout's study even though his data was much different. Unfortunately Smith's data did not distinguish read accesses from write accesses. Thus, he did not determine the impact of the writing policy on the traffic ratio. Nevertheless, his results indicate that caches from 2 to 8 megabytes are very effective, reducing disk traffic by over 80 percent.

The results from the trace-driven analyses of timesharing traces indicate that server caches should be very effective in reducing disk accesses. However, this has not been verified by either measurement of existing systems or trace-driven analyses of traces taken from networks of workstations.

2.3. Client Caches

Whereas the purpose of caches on server workstations is to reduce disk accesses, the purpose of caches on client workstations is to reduce network accesses. If client caches are as effective in reducing network traffic as server caches appear to be in

reducing disk traffic, then caches on clients could have a great impact on the performance of clients, the load on file servers and the load on the network. A reduction in the load on the network and the server will result in greater system scalability because there can be more clients per network and more clients per server. The relation between server load and system scalability was shown by Lazowska *et al.* [LZC86] in a study of remote file access where they concluded that the server CPU is the primary bottleneck that limits system scalability.

Caches can be used on clients for two purposes: to cache file data and to cache naming information. Caching of file data reduces the number of read and write operations that require server accesses, and caching naming information can reduce the number of open and close operations that require server accesses. In this section I will concentrate on data caching, and in Section 2.5 I will explore the impact of name caching.

Systems that have implemented client caching have taken one of two approaches: cache file blocks in memory (e.g. LOCUS [PoW85, Wal83] and Sun's Network File System (NFS) [San85]) or cache whole files on a local disk (e.g. Andrew [Mor86, Sat85] and Cedar [SGN85]). The advantage of caching on a local disk is that local disks are generally much larger than physical memories. However, caching in main memory has numerous advantages over caching on a local disk. First, main-memory caches permit workstations to be diskless. Second, data can be accessed much more quickly from a cache in main memory than a cache on a local disk. Third, if the studies done by Ousterhout or Kent are indicative of client cache performance, then physical memories on client workstations are already large enough to provide high hit

ratios. As memories get larger, main-memory caches will grow to achieve even higher hit ratios.

Although several systems have implemented client caching in various forms, none of these systems has been analyzed to determine the impact of caching on system performance. For example, Howard *et al.* [How88] showed that with caches on clients, the load placed on the server by each client is very small. However, they did not determine what the load would have been if there had been no caches on the client workstations. The only analyses of the impact of client caching have been made with trace-driven simulations from UNIX timesharing traces. These simulations have shown that client caching can be effective in reducing network and server loading. Since the simulations have depended on the writing policy and the cache consistency policy used, I will not discuss the results of these simulations until after I have discussed the writing policy issues and cache consistency policies.

2.4. Writing Policy

The performance advantages of caching depend on the policy used for handling modified data blocks. In a distributed system, both the writing policy used on servers and the policy used on clients can have a performance impact. Although different file systems have used different writing policies, there have been no measurements of the performance impact of the writing policy. However, results from four studies of UNIX timesharing traces can be used to help predict the best writing policy for clients and servers. In addition to the two previously-mentioned studies by Ousterhout and Kent there are also studies that were done by Floyd [Flo86] and Thompson [Tho87]. Floyd's

studies are nearly identical to Ousterhout's studies so I will not mention them further. Thompson's study was a follow-on study to the study done by Ousterhout *et al.*; Thompson's results are based on very detailed traces of UNIX timesharing systems.

The simplest policy for managing modified data blocks is to write them through to the server and/or the disk as soon as they are placed into the cache. NFS uses write-through on the server and RFS [BLM87] uses write-through on clients. The advantage of a write-through policy is its reliability: little information is lost when a client or server crashes. However, each write must wait for the data to be written to the server and/or disk, which results in poor write performance. Also, Ousterhout's study determined that about 1/3 of all file accesses are writes. This means that with a write-through policy disk or server traffic cannot be reduced by more than about 2/3. Kent's study of UNIX file system activity confirmed this by demonstrating that with a write-through policy the traffic ratio was over 27 percent.

An alternative policy to write-through is *buffered write*, which delays the write to the server or disk until the last byte of a cache block is written. If a user writes data in chunks smaller than the file system block size, then disk and network traffic can be reduced. This is actually the policy that was used by the Ousterhout study when the authors measured the effect of different writing policies. Thompson simulated this policy and discovered that over half of all write traffic caused by a pure write-through policy can be eliminated with buffered write. Thus even buffering a single block can have a profound effect on writing performance.

The Andrew and LOCUS systems use a writing policy called *write-back-on-close*. Under this policy, writes return as soon as the data is in the cache, but the data is written back to the server when the file is closed. This results in better write performance but causes processes to wait when they close the file.

The policy used by NFS clients is a combination of write-back-on-close and write-back-as-soon-as-possible (ASAP). When data is written to the cache it is scheduled to be written through to the server as soon as possible†, but the write returns immediately. When the file is closed, the client ensures that all of the file data has been written through to the server. This should have similar performance to a pure write-back-on-close policy except that the close of the file may not have to wait as long because some of the dirty data may have already been written back when the file is closed. Unfortunately, the Ousterhout study determined that most files are open only a very short period of time: 75% of files are open less than 0.5 seconds and 90% less than 10 seconds. These short open times imply that many files may be not be open long enough to allow their dirty blocks to be written back before the file is closed.

The best policy for performance is to delay the writing of blocks until the block is ejected from the cache. A delayed-write policy has two advantages. First, writes and closes can complete without waiting for data to be written through. Second, Ousterhout's study determined that 20 to 30 percent of new data is deleted within 30 seconds and 50 percent is deleted within 5 minutes. Under a delayed-write policy, many blocks will never need to be written to disk at all; they will live and die in the

---

† NFS actually does not schedule the write-back of the block until the block is full.

cache. Unfortunately, a delayed-write policy has reliability problems, since large amounts of data can be lost during a system crash. UNIX uses a compromise solution in which blocks are not written through to disk until they have been in the cache for 30 seconds. This gives better reliability than a true delayed-write policy, yet eliminates 20 to 30 percent of server and/or disk writes.

A different type of policy that could be used is a combination of delayed-write and write-through policies depending on the file type. This type of policy has not been implemented in any system, but Thompson simulated two mixed policies. In one policy he varied from a 1 second delayed-write policy for editor temporaries up to full delay for temporary files (he called this the *mixed-policy*), and in the other policy he used buffered-write for all except temporary files (he called this the *delay-temp* policy). The delay-temp policy provides a write traffic ratio slightly lower than the 30-second-delay policy. The mixed-policy lies between the delay-temp policy and a 5-minute-delay policy. Thus, by special-casing temporary files, clients can get write-traffic ratios that are better than a 30 second delayed-write policy, but with higher reliability.

One thing to note about all of the UNIX studies is that their data does not include writes of file meta-data: data that describes the contents of the file. In a UNIX file system there are two types of meta-data: indirect blocks and file descriptor blocks. File descriptors describe the attributes of the file and where the first few blocks for the file are on disk. Indirect blocks are used to describe where the data blocks for large files are kept on disk. Depending on the implementation of the file system, each write-back of data may require writes of both indirect blocks and file descriptor blocks. For example, if a write-through policy is used on a server, then each time that a data block is written

to disk for a large file both the file descriptor and the indirect block must be written to disk as well; if the descriptor and indirect blocks are not written to disk, then during a system crash the location of the data block may be lost. Thus, because of file meta-data, write-through and similar types of policies may cause the traffic ratio to go up by at least a factor of three.

2.4.1. Client and Server Writing Policies

In a system that contains both clients and servers, the best approach may be to use different policies on the client and the server. For example, a policy that uses write-through on servers and delayed-write on clients would result in no loss of data from a server crash, yet allow clients to achieve very high performance. Unfortunately, there have been no simulations or measurements of the various combinations of client and server writing policies.

2.5. 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? The definition of consistency that I will use is that a client workstation sees a consistent view of a file if each read operation returns the most recently written data for the file. The class of cache consistency algorithms that I will examine in this section are all based on performing consistency on a per-file rather than a per-block basis. This is the method used in most existing file systems and is practical because studies have shown that files are generally read and written in their entirety [Ous85]. Per-file approaches are simpler and can potentially lower

the cost of consistency by requiring fewer consistency actions (one per file rather than one per block).

It is important to distinguish between consistency and correct synchronization. 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. The purpose of cache consistency is to eliminate the network issues and reduce the problem to what it was on timesharing systems.

There are two types of write sharing that can cause consistency problems: sequential write-sharing and concurrent write-sharing (see Figure 2-2). Sequential write-sharing occurs when a file is shared but is never open simultaneously for reading and writing on different clients. This can result in clients maintaining stale data for a file in their cache after they have closed the file. In order to achieve consistency, the client must be able to detect this stale data by the time it reopens the file.

The other type of sharing is concurrent write-sharing. This type of sharing occurs when a file is open on one or more clients at the same time and at least one of the clients modifies the file. In this case a client must be able to detect its stale data whenever it attempts to read data from the file.

The amount of file sharing that occurs has an impact on the importance of cache consistency. Jim Thompson analyzed the amount of file sharing that occurred in a UNIX environment [Tho87] and got several interesting results:

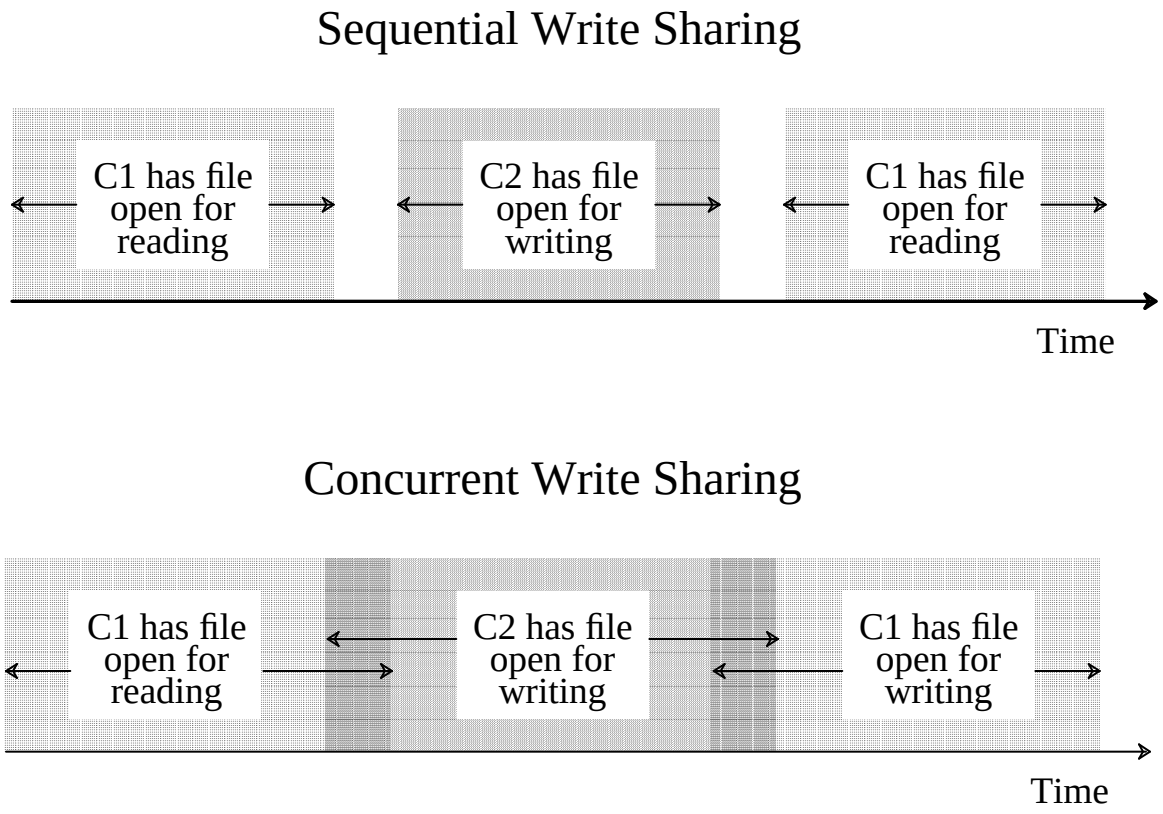

Figure 2-2. Sequential and concurrent write sharing. The figure on the top shows sequential write sharing. C1 opens a file for reading, loads blocks into its cache and then closes the file. C2 then opens the same file, modifies it and closes. When C1 opens the file again it needs to make sure that the data that it loaded into its cache from the first open is not stale; C2 could have overwritten data that C1 had previously loaded into its cache. The figure on the bottom shows concurrent write sharing. C1 opens a file for reading and before it closes it C2 opens the same file for writing; the dark shaded region on the left shows the time where C1 and C2 are concurrently read-write-sharing the file. After C2 opens the file C1 closes the file and then opens the file for writing before C2 closes the file; the dark shaded region on the right shows the time where C1 and C2 are concurrently write-write-sharing the file.

- 2.2% of the opens of files resulted in concurrent write-sharing.
- Only 2% of the bytes transferred were to files that were undergoing concurrent write-sharing.
- Nearly all concurrent write-sharing occurred to a single file, the */etc/utmp* file, which keeps track of users logged on.

- Slightly more than 25% of all opens occur to files that are sequentially write shared.

These results indicate that although concurrent write-sharing does happen, it is very rare. In contrast sequential write-sharing happens fairly frequently (one out of every 4 opens).

2.5.1. Previous Implementations of Cache Consistency

Each of the many network file systems in existence provides a different implementation and level of consistency. This section gives a survey of the current methods used for cache consistency. All of the file systems that I will describe cache file data on both client and server workstations.

2.5.1.1. NFS

NFS is based on *stateless servers*, which means that servers keep no information that can be lost upon a server crash. This requires all state to be kept in non-volatile memory (i.e. on disk). As a consequence of the stateless implementation, servers keep no information about which clients have files open. This makes precise cache consistency difficult. The result is that NFS does not provide exact cache consistency for either type of sharing. If a file is undergoing concurrent write-sharing, then the outcome is undefined. Users are warned to avoid this type of sharing. Sequential write-sharing is handled using a probabilistic approach. Each client caches file version information for three seconds. If when a file is opened, the local version information is less than 3 seconds old, then the client believes that it has the most recent copy of the file. Otherwise it will verify its version with the file's server and flush its cache if necessary.

2.5.1.2. Cedar

The Cedar file system [SGN85] provides consistency through the use of ‘‘immutable files.’’ Each time that a file is modified, a new version of the file is created. When a file is opened, a user specifies which version of the file to use. If the user specifies a version that the client does not have cached on its disk, then a new copy of the file is loaded from the server. Once a client opens a given version of the file, it is guaranteed to see a ‘‘consistent’’ view of that version because the file is immutable; if two clients are concurrently write-sharing a file, they will both be accessing different versions of the file. Note that Cedar does not satisfy my definition of cache consistency because once a file is open reads are not guaranteed to return the most recently written data.

2.5.1.3. Andrew

Andrew [Mor86, Sat85] only supports sequential write-sharing. If two clients are undergoing concurrent write-sharing, then clients will not see a consistent view of the file. Sequential write-sharing is supported by guaranteeing that, once a file is closed, all data is back on the server, and by ensuring that a client is notified by the server whenever the client’s cached copy becomes out-of-date.

2.5.1.4. LOCUS

LOCUS [PoW85, Wal83] supports both concurrent and sequential write-sharing. It uses a complex mechanism based on passing tokens between workstations that are accessing the file. There are two types of tokens: read and write. A client must possess a token in order to access a file. Multiple clients may hold a read token if there is no write token. If there is a write token, then no client may possess a read token and only

one client may hold the write token. When a token is released, the file that the token pertains to must be written back to the server and invalidated from the cache. The algorithm must ensure that all sharers of a file get a fair chance at accessing the file.

2.5.1.5. Apollo

The Apollo Aegis file system [LLH85, Lea83] uses file locking to guarantee consistency; consistency is not guaranteed unless clients lock files before they perform read or write operations. A file can be locked by multiple clients when there are only readers, and by only a single client if the file is locked for writing. Caches are kept consistent by bringing a file to a consistent state when a client locks a file. Before a client reads or writes a newly locked file, all stale data is removed from the client's cache and the server makes sure that it has the most recent data from the file. The file system guarantees that the server has the most recent data by writing back all modified data whenever a file is unlocked. Like in NFS, stale data is eliminated by associating a version number with each file. This version number is the time that the file was last modified. It is stored in the server that stores the file and in each client that has pages of the file stored in its memory. When a client locks a file, it compares its version number for the file with the version number returned by the server. If the version numbers do not match, then the client removes the file's blocks from its memory.

2.5.1.6. RFS

The RFS system [Rif86] handles both sequential and concurrent write-sharing. Sequential write-sharing is handled by using a write-through writing policy and by contacting the server whenever a file is opened to ensure that the cached copy is up to date.

RFS handles concurrent write-sharing by disabling client caching when it occurs. Since RFS is based on write-through and hence must contact the server on every write, it can detect on the first write to a file that concurrent write-sharing is about to occur. When it detects this, it forces all reads and writes to go through to the server for the file that is being shared.

2.5.1.7. V Storage Server

The V Storage Server at Stanford [ChR85] provides multiple approaches to consistency. One approach is called *T-consistency* and is used for immutable files. The data pages read from an immutable cached file are consistent with some version of the file, either the current version or a version that is at most T milliseconds out of date. Each client polls the server of cached files every T milliseconds to determine if its cached files are up to date. The other approaches to consistency rely on block- or file-level locking.

2.5.2. Verifying Consistency

All of the consistency mechanisms that I have described require that a client be informed when a cached copy becomes out of date. This can be done in two ways: the client can ask the server about the state of the file before it begins using it, or the server can inform the client when the client's cached copy becomes out of date. The first approach generally requires that the server be contacted whenever a file is opened. This has the advantage over the second approach that it does not require that clients use local name caching; the server can do all name lookups for the client. However, because the second approach allows opens to happen locally, it offloads the server and the network,

and decreases the amount of time that it takes for a client to open a file. Most systems verify consistency when a file is opened or locked. The Andrew file system initially verified consistency when a file was opened, but, after discovering that their servers were becoming seriously overloaded, they changed to use the second approach [How88].

2.6. Trace-Driven Analyses of Client Caching

Jim Thompson used UNIX traces gathered from a single timeshared machine to perform a trace-driven simulation of the impact of client caching on performance [Tho87]. In his simulations every user on the timesharing system represents a different client. His measurements depend on which of 5 cache consistency algorithms are used; all of his algorithms provide consistency for both concurrent and sequential write-sharing. One of the cache consistency policies that Thompson simulated is the Sprite policy, which I will describe in the next chapter; I will examine his results in more detail after I describe the Sprite policy (see Section 3.3.4.2).

Thompson used two metrics to measure the impact of client caching. One is the *miss ratio*, which is an indication of the effect of client caching in reducing server interactions. The other metric is the *transfer ratio*, which reflects both server load and network bytes transferred for all types of client requests including reads, writes and opens. Thompson's results indicate that, depending on the cache consistency policy used, client caching can cut the miss ratio to 5-30 percent and lower the transfer ratio to 23-45 percent. Thus, client caching can potentially make clients run up to 20 times as fast and reduce server and network loading by more than a factor of 4. However,

Thompson's studies are merely an indication of the effect of client caching on performance. The actual impact will depend on the fraction of time that each client spends doing file system operations.

2.7. Summary and Conclusions

This chapter has explored the important issues in file data caching and its impact on performance by looking at previous work done in this area. Because there has been little measurement of the impact of file caching on real systems, the impact of caching on performance can only be predicted by using the results of trace-driven simulations of data taken from timesharing systems (e.g. from UNIX). The simulations show that caches on client and server workstations can potentially have a large impact on performance; the caches on servers can reduce the number of disk accesses, and the caches on clients the number of server accesses. However, the simulations can only predict the impact of caching on performance; the actual impact of caching on performance must be determined by measuring a real system.

One important factor when designing a caching mechanism is the writing policy. In a system that uses both client and server caching, the writing policy on both the client and the server is important. Unfortunately, there have been simulations of writing policies that have looked at either the server's policy or the client's policy, but not both together. Simulations indicate that the most effective writing policy is the delayed-write policy, which provides the lowest number of disk and server accesses and the smallest delay to user processes. However, delayed-write policies are also the least reliable policies.

Another important factor to consider when designing a file system that uses client caching is the cache consistency policy. In order to allow users to share files as easily in a distributed system as they once could on timesharing systems files must be kept consistent. However, most current distributed systems do not provide the same level of consistency that was available in timesharing systems; some do not provide consistency at all and others do not handle the case when a file is being concurrently write-shared.

In summary, previous work in the area of file data caching has been lacking in several important areas. First, there has not been any measurement of real systems; all results have been obtained through trace-driven simulation. This goes for analyses of caching performance, the effect of writing policies and the impact of cache consistency. Second, there has not been any analysis of writing policies where both the client and the server policies have been taken into account. Finally, most systems do not provide strong enough consistency. The next three chapters address these areas by presenting the design and measurement of the Sprite file system.

CHAPTER 3

Sprite File System Caching

3.1. Introduction

We had four main goals in mind when designing the Sprite caching mechanism:

- To build a high-performance file system for both clients with disks and clients without disks.
- To gain insight into the tradeoffs involved in building a caching mechanism.
- To maintain UNIX semantics including supporting all normal user-level file system operations.
- To keep things as simple as possible.

From the results given in the previous chapter, it was evident that the way to attain the highest-performance file system was to use large file data caches on both clients and servers. In addition, non-write-through caching on clients was clearly the method to use to attain the highest possible writing performance; we chose to use a 30-second delayed-write policy like the one used in the original versions of UNIX.

Although it was clear that caching was necessary to attain high performance, it was not clear whether caches on clients were absolutely necessary; maybe caches on servers would be enough. If client caches could be eliminated, then many portions of the file system could be simplified; for example, there would be no cache consistency problems. I was interested in measuring the impact of caching on diskless client

performance, network loading, and server loading. In order to allow these measurements to take place the Sprite file system can disable client caching. This ability to turn off caching is also used as part of the Sprite cache consistency algorithm.

In addition to providing clients with high performance, we also wanted to provide the same view of file data to users of the Sprite distributed file system at that given by timesharing UNIX; this includes providing the same user-level file system operations that are supported by UNIX (see Table 3-1 for a list of file system operations supported by Sprite). On timeshared UNIX, all the files and processes are on a single machine, so each read returns the most recently written data; thus, users do not have to take any explicit actions such as file locking in order to ensure data consistency. This allows users to easily share file data without worrying about inconsistencies. In order to allow easy sharing in Sprite, we provide a simple cache consistency mechanism that keeps caches consistent both for concurrent and sequential write-sharing.

Sprite User-Level File System Operations	
Operation	Action
open	Open a file given a name.
close	Close a file.
read	Read data from a file.
write	Write data to a file.
get attributes	Get the attributes of a file such as access times, file size and permissions.

Table 3-1. User-level operations supported by the Sprite file system. There are other operations supported by Sprite (such as flock) but they are not relevant to the caching issues described in this chapter.

A high-performance distributed file system, especially one that maintains cache consistency, can potentially be complex. However, during the implementation of the file system, we tried to make design decisions that would allow us to simplify the implementation without sacrificing performance or consistency. One major simplifying design decision was that we decided to do no local name caching; all naming operations on files (e.g., open) and all closes of files must go through to the server of the file. This simplified the file system for two reasons. First, we did not have to worry about name caching at all. Second, it allowed us to build a very simple data cache consistency algorithm. However, it had the potential to increase server load, as was discovered by the Andrew file system when its authors also required that the server be contacted on each file open [How88]. The next chapter will include a discussion of the impact of this decision on Sprite file system performance.

The rest of this chapter covers the design of the caching mechanism in the Sprite file system, and is organized as follows: Section 3.2 covers the basic structure of the cache; Section 3.3 presents the Sprite cache consistency mechanism; Section 3.4 describes how files are represented on disk; Section 3.5 covers details of the implementation of the file system, including discussions of reliability and crash recovery.

3.2. Basic Cache Structure

The Sprite caches are organized on a block basis using a fixed block size of 4 Kbytes. The cache block size corresponds to the disk block size, which is also 4 Kbytes. We chose the disk block size based on the results obtained by McKusick *et al.* [MJL84], who determined that large block sizes on the order of 4 Kbytes result in sub-

stantially better file system performance than smaller block sizes. In addition, studies by Kent [Ken86] and Ousterhout [Ous85] also demonstrate the virtues of a large block size. Whether the disk block size should be even larger is an open question which we will address as we gain more experience with the system.

The choice to use a fixed block size was dictated by our striving for simplicity. The other option was to use block sizes in the range from 1 Kbyte up to 4 Kbytes depending on the amount of data in the block. The potential advantage of this scheme is that it may waste less space than the fixed block size scheme. However, it is more complex and, as memories get larger, the advantage of conserving file system cache space should diminish.

3.2.1. Block Addressing

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 allocate physical disk blocks. Virtual addressing also allows blocks in the cache to be located without traversing the file's disk map. By using virtual addresses we were able to use the same implementation for the client cache as for the server cache.

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 server's cache using the blocks' physical disk addresses along with a special ''file identifier'' corresponding to the physical device.

Although a file's disk map does not have to be consulted when locating a block in the server's cache, the map does have to be used when the block is read into the server's cache and when it is written to disk. Since looking in a file map is a fairly expensive operation, the server keeps with each cache block the physical location of the block on disk. In this way. the location of the block on disk only has to be looked up when it is put into the cache, not when the block is written out to disk.

3.2.2. Writing Policy

As mentioned earlier, Sprite uses a 30-second delayed-write policy. Under this policy, blocks are initially written only to the cache, and then written back 30 seconds later. This policy is used both on servers and clients, and is implemented by having a process scan through the cache every 5 seconds and schedule write-backs for all dirty blocks that have not been modified in the last 30 seconds. A block written on a client will be written to the server's cache in 30-35 seconds, and will be written to disk in 30-35 more seconds. Thus a block can be dirty for up to 70 seconds before it ends up getting written back to disk.

3.2.3. Block Management

Sprite uses a *least-recently-used* (LRU) block replacement strategy. Each block in the cache that contains valid data is kept on a linked list called the *LRU list*; whenever a block is accessed, it is moved to the tail of the list. All blocks that do not contain valid data are kept on a separate list called the *free list*. A new block is allocated in the following manner. If the free list contains a block, then the first block on the free list is used. Otherwise blocks are removed from the head of the LRU list until a clean block

is found; any dirty blocks that were removed from the head of the LRU list are scheduled to be written back to the server's cache or disk. Once a new block is allocated it is moved to the tail of the LRU list.

Dirty blocks that need to be written back are kept on a dirty list that is associated with each file, and all files with non-empty dirty lists are kept on a list of dirty files. The dirty blocks are written back by a group of block cleaner processes. A dirty block is scheduled to be written back either because it comes to the head of the LRU list or because it is dirty and it has not been modified in 30 seconds. When a block is scheduled for write back, it is put onto the dirty list for the file in which it resides, the file is put onto the list of dirty files, and one of the block cleaner processes is awakened and given the responsibility of writing back all the blocks on the file's dirty list. In order to reduce synchronization problems, there is only one process writing back a file's dirty blocks at any given time. Normally, after a block is written back, it is left in its current position in the LRU list. However, if the block was placed onto the dirty list because it came to the head of the LRU list and needs to be recycled, then it is put onto the free list instead (see Figure 3-1 for a summary of the list data structures).

3.2.4. Synchronization

The Sprite kernel is written so that multiple processes can be executing in the kernel at the same time. Since multiple processes could be accessing the same file at the same time, the file system uses locking to ensure that only one operation is occurring on a file at once. These operations include reading, writing, opening, closing, and getting the attributes of a file. If multiple user processes wish to access the same file at the

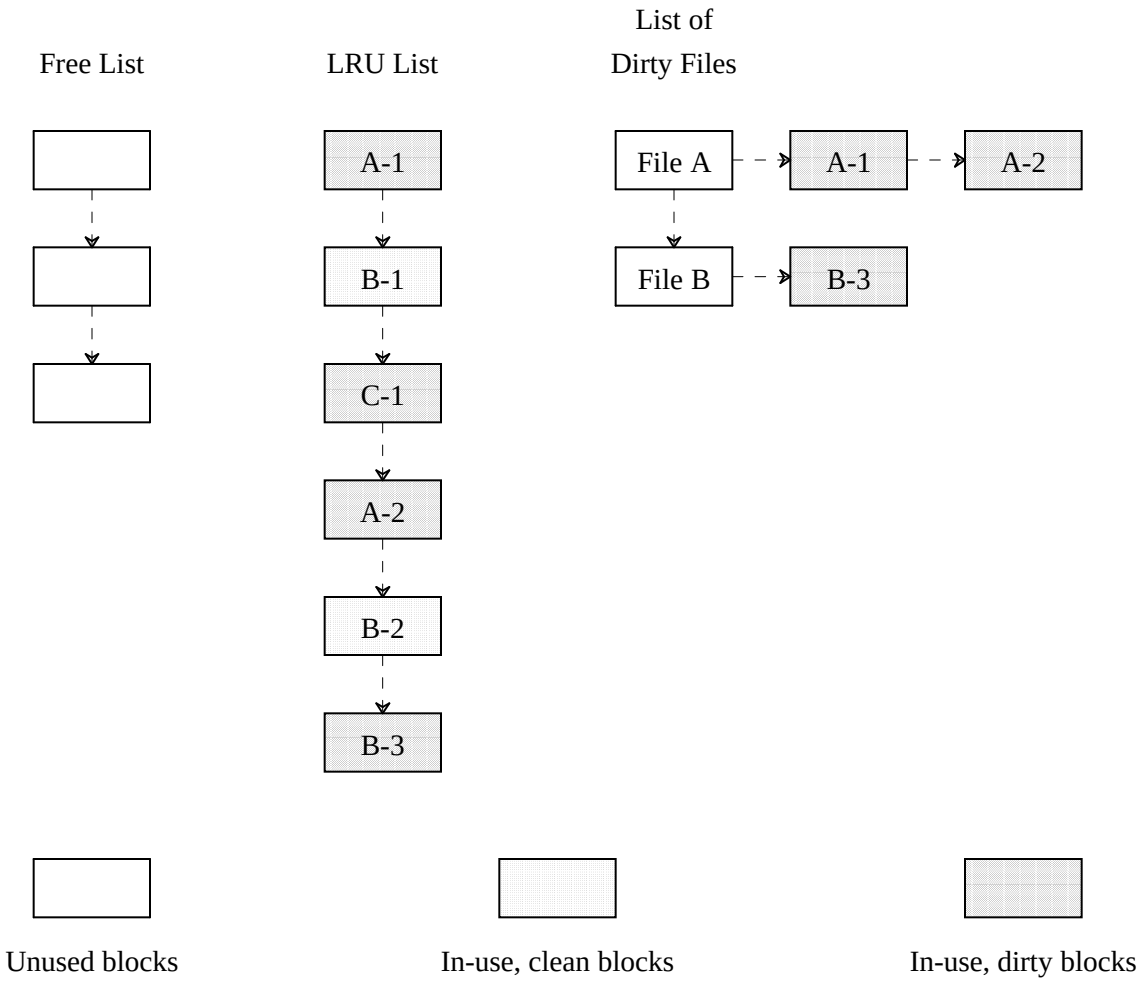

Figure 3-1. List data structures. The file system maintains three global lists and one per-file list. All blocks that are not currently being used to cache file data are on the free list. All blocks that are being used to cache data are on the LRU list. Dirty blocks that are scheduled to be written back are on the dirty list for the file that they reside in and all the file dirty lists are linked together. In this example there are 3 unused blocks that are on the free list. The LRU list contains 2 blocks from file A (denoted A-1, A-2), 3 blocks from file B (denoted B-1, B-2 and B-3) and one block from file C (denoted C-1). Blocks A-1 and A-2 are dirty and they are both on file A's dirty list because they have been scheduled to be written back. Block B-3 is dirty and it is on file B's dirty list because it also has been scheduled to be written back. Block C-1 is also dirty but it is not on file C's dirty list because it has not been scheduled to be written back yet.

same time, the accesses will be serialized once the processes begin executing inside the file system code. This explicit locking is required in order to protect kernel data structures that are associated with each file.

3.3. Cache Consistency

The Sprite file system provides cache consistency for both concurrent and sequential write-sharing. However, because of the expected infrequency of concurrent write-sharing, the algorithm is optimized for the case when there is no concurrent write-sharing. 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.

3.3.1. Concurrent Write-Sharing

Concurrent write-sharing occurs for a file when it is open by 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 (both reads and writes) 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.

3.3.2. 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 the second client opens the 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 those of NFS and of the early versions of Andrew.

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.

3.3.3. Simulation Results

3.3.3.1. Cache Consistency Overhead

While we were designing the Sprite caching mechanism, I used the trace data from the Ousterhout *et al.* study to estimate the overheads associated with cache consistency. I also estimated 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 [Ous85] 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 3-2. Client caching reduced server traffic by over 70%, and resulted in read hit ratios of more than 80%.

Table 3-3 presents similar data for a simulation where no attempt was made to guarantee cache consistency. A comparison of the bottom-right entries in Tables 3-2 and 3-3 shows that about one-fourth of all server traffic in Table 3-2 is due to cache consistency. Table 3-3 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

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 3-2. 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 improvements that might be possible with a more clever cache consistency mechanism.

I performed these simulations before we implemented our Sprite file system design, so that I could determine if our design was sound. The results from these simulations show that a) client caching can greatly reduce server traffic and b) our cache

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 3-3. Traffic without cache consistency. Similar to Table 3-1 except that cache consistency issues were ignored completely.

consistency algorithm does not introduce a significant overhead. These results strengthened our hypotheses about the effectiveness of client caching and our simple cache consistency algorithm, and indicated to us that we should proceed with the implementation.

3.3.3.2. Simulation of Several Mechanisms

Jim Thompson [Tho87] did a much more detailed simulation of cache consistency policies than we did. He simulated not only the Sprite policy, but several other policies as well. His simulation was done after we had already implemented the Sprite mechanism, used the same detailed traces that were described in Chapter 2, and used his transfer ratio, a complex measure of server and network loading, as the metric by which to judge performance. The Sprite mechanism was by far the simplest of all the mechanisms that he simulated, but also had the worst performance of all of the methods, with a transfer ratio of 45%. He estimates that the transfer ratio can be lowered to 35% if opens and closes do not have to go through to the server and to 23% if very sophisticated and potentially less practical algorithms are used. The result is that a sophisticated algorithm can reduce the transfer ratio by up to a factor of 2.

Thompson's simulations indicate that the Sprite algorithm may provide a much higher load on the network and the server relative to more sophisticated algorithms. The results in the next chapter will support Thompson's results by showing that, if clients are allowed to cache naming information so that they can open and close files without contacting a server, the server utilization and network utilization can be cut by nearly a factor of 2. However, the next chapter will also show that, even with the sim-

ple Sprite cache consistency algorithm, client caching provides excellent diskless client performance while reducing the server load and network load to very reasonable levels. Thus, although more complex cache consistency algorithms may reduce server and network loading, in practice it does not matter; the use of client caching is much more important to performance than which cache consistency algorithm is used.

3.4. Sprite File Structure on Disk

The Sprite file system's data structures used to describe where files are located on disk are similar to the UNIX data structures. Each disk contains three types of data: file descriptors, file data blocks and indirect blocks. Among other file attributes, each file descriptor contains information about where on disk a file's data blocks are located. Each descriptor contains 10 *direct block pointers*, one *singly-indirect block pointer* and one *doubly-indirect block pointer* (see Figure 3-2).

The file descriptors contain low-level descriptions of files. Built on top of the file descriptors is the directory structure, which gives a mapping from a file name to a file descriptor. As in UNIX, in Sprite directories are stored like normal files. Each directory contains a list of (file name, file descriptor id) pairs; the file descriptor identifier is used to locate the file descriptor for the file.

Although Sprite's file descriptor and directory structures are similar to those in UNIX, the organization of the disk is different; we decided to concentrate our efforts on building an efficient caching mechanism rather than on optimizing disk performance. All of the file descriptors are grouped together at the beginning of the disk; since each file descriptor is only 128 bytes, each file system block contains 32 file descriptors. The

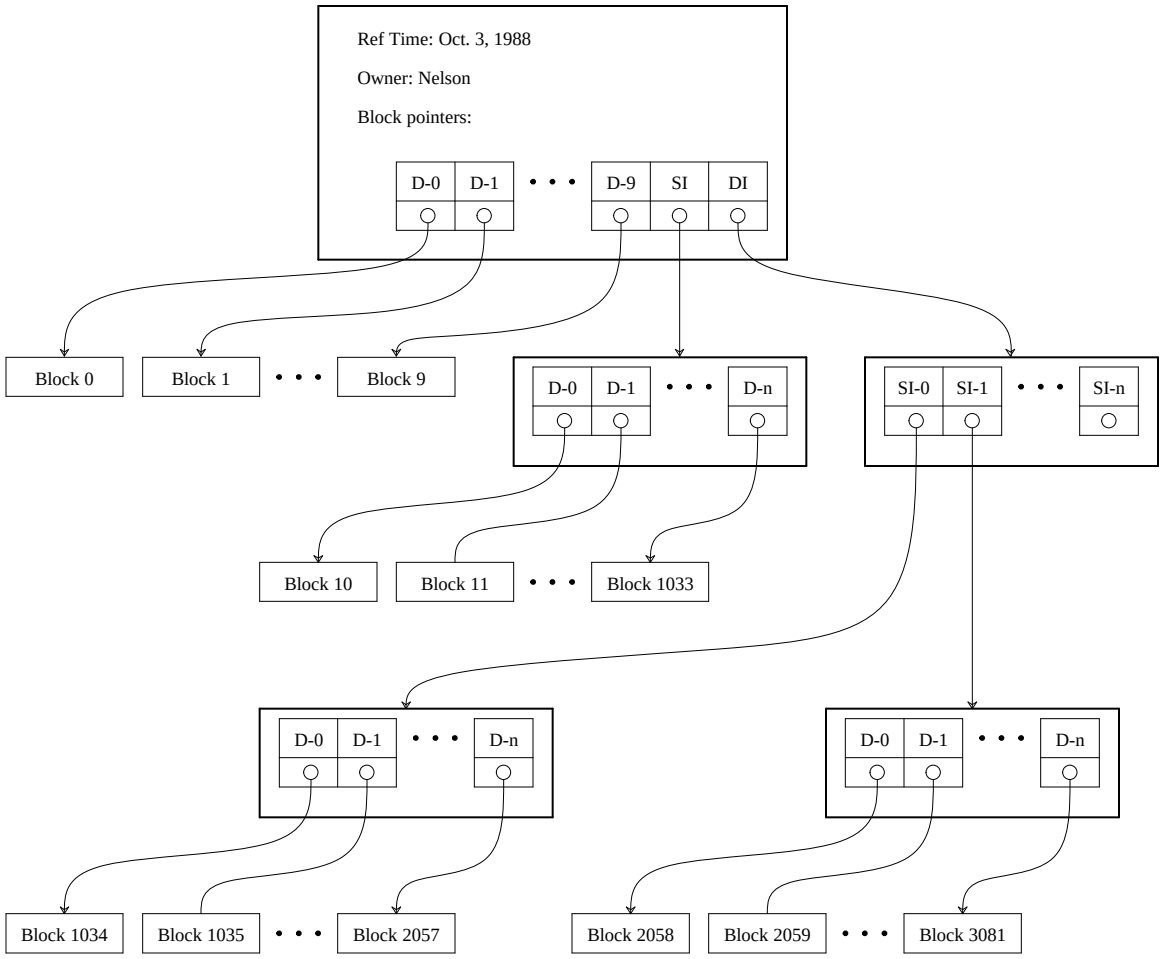

Figure 3-2. File disk structure. Among other attributes such as the reference time and the owner, a file descriptor contains the location of the data blocks on disk. Each descriptor contains 10 *direct block pointers*, 1 *singly-indirect block pointer* and 1 *doubly-indirect block pointer*. In this picture the direct block pointers are denoted D-0 through D-9 and they contain the disk addresses of blocks 0 through 9 in the file. The singly-indirect block pointer is denoted SI and it points to a block of 1024 direct block pointers; these pointers point to blocks 10 through 1033 in the file. The doubly-indirect block pointer is denoted DI and it points to a block with 1024 singly-indirect block pointers. The first singly-indirect block contains pointers to blocks 1034 through 2057 in the file, the second singly-indirect block points to blocks 2058 through 3081 and so on.

rest of the disk consists of data blocks and indirect blocks.

When a new block is allocated to a file, a data block and possibly an indirect block will have to be allocated. If a data block has no preceding block in the file, then a

random data block is chosen out of all available data blocks. Otherwise, a block that is nearest on the disk to the preceding block is chosen. This is done to reduce the number of seeks between reads and writes of successive data blocks. When an indirect block is allocated, a random block is chosen.

When a new file is created, a file descriptor must be allocated for the new file. If the file that is being created is a normal file, then Sprite attempts to allocate a file descriptor that is in the same or nearby file descriptor block as the file's directory. This allows the file descriptors for many files within a given directory to be read or written with only one disk operation. When a new directory is created it is put into a random descriptor block. This is done so that the directories will be randomly distributed amongst the file descriptors; otherwise all directories would end up fighting for file descriptors in the same file descriptor blocks.

There are two potential problems with the simple Sprite disk layout. First, when a block is allocated to a file, Sprite chooses the nearest block on disk even if the block is not rotationally optimal; the result is that, in general, Sprite is only able to transfer one block per disk revolution. Second, Sprite does not attempt to put either the file descriptor or the indirect blocks for a file near to the data blocks for the file. This is different from the UNIX 4.2 BSD implementation, which puts file descriptors, indirect blocks and data blocks for a file within the same group of cylinders on disk [MJL84]. The result is that Sprite may have to perform longer seeks between reads and writes of the three types of disk data. Because all three types of data are cached by Sprite, reading the data from disk should not be a problem. However, the disk layout does impact writing performance and will be discussed further in Chapter 5.

3.5. Details of the Implementation

3.5.1. Implementing Delayed-Write

The delayed-write policy used by Sprite provides good writing performance but it complicates the implementation of the file system in two ways. First, since the server is not contacted on every write of data, disk space cannot be allocated for newly written data blocks. This means that, when the client eventually writes the new block back to the server (as much as 35 seconds later) there may be no disk space available; what is even worse is that the user process that wrote the data to the cache may have exited with the belief that the data that it generated is safe. This is handled in Sprite in a simple manner: when it is detected on a delayed write that there is no disk space available, the user is informed of the situation (including the names of files that cannot be written back), and the delayed write will be tried again 30 seconds later. It is up to the user to free up enough space on disk to store the data that cannot be written back.

Another complication from the delayed-write scheme is that, for up to 35 seconds after new data is written, the client, not the server, will know the current modify time for the file and the current file size. Likewise, since reads do not go through to the server, the client will also know the current access time for the file. This presents a problem if a client other than the one with the most up-to-date attributes tries to get the attributes of a file. Since in Sprite all attempts to get the attributes of a file must go through to the server of the file, the server can keep the attributes consistent. If the server detects that it does not have the most recent attributes for a file, it will retrieve the attributes from the client that does have the most recent attributes. This call-back

mechanism is implemented in a similar way to that used for cache consistency explained above.

3.5.2. Providing Reliability

The design of the Sprite file system has emphasized performance, not reliability. We chose to use a 30-second delayed-write policy similar to the one that has been used successfully in many versions of UNIX for the past 15 years. The use of the 30-second delayed-write policy introduces the possibility of data getting lost on a system crash: up to 35 seconds of data on a client crash, and up to 70 seconds on a server crash. In order to reduce the likelihood of data getting lost during a crash, the Sprite caching code has been carefully written, so that, when a machine crashes, there is a high probability that it can write its cache back to the server or to disk. This is done by ensuring that the cache write-back code only relies on either the RPC system or the disk sub-system to be functional; both of these are very stable and have no known bugs.

Even though the cache can usually be written back on system crashes, there is still the possibility of lost data. In fact, because of the behavior of certain important programs that manage files (e.g., source code control systems and editors), much more serious damage can occur on a system crash. For example in the *mx* editor developed by John Ousterhout, whenever the file that is being edited is saved by the user, the editor truncates the file and rewrites it. The truncate command goes through to the file server so that disk space can be reclaimed, but the rewritten data does not for at least 30 seconds. As a result, on a system crash the entire contents of the file, including data that could have been written in days past, can be lost.

In order to provide higher reliability to those programs that require it (e.g., editors), the file system provides a function, callable by user programs, that forces a file to be synchronously flushed from the client's cache to the server's disk. This function only provides a partial solution to the reliability problem, because a crash could occur between a file truncation operation and a forced write-back operation; the truncation will delete the file data and the new data may be lost during the crash. A common solution used by many programs is to use temporary files and file move operations. A program that used this method would first write data to a temporary file, force the data to be written through to the server's disk, and then rename the temporary file so that it has the same name as the original file. In order for this to work safely, the file system provides an atomic file rename operation with the semantics that either the original copy of the file exists or it has been replaced by the new copy of the file.

The solutions that have been used in other file systems to provide a higher measure of reliability than Sprite's are based on file versions [CaW86, SGN85] or atomic transactions [BKT85, PoW85]. The systems that use file versions create a new version each time that a file is written. Thus, files will never be destroyed as a result of client or server crashes, because old versions of files will remain safely on disk. We chose not to use the version mechanism so that we could stay compatible with the standard UNIX paradigm for accessing files.

Transaction systems guarantee that, when a file is rewritten, either the new version of the file will exist or the old version will exist, but the file's original contents will not be lost. We did not implement transactions for two reasons. First, we did not feel that the application environment that we were targeting for required transactions. Second,

transactions are inherently complex and potentially have a negative impact on performance.

Although Sprite does not provide the same measure of reliability as some other systems, we are satisfied with its reliability. Data does still occasionally get lost during system crashes, but the system is becoming much more stable and, as a result, file data is rarely lost. We could have made the system more reliable by using transactions or file versions, but it would have resulted in a more complex and possibly less efficient implementation. The delayed-write policy used in Sprite is a compromise between reliability and performance: it gives the best performance while giving reliability that is quite acceptable in our environment.

3.5.3. Cache Consistency Implementation

Although the Sprite cache consistency mechanism is simple in principle, there are several complexities in its implementation. One such complexity is synchronizing access to the per-file cache state information. In order to allow the server to determine the consistency state for a file, the server maintains two pieces of state information for each file: a list of clients that are using the file and the client that was the last writer. The server does not need to maintain state information about clients that have closed a file and only have clean data in their cache; version number verification at file open time will keep these files consistent. Access to the consistency data structures must be serialized. For example, when a file is being opened, no other open of the file can occur until the file is brought to a consistent state, because another open could potentially change the cacheable state of the file.

In order to allow files to be safely brought to a consistent state the file system has two types of locks for each file. One lock is called the I/O lock and is used to ensure that only one read or write can occur to a file at one time; this lock is necessary to protect certain kernel data structures associated with each file. The other lock is called the consistency lock and is used to synchronize access to the cache consistency data structures. Two separate locks are required because the act of bringing a file to a consistent state may require that the server call back to clients to force them to write back their dirty data. Thus, while access to the cache consistency data structures for a file is being serialized, a write to the file must be able to occur.

Another complexity in the Sprite cache consistency mechanism is performing the client call-backs when the cacheable state of a file changes. Inherent in any network implementation is the possibility that messages may arrive out of order. One possible way that this can happen is when messages get lost and have to be resent. This message ordering problem adds the potential of a race condition to the Sprite cache consistency algorithm (see Figure 3-3). When the open of a file by a client completes, the server sends back a reply to the client that indicates whether the file is cacheable or not. Once the reply is sent, an open by another client can occur on the file. If the second open makes the file change from cacheable to non-cacheable then the server will send a message to the first client telling it not to cache the file after all. However, if the reply to the first open gets lost, then the server's message telling the client not to cache the file could be received before the reply from the open. Therefore if a client derives the cacheable state for a file from the most recent server message about the file, a client could erroneously believe that it can cache a file.

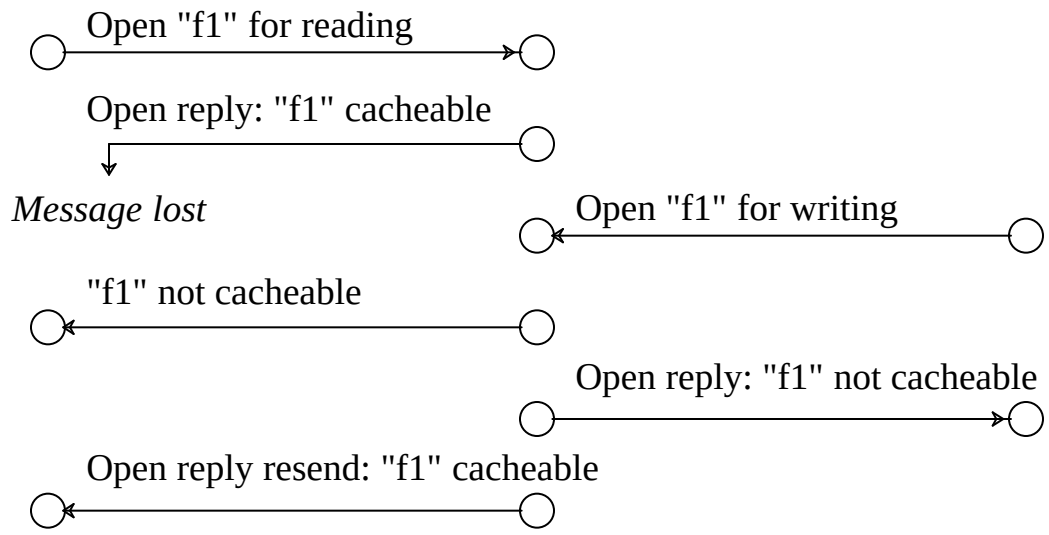

Figure 3-3. Open race condition. Client 1 opens file f1 for reading. The server sends a reply to the open which indicates that the client can cache the file. However, the reply gets lost. Before the server detects that the reply got lost, client 2 opens file f1 for writing. Since client 1 has the file open for reading, the server detects that concurrent write sharing is about to occur, tells client 1 that it can no longer cache the file, and replies to client 2. The server then resends the reply to the original open request made by client 1. If client 1 only pays attention to the last message from the server, then it will mistakenly think that it can cache file f1.

This race condition is solved by introducing open time stamps (see Figure 3-4). Each time that a client opens a file, the server stores the time when the open occurred with the client state information it keeps with each file. This time stamp is also sent back to the client with the open reply, and clients keep the most recent time stamp with each file. When a server sends a cache consistency message for a file to a client, it includes the time of the most recent open of the file by that client. There are three possibilities when a client receives a consistency message. The most likely possibility is that the client and server time stamps are equal. In this case the client will process the message and inform the server when it has finished taking the necessary cache

consistency actions. The second possibility is that the client's time stamp is greater than the server's time stamp. When this happens the client will drop the message because it realizes that the message pertains to an old open of the file.

The final possibility is that the client's time stamp for the file is less than the server's time stamp; this is the race condition that the time stamps were designed to solve. When this occurs, the client realizes that the server is referring to an open for which the client has yet to receive the reply. The client will force the server to resend the message in the hope that the open reply will come in before the server is able to resend the cache consistency message (see Figure 3-4). The reason why the client forces the server to resend rather than queue up the message was done for to reduce the amount of state information to be maintained by the client.

One final detail of the implementation is the management of the last writer information. Since Sprite uses a 30-second delayed-write policy, all of a file's blocks will be up-to-date in the server's cache within 35 seconds after the file is closed on the client. There is no reason to maintain the last writer information when there are no more dirty blocks in the last writer's cache. This state information is cleaned up by having the client inform the server when it no longer has dirty blocks for a file; this can happen either when the file is closed or when the last dirty block is written back. This is not only an optimization, but is also a necessity in order to allow client workstations to clean up state information for files that are no longer cached. If a client deletes the state information about a closed file, it will not be able to handle cache consistency messages for the file; it will not know if a cache consistency message is for an open that has not yet completed or for an open that happened before the file state information was

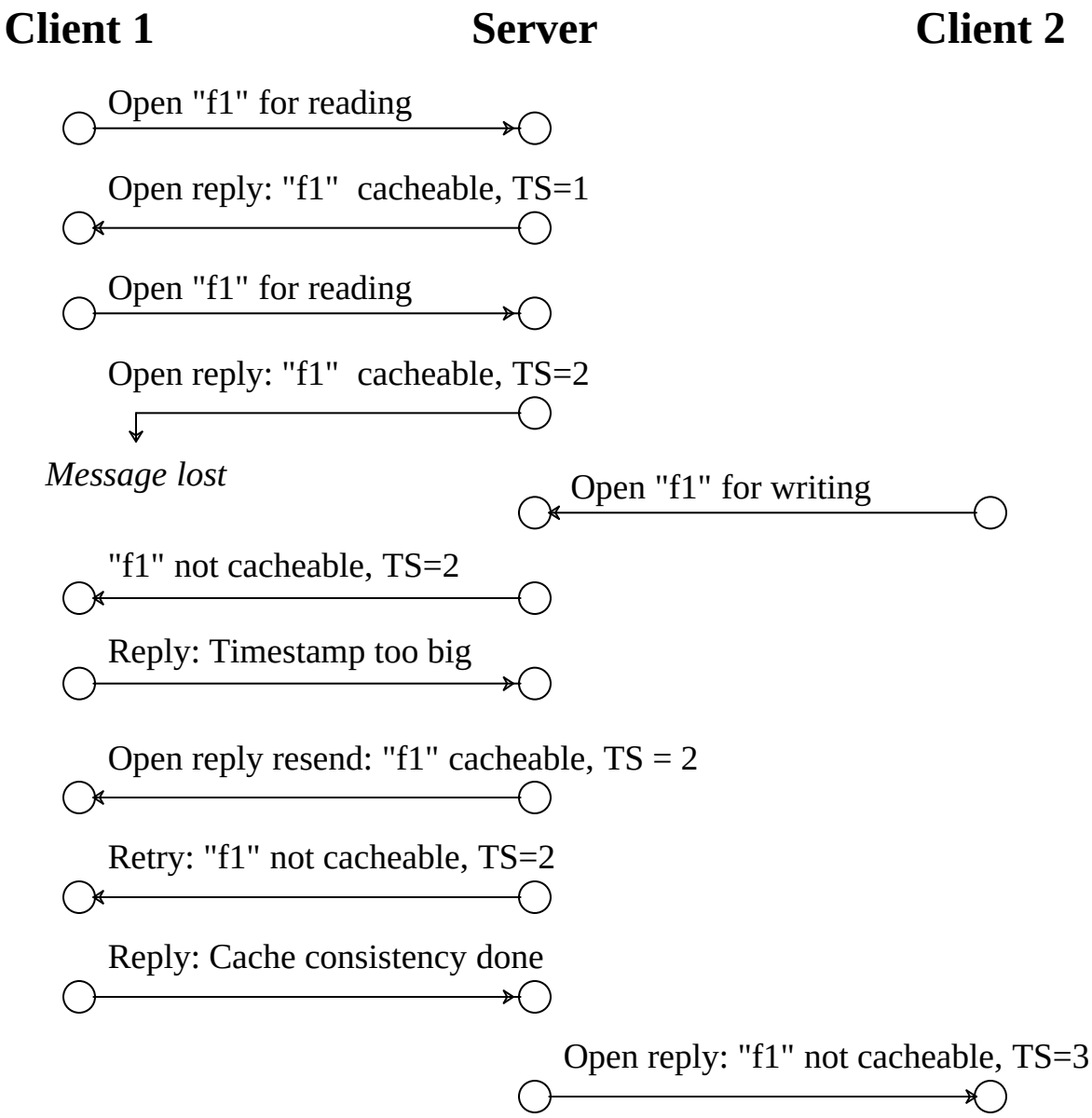

Figure 3-4. Solution to open race condition. The problem is solved with time stamps. Client 1 first opens file f1 for reading and gets back a time-stamp equal to 1. Client 1 then opens f1 again for reading, but this time the server's reply gets lost. Before the server detects that the reply got lost, Client 2 opens file f1 for writing. The server detects that concurrent write sharing is about to occur and sends a cache consistency message to Client 1. However, by comparing time stamps Client 1 determines that the server is referring to an open that the client has not got the reply for yet. As a result the client tells the server that the time stamp that it gave was too large and it should try again. Meanwhile the server resends the reply to the latest open for Client 1. The server then resends the cache consistency message. This time the client has the same time stamp as the server. Once the server gets the successful reply from Client 1 it replies to the open from Client 2.

cleaned up. Thus, the client must ensure that the server knows that the client no longer has dirty blocks for a file before it deletes important state information.

Unfortunately, there is a race condition when trying to detect that a client no longer has dirty blocks for a file. When a file is closed, the client must determine if it has dirty blocks for the file. If not, it includes with the close message an indication that it does not have any dirty blocks for the file. In addition, when a client writes back a dirty block (as part of a 30-second dealyed write) it must indicate to the server whether or not this is the last dirty block for the file. The race occurs between the delayed write-back and the close. Assume that when a file is closed there remains one dirty block. The client will inform the server in this case that it still has dirty blocks for the file. Now assume that, immediately after the close, the last block for the file is written back. On this operation, the client will inform the server that there are no more dirty blocks for the file. The problem occurs if the write-back message arrives before the close message. The server cannot believe the write-back message because it thinks that the file is still open on the client and that the client can still generate dirty blocks. However, if the server ignores the write-back message, then it will lose the fact that there are really no more dirty blocks for the client. This problem is solved by synchronizing delayed write-backs and closes: while a file is being written-back, the file cannot be closed and vice versa. This guarantees that the messages will arrive in the right order.

3.5.4. Crash Recovery

One of the disadvantages of the Sprite caching mechanism is that servers must maintain a large amount of state information in their main memories. This includes

both file data as well as information about which clients have open files. In order for clients to be allowed to continue after a server crashes and reboots, this state must be recoverable. In contrast, the servers in Sun's NFS are stateless. This results in less efficient operation (since all important information must continually 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.

Sprite's approach is to recover from the most common cases and be able to detect when uncommon, non-recoverable cases occur. The server's state information about open files is recovered with help from the clients. The Sprite RPC system allows clients to determine when a server crashes and when a server reboots. When a client detects a server crash, it delays write-backs of dirty blocks to the server until it detects a reboot. When the server reboots, the client attempts to reopen all of its files and then writes back any dirty blocks that need to be written back to the server.

In all but two cases, a client will be able to reopen its files and continue normally. The first case is a race condition between clients reopening files and clients opening files; in some cases a cache consistency violation may occur. For example, assume that client C1 is caching file F1 for writing when the server crashes. Now if, when the server reboots, client C1 is unable to reopen F1 before some other client opens F1, then a cache consistency violation will occur. If such a violation occurs, the reopen fails. The probability of these violations occurring is diminished by having servers give clients time to reopen their files before accepting new opens for files.

The second case where a client will not be able to reopen files is when the server lost dirty blocks that the client had written back. The current mechanism that is used to handle this case is to detect when the server is unable to write-back its data to disk on a crash. When the system reboots, if it was able to successfully write back its cache to disk when it crashed (the server marks its disk when it is able to successfully flush the cache), then clients are allowed to reopen files normally. Otherwise, all reopens for files on the disk are refused. As mentioned earlier, the file caching code is carefully written, so that, unless there is an error in the cache data structures or the disk subsystem, the server will be able to write its cache back to disk; based on current experience with the system, the server very rarely fails while trying to write its cache to disk after a crash.

The other option that can be used to allow the server to recover file data information after it reboots is to use a more secure writing policy. For example, if file servers used a write-through policy, then there would be no chance of data getting lost on a server crash. Chapter 5 looks into the performance impact of such a writing policy.

3.6. Summary

In this chapter I have presented the design of the Sprite file system. The file system has been designed for high performance and to maintain the ease of file sharing that was available in timesharing systems. In order to achieve this performance, Sprite provides caching on both client and server machines. A 30-second delayed-write policy is used on both client and server machines in order to get the best writing performance. The file system guarantees workstations a consistent view of the file data, 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.

One of the disadvantages of the Sprite approach is that it is not as reliable as many other systems because we set performance as our primary goal. This introduced a few potential reliability problems, which we are solving as we encounter them. I am confident in our ability to provide an acceptable level of reliability. Efficient methods of providing better reliability by allowing programs to force data onto the server's disk will be discussed in Chapter 5.

Although the file system must maintain state information in order to provide cache consistency, it is designed to gracefully recover from most client and server crashes. The recovery mechanism is designed so that full recovery is possible in the normal case, but certain rare cases may not be recoverable. The mechanism is simple, yet should work in most cases.

CHAPTER 4

File System Performance

4.1. Introduction

This chapter presents performance measurements of the benefits of client data caching. The measurements were made by running a series of file-intensive benchmark programs against the Sprite file system. The 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 sizes increase?

All of the 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 16 Mbytes of memory and a 400-Mbyte Fujitsu Eagle disk.

4.2. Micro-benchmarks

I 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 four common file lookup operations (see Table 4-1). The remote versions took 3-6 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, I rigged the system so that all the accesses would be satisfied in a particular place (e.g. the client's cache). Table 4-2 shows the I/O speeds achieved to and from caches and disks in different locations.

Table 4-2 contains two important results. First, a client can access bytes in its own cache 7-8 times faster than those in the server's cache. This means that, in the best

File Lookup Operations			
Operation	Local Disk	Diskless	
		Elapsed Time	Server CPU Time
Open/Close	3.30ms	10.06ms	5.34ms
Failed Open	1.30ms	4.15ms	2.08ms
Get Attributes	1.10ms	4.32ms	2.21ms
Get Attributes ID	0.54ms	3.63ms	1.71ms

Table 4-1. Cost of four common file lookup operations in Sprite. Each of these operations requires contacting the server of the given 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, the third row is the cost of getting the attributes of a file (''stat''), and the fourth row is the cost of getting the attributes of a file that is already open.

case, client caching could permit an application program to run as much as 7-8 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 the 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's, or about 2-3 times the rates listed in Table 4-2; 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 do not, the server's cache should become much faster than a local disk, up to the limits of network bandwidth. For example, if the clients and servers were 8-MIPS Sun-4s instead of 2-MIPS Sun-3s, then a client should be able to read the server's cache up to 4 times as fast as a local disk. Even for paging traffic, this suggests that a large server cache may provide better performance than a local disk.

4.3. Macro-benchmarks

The micro-benchmarks discussed in the previous section 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, I ported several well-known programs

Read & Write Throughput, Kbytes/second				
	Local Cache	Server Cache	Local Disk	Server Disk
Read	3357	470	222	207
Write	2786	368	200	178

Table 4-2. Maximum rates at which programs can read and write file data in various places, using large files accessed sequentially.

from UNIX to Sprite and measured them under varying conditions. I ran each benchmark three times for each data point measured and took the average of the three runs. Table 4-3 describes the benchmark programs. See Appendix A for detailed tables with the results of running the 5 benchmarks including standard deviations.

4.3.1. Application Speedups

Table 4-4 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 about 10-20% slower than those with disks; one benchmark, Diff, was actually 85% slower on diskless machines than on machines 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 8%. Figure 4-1(a) shows how the performance varied with the size of the client cache.

4.3.1.1. Server Load

One of the most beneficial effects of client caching is its reduction in the load placed on server CPUs. Figure 4-2 shows the server CPU utilization with and without client caching. In general, a diskless client without a client cache utilized about 5-27% of the server's CPU. With client caching, the server utilization dropped by a factor of 1.5 or more, to 1.5-12%.

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 [How88] for details.	58.0	36.5
Vm-make	Use the ‘‘make’’ program to recompile the Sprite virtual memory system: 14 source files, 12600 lines of C source code.	42.3	25.9
Sort	Sort a 1-Mbyte file.	46.4	89.9
Diff	Compare 2 identical 1-Mbyte files.	452.2	4.3
Ditroff	Format a paper which contains both figures and tables. The input file contains 56 Kbytes of data.	7.0	10.4

Table 4-3. 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	265 104%	255 100%	321 126%	307 120%	288 113%	275 108%
Vm-make	284 103%	277 100%	337 122%	330 119%	305 110%	296 107%
Sort	64 107%	60 100%	75 125%	71 118%	65 108%	59 98%
Diff	21 457%	4.6 100%	25 543%	8.5 185%	25 543%	4.5 98%
Ditroff	128 102%	125 100%	133 106%	131 105%	128 102%	125 100%

Table 4-4. 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 described in Chapter 6.

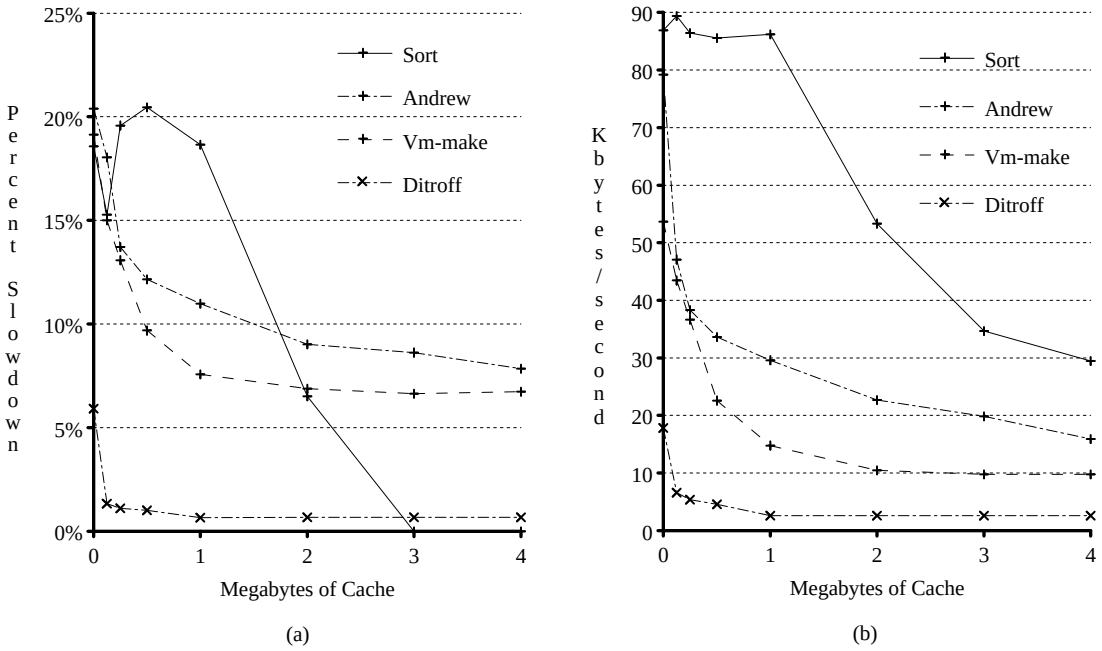

Figure 4-1. Client degradation and network traffic (diskless Sun-3's with client caches, warm start) as a function of maximum client cache size. For each point, the maximum size of the client cache was limited to a particular value. The ''degradation'' shown in (a) is relative to the time required to execute the benchmark with a local disk and a 4-Mbyte warm cache. The diff benchmark did not fit on graph (a); for all cache sizes less than 2 Mbytes it has a degradation of 85% and for all larger cache sizes it has no degradation. 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) either; for all cache sizes less than 2 Mbytes it has an I/O rate of 260 Kbytes/second and for all larger cache sizes it has an I/O rate of only 1.3 Kbytes/second.

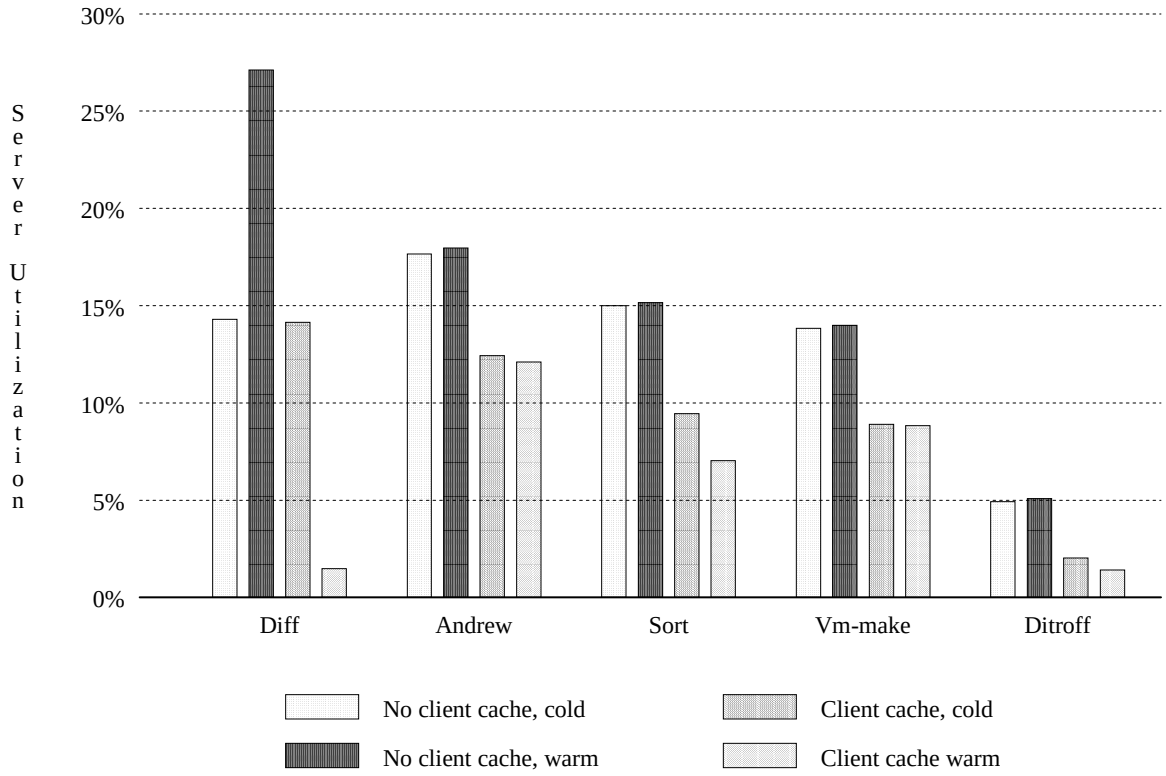

Figure 4-2. Client caching reduces server loading by at least a factor of 1.5-3 (measured on Sun-3's with variable-size client caches).

4.3.1.2. 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 [LZC86]. However, as CPU speeds increase, the network bandwidth is becoming more and more of an issue. Figure 4-1(b) plots network traffic as a function of cache size for the benchmarks running on Sun-3's. Without client caching the benchmarks averaged 7.8% utilization of the 10-Mbit/second Ethernet. The most intensive application, diff, used 20% of the network bandwidth for a single client; the other 4 benchmarks averaged 4.65% of the 10-Mbit/second Ethernet. Machines at least five times faster than Sun-3's are already available (e.g., Sun-4 workstations); a single one of these machines

would utilize 25-100% 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 4-1(b) shows that client caching reduces network utilization by a factor of 4-10, to an average of about 0.66% for the benchmarks. The most I/O-intensive benchmark, Sort uses only 2.6% of the ethernet bandwidth. This suggests that 10-Mbit Ethernets will be adequate for the new 10-MIPS generation of CPUs, and perhaps one more generation to follow. After that, higher-performance networks will become essential.

Ricardo Gusella in an analysis of diskless workstation Ethernet traffic also noticed that Ethernets are becoming heavily loaded with the introduction of faster machines [Gus87]. He measured the traffic on a 10-Mbit Ethernet over a 24 hour period. He determined that two Sun-3 workstations (a Sun-3/180 server and a Sun-3/50 client each with 4 Mbytes of memory) running UNIX with Sun's Network File System (NFS) [San85] can utilize over 20% of the Ethernet. Since the workstations that Gusella measured had smaller memories than the Sprite workstations and NFS does not utilize file data caches as effectively as Sprite, I would not expect Sprite to exhibit the same loads that were measured by Gusella. However, Gusella's measurements are another indication that higher-performance networks will be necessary in the near future.

4.3.1.3. Disk Utilization

Figure 4-3 shows the disk utilizations of the benchmarks. For most of the benchmarks, the disk utilization with a warm cache is less than 6% with or without client caching. This shows that, for most of the benchmarks, a cache on the server is able to reduce the disk traffic to reasonable levels.

Sort is the one benchmark that has a fairly high disk utilization without client caching; with client caching the disk utilization is cut in half. This demonstrates the advantage of the 30-second delayed write policy. The Sort benchmark completes in

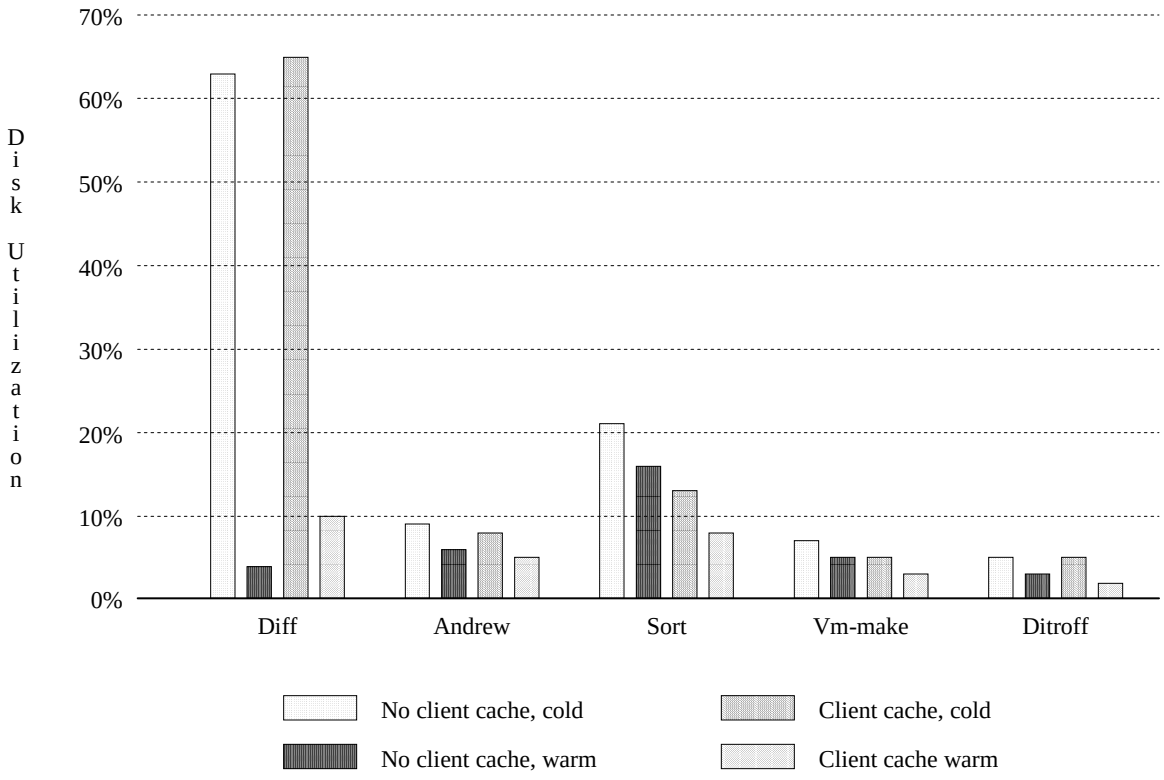

Figure 4-3. Client caching reduces disk utilization by up to a factor of 2 (measured on Sun-3's with variable-size client caches).

around 60 seconds. When client caching is used, all writes to disk will be delayed by 30 seconds on the client and 30 seconds on the server. Thus, only the final result will end up getting written to disk. Without client caching modified data will only be delayed by 30 seconds; any intermediate files that live longer than 30 seconds will get written through to disk. If the server were changed to use a 60 second delayed-write policy, then many of the extra disk writes without client caching would be eliminated.

With warm caches the disk utilization of these benchmarks is up to a factor of two lower than the CPU utilization. The disk utilization would be even lower if Sprite did a better job of utilizing the disk bandwidth; currently only one block can be transferred per disk revolution. Therefore, currently the CPU should saturate before the disk. However, as CPUs get much faster and disks do not, the disk may become the bottleneck that will limit system scalability.

4.3.1.4. Contention

In order to measure the effects of loading on the performance of the Sprite file system, I 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, so there was no cache consistency overhead. I ran each contention benchmark three times for each data point measured and took the average of the three runs.

Table 4-5 and Figure 4-4 show 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 five clients and no client caching, the clients were executing 80% more slowly, the server was

Andrew Contention Results										
Number of Clients	Elapsed Time		Network Mbytes		Server Util		Disk I/Os		Disk Util	
	No Client Cache	With Client Cache	No Client Cache	With Client Cache	No Client Cache	With Client Cache	No Client Cache	With Client Cache	No Client Cache	With Client Cache
1	307 6.1	275 0.0	23.8 0.0	4.3 0.0	18.0% 0.1	12.1% 0.0	863 359.8	647 1.7	6.0% 1.7	5.0% 0.0
2	324 2.6	275 0.4	47.7 0.6	8.6 0.0	34.6% 0.1	21.8% 0.1	1397 190.0	1141 2.1	12.0% 1.7	11.3% 0.6
3	353 3.5	286 1.7	71.7 0.6	12.9 0.0	48.1% 0.5	31.2% 0.1	2401 192.5	1644 12.5	19.3% 0.6	15.3% 0.6
5	450 2.3	321 9.7	120.3 0.5	21.6 0.0	65.7% 0.1	45.0% 0.4	4369 92.6	2742 28.5	30.2% 0.4	23.3% 1.2
7	519 22.2	372 8.3	168.8 0.6	30.2 0.0	74.3% 2.3	58.2% 0.3	6146 407.6	3843 48.4	38.0% 2.0	30.7% 0.6
10	753 3.3	456 15.3	245.9 0.6	44.0 0.6	83.3% 0.3	70.8% 0.1	9935 64.1	5659 234.9	50.7% 0.6	42.7% 2.1

Table 4-5. Andrew contention results. Each row contains two lines of data. The first line contains the results of running the benchmarks and the second line contains the standard deviations. Each row of the table is for a different number of clients running the Andrew benchmark at the same time against a single server. Each of the five columns of results are divided into the result when the benchmark was run without client caching and the result with client caching. The five columns show, in the following order, average elapsed time to execute the benchmark in seconds, network bytes transferred in megabytes, server utilization, number of disk reads and writes and disk utilization.

nearly 70% utilized, the network was over 20% utilized, and the disk was 30% utilized. With client caching and five active clients, each ran at a speed within 25% of what it could have achieved with a local disk; server utilization in this situation was about 45%, network utilization was only 5% and disk utilization was 23%. Basically, client caching allows servers to support twice as many clients and networks to support at least 4 times as many clients.

The measurements of Figure 4-4 suggest that client caches allow a single Sun-3 server to support 5-7 Sun-3 clients simultaneously running the Andrew benchmark. However, typical users spend only a small fraction of their time running such intensive programs. I estimate that one instance of the Andrew benchmark corresponds to about

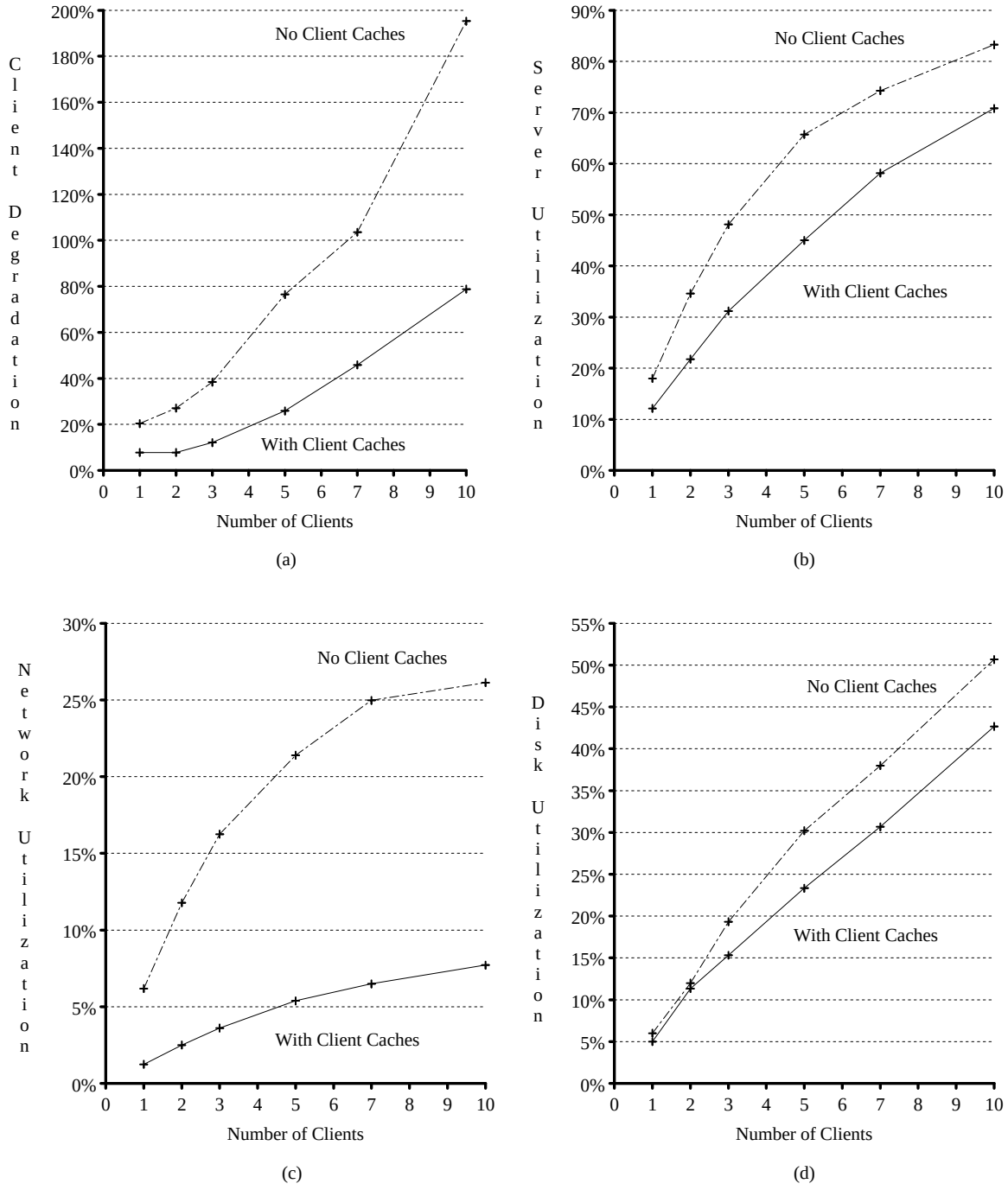

Figure 4-4. Effect of multiple diskless clients running the Andrew benchmark simultaneously on different files in a Sun-3 configuration with variable-size client caches. (a) shows additional time required by each diskless client to complete the benchmark, relative to a single client running with local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

5-20 active users, so that one Sun-3 Sprite file server should be able to support at least 30 Sun-3 users. This estimate is based on the study of UNIX done by Ousterhout *et al.* [Ous85], which reported average file I/O rates per active user of 0.5-1.8 Kbytes/second. I 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 Ousterhout's study did not include paging traffic and was based on slower machines used in a timesharing mode (I 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 [How88], and by Lazowska *et al.*, who estimated about 4 Kbytes/second of I/O per user on slower Sun-2 workstations [LZC86].

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.

4.4. Advantage of Local Name Caching

Although I am generally satisfied with Sprite's performance and scalability, I have estimated how much improvement would be possible if we implemented client-level name caching with an Andrew-like callback mechanism. Table 4-6 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 a factor of 2. This could potentially allow a single server or network to support up to twice the number of clients that the current implementation supports. Thus, in terms of CPU utilization, client name caching would provide about the same improvement as client data caching.

My estimate for improvement in Sprite is much smaller than the measured improvement in Andrew when they switched to callback. I 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, I suspect that there would have been less motivation to switch to a callback approach.

Benchmark	Degradation		Server Utilization		Network Utilization	
	Original	Handle Locally	Original	Handle Locally	Original	Handle Locally
Andrew	7.8%	0.0%	12.1%	6.3%	1.24%	0.67%
Vm-make	6.7%	0.5%	6.7%	4.7%	0.76%	0.35%

Table 4-6. 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.

4.5. Comparison to Other Systems

Figure 4-5 compares Sprite to the Andrew and NFS filesystems using the Andrew benchmark. The measurements for the NFS and Andrew file systems were obtained from [How88]. Unfortunately, the measurements in [How88] were taken using Sun-3/50 clients, whereas I 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, I re-normalized the Sprite numbers for Sun-3/50 clients: the Sprite elapsed times from Table 4-5 were multiplied by 1.3, and the server utilizations from Table 4-5 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 my measurements and the ones in [How88] 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 4-5 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, I 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

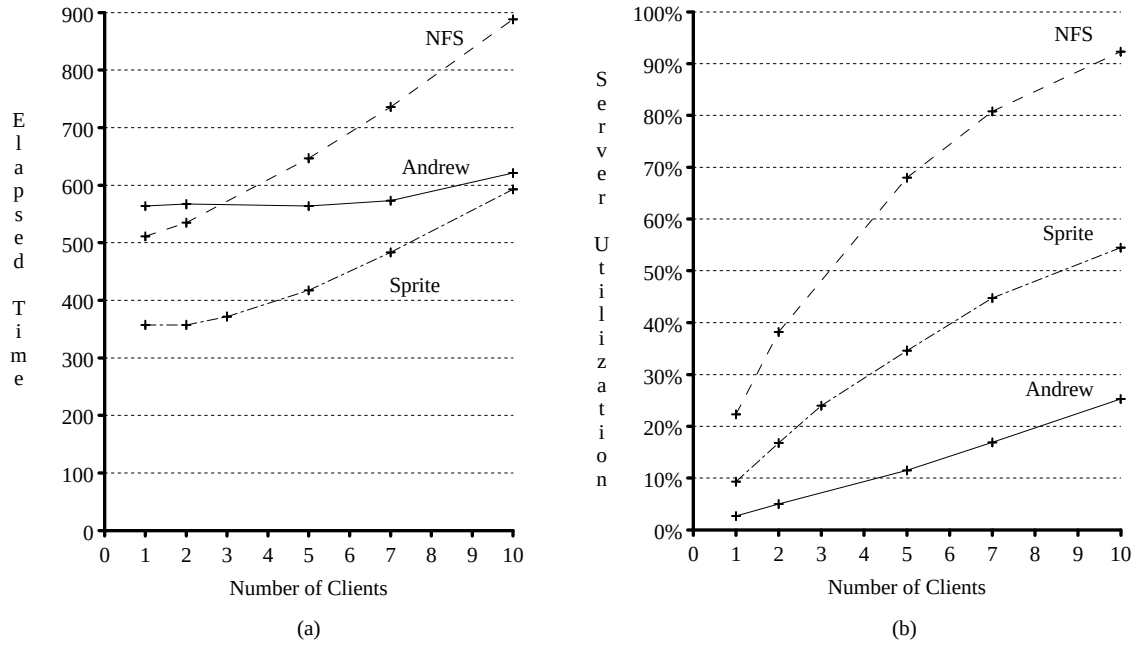

Figure 4-5. Performance of the Andrew benchmark on three different file systems; Sprite, Andrew, and NFS. (a) shows the absolute running time of the benchmark as a function of the number of clients executing the benchmark simultaneously, and (b) shows the server CPU utilization as a function of number of clients. The Andrew and NFS numbers were taken from [How88] and are based Sun-3/50 clients. The Sprite numbers were taken from Table 4-5 and re-normalized for Sun-3/50 clients.

2.4% server CPU utilization, vs. 7.5% in Sprite and 20% in NFS. I attribute Andrew's low server CPU utilization to its use of callbacks. Figure 4-5 reinforces my claim that a Sprite file server should be able to support at least 30 clients: our experience with NFS is that a single server can support 10-15 clients, and Sprite's server utilization is only one-third that of NFS.

4.6. Summary

In this chapter I presented detailed measurements of the performance of client caching. On average, client caching resulted in a speedup of about 10-20% for programs running on diskless workstations, relative to diskless workstations without client

caches. With client caching enabled, diskless workstations completed the benchmarks only 0-8% more slowly than workstations with disks. Client caches reduced the server utilization from about 5-27% per active client to about 1-12% per active client. Since normal users are rarely active, my measurements suggest that a single server should be able to support at least 30 clients.

In addition to measuring the absolute performance of Sprite, I also compared the performance of the Sprite file system, the Andrew file system [Sat85], and Sun's Network File System [San85] for a particular file-intensive benchmark. I showed that Sprite completes the benchmark 30-35% faster than the other systems. Sprite's server utilization was one-third of NFS's utilization but three times Andrew's utilization.

CHAPTER 5

Writing Policies

5.1. Introduction

The policy that is used to handle data after it has been written impacts performance, reliability and the cache consistency mechanism; the writing policies on both the server and the client are important. In the implementation of Sprite that I described in the previous chapters the 30-second delayed write policy was used on both clients and servers. This allows Sprite to attain high performance, but it potentially reduces its reliability and complicates its cache consistency mechanism. This chapter focuses on the performance-reliability tradeoff: are there writing policies that provide both high performance and high reliability?

All of the previous work on the impact of the writing policy has been done by using traces of UNIX timesharing systems. In addition, there have been no analyses of the trade-offs between the client writing policy and the server policy; previous work has concentrated on analyzing either the server policy or the client writing policy, but not both. In this chapter, I will explore the impact of the writing policy by measuring the results of running benchmarks against the Sprite file system. This will include an analysis of numerous writing policies on clients and several policies for the server. The measurements will answer the following questions:

- What is the impact of the client writing policy on client performance and the amount of network traffic?
- What effect does the server writing policy have on the amount of disk traffic, on the utilization of the server's CPU, and on the performance of client workstations?
- Does the impact of the server policy differ depending on which policy the client uses?
- Does the impact of the client policy differ depending on which policy the server uses?

The client writing policies that I will analyze are shown in Table 5-1. These include the policies used by all file systems I know, and they cover the whole range of the performance-reliability tradeoff: from write-through to delayed-write. In addition, policies that treat temporary files specially are included, in order to determine whether delaying temporary files will allow higher reliability for most files while still providing good performance.

The server policies are shown in Table 5-2. Delay-30 and write-through (WT) are the ones that are most commonly implemented on currently existing file servers. The other two policies, as-soon-as-possible (ASAP) and last-dirty-block (LDB), have been included as alternatives that provide higher reliability than delay-30 but with higher performance than WT. In particular, with the cooperation of clients, the LDB policy can provide nearly the same reliability as WT. If clients do not remove any blocks from their cache until the last dirty block for a file has been written back to the server, then the LDB and WT policies will provide nearly the same server reliability.

Policy	Description
Write-through (WT)	A write call does not return until the data has been written to the server's cache.
Write-back-on-close (WBOC)	Write calls return as soon as data has been written to the client's cache, but a close call will not return until all of the modified data has been written to the server's cache.
As-soon-as-possible (ASAP)	Write calls return as soon as data has been written to the client's cache but the data is scheduled to be written back to the server's cache when either a block is full or the file is closed.
WBOC + ASAP	Combination of write-back-on-close and as-soon-as-possible.
Full Delay (full-delay)	Write calls return as soon as data has been written to the client's cache and blocks are not written back unless they are ejected from the cache.
30 Second Delay (delay-30)	Like full-delay except that every 5 seconds the cache is scanned and dirty blocks that have not been modified in at least 30 seconds are written back.
WT + delay /tmp files (WT-TMP)	Use full-delay policy for all files in the /tmp directory and write-through for all other files.
WBOC + TMP (WBOC-TMP)	Use full delay policy for all files in the /tmp directory and WBOC for all other files.
ASAP + TMP (ASAP-TMP)	Use full delay policy for all files in the /tmp directory and ASAP for all other files.

Table 5-1. Client writing policies. Each of these policies represents the action that the file system takes when an application program issues a write system call or a close call.

Policy	Description
30 Second Delay (delay-30)	The client RPC returns immediately after the data has been loaded into the cache. Data blocks, indirect blocks and file descriptors are not written back until either they are ejected from the cache or they are dirty and they have not been modified for at least 30 seconds.
Write-through (WT)	The client RPC does not return until the data, any modified indirect blocks, and the file descriptor have been written to the server's disk.
As-soon-as-possible (ASAP)	The client RPC returns immediately after the data has been loaded into the cache but the data, the file descriptor and dirty indirect blocks are all scheduled to be written to disk as soon as possible.
Last Dirty Block (LDB)	All client writes return immediately except for the one that contains the last dirty block for the file. The write that contains the file's last dirty block will not return until all dirty data blocks, dirty indirect blocks and the file descriptor have been written to disk. This policy is used in conjunction with the delay-30 policy on the server.

Table 5-2. Server writing policies. Each of these policies represents the action that the file system takes when a client delivers dirty data to the server via a remote procedure call (RPC).

Table 5-3 describes the benchmarks that I used to measure the impact of the writing policy. These benchmarks were chosen because, of all of the benchmarks given in

the previous chapter, they are the only ones that generate a large amount of write traffic. All of the measurements were made on configurations of Sun-3 workstations. The client was a Sun-3/75 with 16 Mbytes of memory and the server was a Sun-3/180 with 16 Mbytes of memory and 400-Mbyte Fujitsu Eagle disk. Both the client and server caches were 8 Mbytes, which were large enough to hold the entire input and output of each benchmark; thus, blocks were never written back during the execution of the benchmark unless the writing policy explicitly forced the block to be written back. However, both the client and server caches were written back at the end of each benchmark. This was done to capture the number of useful bytes of data generated by the benchmarks, but was not included in the measured elapsed time.

The rest of this chapter is organized as follows: Section 5.2 measures the impact of the client writing policy on network load; Sections 5.3, 5.4, and 5.5 analyze the impact of the server and client writing policies on disk traffic, server utilization and client performance, respectively.

5.2. Network Load

The load placed on the network by clients depends only on the client writing policy; the policy used on the server does not matter. Table 5-4 gives the number of Kbytes placed on the network by each of the three benchmarks for each of the 9 client writing policies. As expected, the full-delay policy gives the lowest network load for all benchmarks. Figure 5-1 shows the relative differences between the full-delay policy and the other policies.

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 [How88] for details.	58.0	36.5
Vm-make	Use the ‘‘make’’ program to recompile the Sprite virtual memory system: 15 source files, 11,250 lines of C source code.	42.3	25.9
Sort	Sort a 1-Mbyte file.	46.4	89.9

Table 5-3. 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 no disks and warm caches and the highest performance writing policy; they measure the benchmark’s I/O intensity.

Network Kbytes vs. Client Policy									
Client Policy	Andrew			Vm-make			Sort		
	Read	Write	Total	Read	Write	Total	Read	Write	Total
WT	1413 1.07	4853 1.62	6266 1.45	965 1.06	3676 1.87	4641 1.61	114 1.46	2989 2.63	3103 2.56
WBOC	1348 1.02	4783 1.60	6131 1.42	935 1.03	3652 1.86	4588 1.60	114 1.46	2989 2.63	3103 2.56
ASAP	1360 1.03	4933 1.65	6293 1.46	951 1.05	3831 1.95	4781 1.66	114 1.46	2989 2.63	3103 2.56
WBOC-ASAP	1361 1.03	4960 1.66	6321 1.46	951 1.05	3837 1.95	4788 1.66	114 1.46	2989 2.63	3103 2.56
delay-30	1323 1.00	3063 1.02	4386 1.02	909 1.00	1976 1.00	2885 1.00	88 1.12	1637 1.44	1725 1.42
full-delay	1321 1.00	2994 1.00	4315 1.00	908 1.00	1968 1.00	2876 1.00	78 1.00	1135 1.00	1213 1.00
WT-TMP	1387 1.05	3516 1.17	4903 1.14	938 1.03	2018 1.03	2956 1.03	77 0.99	1133 1.00	1211 1.00
WBOC-TMP	1330 1.01	3457 1.15	4787 1.11	908 1.00	1989 1.01	2897 1.01	78 0.99	1133 1.00	1211 1.00
ASAP-TMP	1342 1.02	3621 1.21	4962 1.15	924 1.02	2174 1.10	3098 1.08	77 0.99	1133 1.00	1210 1.00

Table 5-4. Network Kbytes vs. Client Policy. This table shows the amount of network traffic for each of the three benchmarks with each of the different client writing policies. The top line for each policy is the raw number of Kbytes transferred. The bottom line is the bytes transferred for the policy divided by the number of bytes transferred with the full-delay policy. Note that the reason why there are bytes transferred with the full-delay policy is that the client cache is written back at the end of each benchmark.

Figure 5-1 shows that all four of the non-delay policies require significantly more bytes transferred (between 40 and 150 percent) than the full-delay policy. Which non-delayed-write policy is used does not matter because each of the 4 policies transfers approximately the same number of bytes. Since the ASAP, WBOC and WBOC-ASAP policies only transfer file system blocks after they have either filled with data or the file is closed, these policies should transfer the same number of bytes. However, the WT policy sends data to the server as soon as it is written into the client's cache; if the benchmarks write data in pieces smaller than the file system block size, the WT policy will require more network transfers than the other policies. Fortunately, the packet

header overhead per network write is very small (less than 80 bytes). Since the benchmarks rarely write less than 1024 bytes to the cache at a time, the packet overhead is less than 8 percent. Thus, the variations in network bytes transferred between the 4 non-delayed-write policies should be very small.

It is interesting to note that the ASAP and WBOC-ASAP client policies actually transfer more bytes than either WBOC or WT for both the Andrew and Vm-make benchmarks. I believe that this may be because of packet retransmissions by the RPC system. As I explained in Chapter 4, there are several block cleaners in the kernel, but there is only one block cleaner writing back a file at any one time. Since each of the

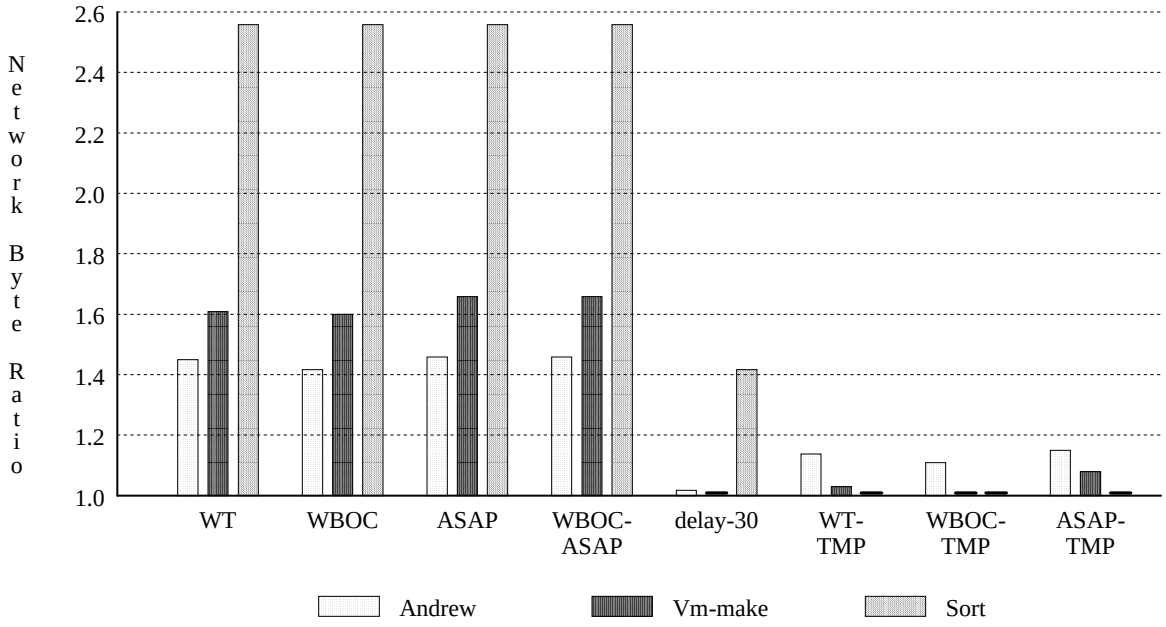

Figure 5-1. Ratio of number of network bytes transferred with each of the client policies to the number of bytes transferred with a full-delay policy. A ratio of 1.0 corresponds to the given client policy transferring the same number of bytes as the full-delay policy.

benchmarks only has one process executing at a time, they can only be writing and closing one file at a time. This means that, when the WT and WBOC policies are used, there can only be one file being written to the server at any given time. The ASAP policies on the other hand do asynchronous write-backs and can therefore have multiple files being written back at once. For this reason the ASAP policies will be interacting more intensely with the server; this may result in contention on the server, which may cause the RPC system to time-out and retransmit packets.

The delay-30 policy eliminates nearly all of the extra network writes caused by the 4 non-delayed-write policies. For the Andrew and Vm-make benchmarks, the delay-30 policy requires only 2% more bytes transferred than full-delay; this implies that nearly all of the temporary files that were used during these two benchmarks were deleted within 30 seconds of their creation. For the Sort benchmark the delay-30 policy requires 40% more bytes transferred than full-delay, because some of the temporary files do in fact live longer than 30 seconds. However, delay-30 still only transfers 2/3 of the bytes transferred by the 4 non-delayed-write policies. Thus, the delay-30 policy gives higher reliability than full-delay while requiring little overhead in terms of network bytes transferred.

The use of a full-delay policy on temporary files also is very effective in eliminating the extra network writes caused by the 4 non-delayed-write policies. This is especially true of the Sort benchmark, which writes all of its data to temporary files except for the final result; delaying temporary files eliminates all network transfers except for the ones required to write back the final result.

5.3. Disk Traffic

The amount of traffic to disk required to execute client programs limits the number of clients that can be supported by a single disk. With large caches on client and server workstations, the amount of data that is read from disks should be small; most of the traffic to disks will be data that is written. This can be seen by looking at the high ratio of network bytes written to network bytes read in Table 5-4. The writing policy will therefore determine the load that is placed on each disk, and hence the number of clients that a disk can support.

There are three types of file blocks that a server must write to disk: data blocks, indirect blocks and file descriptor blocks. As explained in Section 3.4, the data blocks contain the actual data that is written by the client to the server's cache, and the file descriptor and indirect blocks describe where the data blocks reside on disk. In order to make the write of a new data block reliable, the data block, the file descriptor block, and possibly an indirect block need to be written to disk; an indirect block only has to be written if the data block is not one of the first 10 blocks in the file. If a client is writing to a file block that has already been reliably written to disk, then it is not necessary to write the descriptor and indirect blocks through to disk.

The server policy that requires the smallest amount of disk traffic is delay-30 (see Table 5-5). With this policy, not only are data blocks only written back after they have been in the cache for 30 or more seconds, but the same policy applies to directories and indirect blocks as well; file descriptor block write-backs are also delayed, but only for 5 seconds. Since each file descriptor block can hold descriptors for 32 files, delaying

Disk Traffic: 30-Second Delay on Server												
Client Policy	Andrew				Vm-make				Sort			
	Disk Util	Disk Writes			Disk Util	Disk Writes			Disk Util	Disk Writes		
		Data	Ind/ Desc	Total		Data	Ind/ Desc	Total		Data	Ind/ Desc	Total
WT	6%	498	154	652	3%	261	186	447	16%	490	37	528
	1.00	1.03	0.89	0.99	1.00	1.04	1.09	1.06	2.00	1.74	1.37	1.71
WBOC	5%	483	149	632	3%	249	176	425	13%	405	34	439
	0.83	1.00	0.86	0.96	1.00	1.00	1.04	1.01	1.63	1.44	1.26	1.42
ASAP	6%	496	154	650	3%	250	176	426	14%	420	34	454
	1.00	1.02	0.89	0.99	1.00	1.00	1.04	1.01	1.75	1.49	1.26	1.47
WBOC-ASAP	6%	499	145	644	3%	249	177	426	14%	420	33	454
	1.00	1.03	0.84	0.98	1.00	1.00	1.04	1.01	1.75	1.49	1.22	1.47
delay-30	6%	484	156	641	3%	252	171	423	8%	282	27	309
	1.00	1.00	0.90	0.98	1.00	1.01	1.01	1.01	1.00	1.00	1.00	1.00
full-delay	6%	484	173	657	3%	250	170	420	8%	282	27	309
	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
WT-TMP	6%	485	152	637	3%	251	180	431	8%	282	28	310
	1.00	1.00	0.88	0.97	1.00	1.00	1.06	1.03	1.00	1.00	1.04	1.00
WBOC-TMP	5%	482	149	631	3%	249	177	426	8%	282	27	309
	0.83	1.00	0.86	0.96	1.00	1.00	1.04	1.01	1.00	1.00	1.00	1.00
ASAP-TMP	5%	483	145	628	3%	249	177	427	9%	282	27	309
	0.83	1.00	0.84	0.96	1.00	1.00	1.04	1.02	1.13	1.00	1.00	1.00

Table 5-5. The amount of disk traffic with the delay-30 policy on the server. The top line for each client policy is the disk utilization and the number of disk writes for the three benchmarks. The bottom line is the top line divided by the disk utilization or number of disk writes with the full-delay policy on the client and the delay-30 policy on the server. Note that the numbers in the bottom line should never be lower than 1.0 since the full-delay policy should give the smallest number of disk writes. However, because of the small number of disk transfers and slight variabilities

the write of descriptor blocks to disk may allow descriptor information for several files to be written to disk at once. This can be very useful because, as explained in Section 3.4, Sprite attempts to put file descriptor information for files that reside in the same directory in the same or nearby file descriptor blocks; thus, the file descriptors for many files within a given directory can be written to disk with only one disk write.

Table 5-5 shows that, with the delay-30 server policy, the client policy has very little impact on the amount of disk traffic for either the Andrew or Vm-make benchmarks. For these two benchmarks, although more reliable client policies require more

Disk Traffic: Write-Through on Server												
Client Policy	Andrew				Vm-make				Sort			
	Disk Util	Disk Writes			Disk Util	Disk Writes			Disk Util	Disk Writes		
		Data	Ind/ Desc	Total		Data	Ind/ Desc	Total		Data	Ind/ Desc	Total
WT	33% 5.50	1828 3.78	2682 15.50	4511 6.87	20% 6.67	1027 4.11	1596 9.39	2623 6.25	45% 5.63	745 2.64	1356 50.22	2101 6.80
WBOC	26% 4.33	1171 2.42	2016 11.65	3187 4.85	17% 5.67	820 3.28	1385 8.15	2205 5.25	45% 5.63	745 2.64	1354 50.15	2099 6.79
ASAP	29% 4.83	1153 2.38	1941 11.22	3094 4.71	19% 6.33	824 3.30	1386 8.15	2211 5.26	79% 9.88	745 2.64	1339 49.59	2084 6.74
WBOC-ASAP	28% 4.67	1196 2.47	2039 11.79	3235 4.92	19% 6.33	842 3.37	1405 8.26	2247 5.35	64% 8.00	745 2.64	1352 50.07	2097 6.79
delay-30	18% 3.00	734 1.52	1399 8.09	2133 3.25	10% 3.33	395 1.58	801 4.71	1196 2.85	39% 4.88	507 1.80	871 32.26	1378 4.46
full-delay	17% 2.83	718 1.48	1489 8.61	2207 3.36	10% 3.33	384 1.54	779 4.58	1163 2.77	27% 3.38	303 1.07	572 21.19	875 2.83
WT-TMP	27% 4.50	1383 2.86	2134 12.34	3517 5.35	14% 4.67	596 2.38	1026 6.04	1623 3.86	27% 3.38	303 1.07	572 21.19	875 2.83
WBOC-TMP	20% 3.33	829 1.71	1573 9.09	2402 3.66	10% 3.33	390 1.56	806 4.74	1196 2.85	27% 3.38	303 1.07	573 21.22	876 2.83
ASAP-TMP	21% 3.50	841 1.74	1563 9.03	2404 3.66	11% 3.67	412 1.65	828 4.87	1240 2.95	34% 4.25	303 1.07	572 21.19	875 2.83

Table 5-6. The amount of disk traffic with the write-through policy on the server. The top line for each client policy is the disk utilization and the number of disk writes for the three benchmarks. The bottom line is the top line divided by the disk utilization or number of disk writes with the full-delay policy on the client and the delay-30 policy on the server.

server transactions, the server delay-30 policy is able to eliminate many unnecessary disk operations; in the worst case, the disk is only 6% utilized.

For the write-intensive Sort benchmark, the client policy does have an impact on the amount of disk traffic. When the client uses any of the four non-delayed-write policies the disk writes increase by 42-71% over when full-delay is used. In addition, the disk is twice as utilized.

It is interesting to note that with the Andrew benchmark the number of indirect and file descriptor disk writes are highest when the client uses a full-delay policy. This increase is actually totally due to extra descriptor writes and only occurred on two of the three runs of the benchmark. This increase in descriptor traffic is probably due to

variabilities in when the daemon that cleans the cache ran on the server. Dirty descriptor blocks get written to disk the next time that the daemon runs. Thus if the daemon ran while the client was writing back all of its data blocks at the end of the benchmark then the number of descriptor disk writes would be higher than if daemon did not run at all.

When the server uses write-through instead of delay-30, the amount of disk traffic goes up tremendously (see Table 5-6 and Figure 5-2); the number of disk writes increases by up to a factor of 7 and the disk is up to 79% utilized. With write-through, whenever the server receives a write request for a new block from a client, it writes the data, indirect and file descriptor blocks through to disk. As a result, each client write

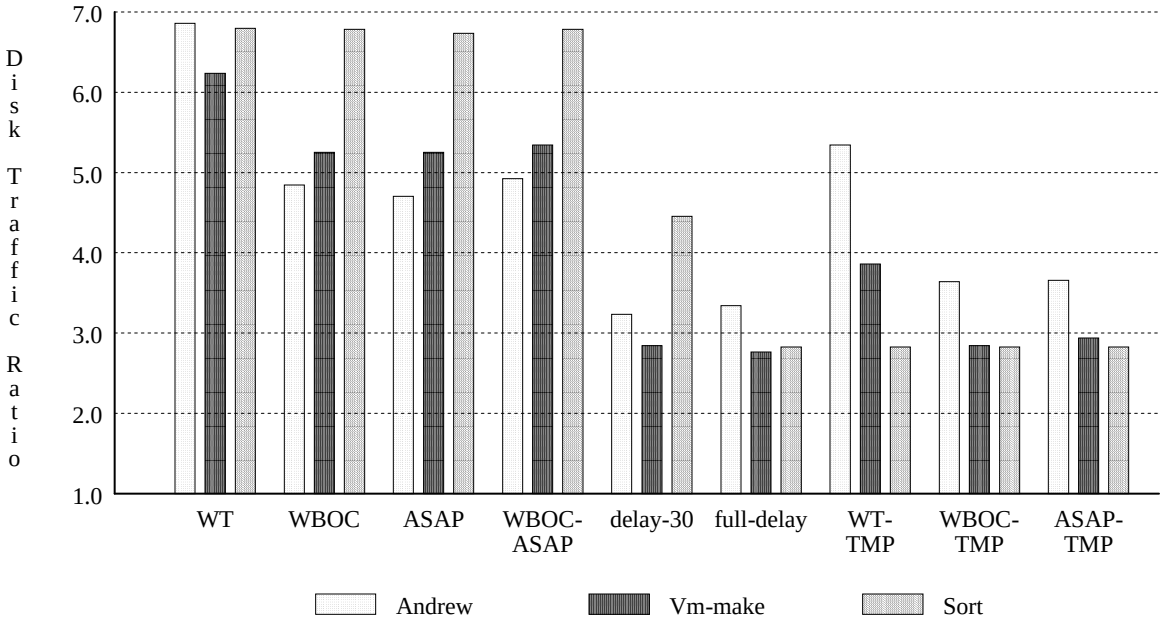

Figure 5-2. Ratio of disk writes with a write-through policy on the server and the client policies in this figure to disk writes with full-delay on the client and delay-30 on the server.

operation can require up to three disk writes. In addition, when files are created, the directory that the file is created in is also written through to disk; this will result in both extra data writes and extra descriptor writes. The greatest increase in disk writes comes from file descriptor and indirect block writes which increase by as much as a factor of 50; whereas with the server delay-30 policy the descriptor and indirect blocks may only be written to disk once for an entire file, with write-through they are written to disk once for every block.

The worst client policy in conjunction with server WT is client WT. Client WT is the worst because it is the only policy that does not wait until either a file system block fills up or the file is closed. As a result, it requires more server transactions and hence more disk writes than any other policy for both the Andrew and Vm-make benchmarks. Client WT does not require more disk transactions with Sort because Sort always writes data in file system block size chunks.

Figure 5-2 shows that for the three benchmarks even the full-delay and delay-30 client policies cause the amount of disk traffic to triple; the main cause of this increase is the extra file descriptor and indirect block writes. As the client uses more reliable writing policies that require more server interactions, disk writes increase by another factor of two. Except for the WT-TMP policy, the use of the full-delay policy with temporary files is effective in eliminating most of the extra disk writes caused by the more reliable policies. Both the WBOC-TMP and ASAP-TMP client policies have the same amount of disk traffic as the full-delay policy. The WT-TMP policy is not nearly as effective as the other temporary file policies in eliminating disk traffic for either the Andrew or Vm-make benchmarks; this is for the same reason given above why normal

client write-through causes the highest number of disk writes.

The server ASAP policy potentially allows the server to eliminate many of the data, indirect and descriptor disk writes required with write-through. With the ASAP policy the data, descriptor and indirect block writes are scheduled to happen as soon as possible but the client write request is allowed to complete before the disk writes do. This has the advantage that, if a client is able to complete multiple accesses to a single data, descriptor or indirect block before the block can be written to disk, then disk writes can be eliminated. Table 5-7 and Figure 5-3 show that ASAP is able to eliminate up to half of the disk writes required with write-through.

Disk Traffic: ASAP on Server												
Client Policy	Andrew				Vm-make				Sort			
	Disk Util	Disk Writes			Disk Util	Disk Writes			Disk Util	Disk Writes		
		Data	Ind/ Desc	Total		Data	Ind/ Desc	Total		Data	Ind/ Desc	Total
WT	23% 3.83	1002 2.07	1497 8.65	2499 3.80	17% 5.67	759 3.04	1096 6.45	1855 4.42	69% 8.63	729 2.59	1075 39.81	1804 5.84
WBOC	16% 2.67	952 1.97	753 4.35	1705 2.60	11% 3.67	737 2.95	528 3.11	1266 3.01	29% 3.63	731 2.59	208 7.70	940 3.04
ASAP	21% 3.50	988 2.04	1106 6.39	2095 3.19	16% 5.33	752 3.01	999 5.88	1752 4.17	70% 8.75	727 2.58	1014 37.56	1742 5.64
WBOC-ASAP	21% 3.50	988 2.04	1101 6.36	2090 3.18	16% 5.33	750 3.00	1006 5.92	1756 4.18	70% 8.75	728 2.58	1020 37.78	1748 5.66
delay-30	11% 1.83	608 1.26	626 3.62	1234 1.88	7% 2.33	365 1.46	430 2.53	795 1.89	19% 2.38	415 1.47	148 5.48	563 1.82
full-delay	10% 1.67	597 1.23	584 3.38	1181 1.80	7% 2.33	363 1.45	391 2.30	755 1.80	14% 1.75	301 1.07	120 4.44	422 1.37
WT-TMP	16% 2.67	718 1.48	1151 6.65	1870 2.85	8% 2.67	389 1.56	545 3.21	935 2.23	35% 4.38	301 1.07	539 19.96	841 2.72
WBOC-TMP	12% 2.00	664 1.37	677 3.91	1342 2.04	7% 2.33	368 1.47	412 2.42	780 1.86	14% 1.75	302 1.07	114 4.22	416 1.35
ASAP-TMP	14% 2.33	712 1.47	787 4.55	1499 2.28	7% 2.33	384 1.54	460 2.71	845 2.01	34% 4.25	299 1.06	507 18.78	806 2.61

Table 5-7. The amount of disk traffic with the ASAP policy on the server. The top line for each client policy is the disk utilization and the number of disk writes for the three benchmarks. The bottom line is the top line divided by the disk utilization or number of disk writes with the full-delay policy on the client and the delay-30 policy on the server.

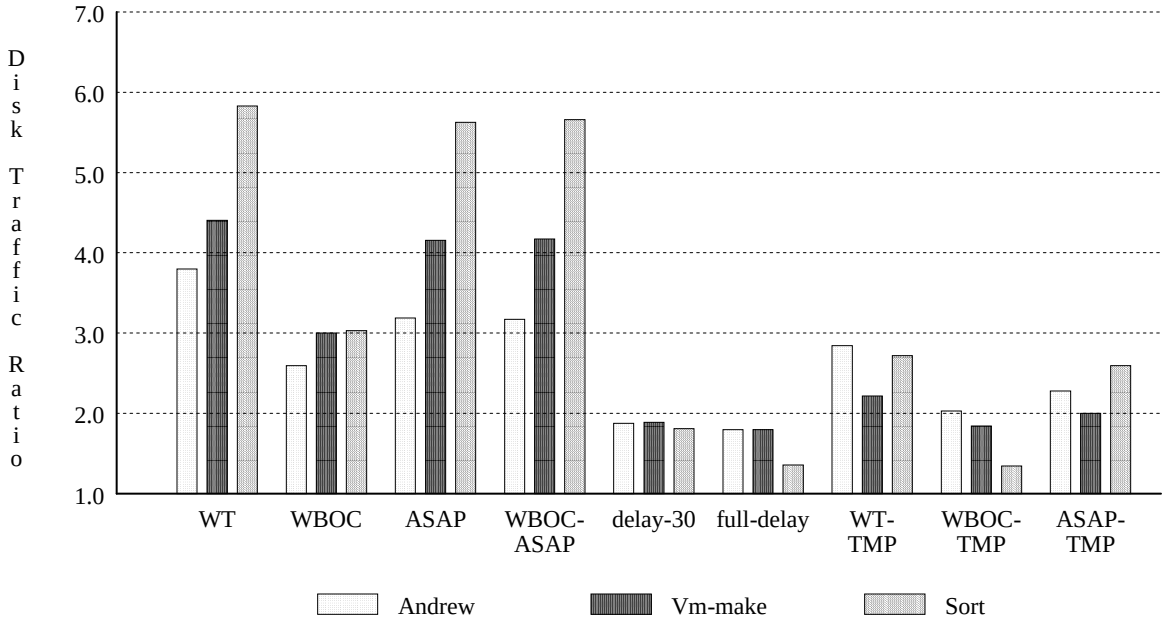

Figure 5-3. Ratio of disk writes with an ASAP policy on the server and the client policies in this figure to disk writes with full-delay on the client and delay-30 on the server.

The most dramatic reduction in disk traffic with server ASAP occurs when the client uses one of the delay-30, full-delay, WBOC or WBOC-TMP policies. The main reason for the reduction in disk traffic for these 4 policies is the tremendous reduction in indirect and file descriptor block writes. Indirect and file descriptor disk traffic is reduced because the delay policies and the WBOC policies all write many blocks of data to the server in succession. For example, WBOC will not write any data blocks through to the server until the file is closed. Once the file is closed the client will send over the entire file. Since many blocks are written in succession, each file descriptor and indirect block can be updated many times before it ends up getting written to disk.

The LDB server policy provides good reliability in the case of server crashes, yet provides reasonably low disk traffic (see Table 5-8 and Figure 5-4). With this policy,

Disk Traffic: Last Dirty Block												
Client Policy	Andrew				Vm-make				Sort			
	Disk Util	Disk Writes			Disk Util	Disk Writes			Disk Util	Disk Writes		
		Data	Ind/ Desc	Total		Data	Ind/ Desc	Total		Data	Ind/ Desc	Total
WT	9% 1.50	939 1.94	368 2.13	1307 1.99	6% 2.00	687 2.75	309 1.82	997 2.37	17% 2.13	721 2.56	60 2.22	781 2.53
WBOC	9% 1.50	937 1.94	316 1.83	1253 1.91	6% 2.00	685 2.74	269 1.58	954 2.27	17% 2.13	724 2.57	50 1.85	774 2.50
ASAP	15% 2.50	818 1.69	621 3.59	1439 2.19	12% 4.00	672 2.69	721 4.24	1393 3.32	52% 6.50	723 2.56	656 24.30	1379 4.46
WBOC-ASAP	15% 2.50	937 1.94	656 3.79	1594 2.43	12% 4.00	689 2.76	744 4.38	1433 3.41	50% 6.25	724 2.57	665 24.63	1389 4.50
delay-30	6% 1.00	484 1.00	205 1.18	690 1.05	3% 1.00	260 1.04	190 1.12	450 1.07	13% 1.63	403 1.43	34 1.26	438 1.42
full-delay	7% 1.17	483 1.00	228 1.32	711 1.08	3% 1.00	250 1.00	184 1.08	434 1.03	8% 1.00	282 1.00	27 1.00	309 1.00
WT-TMP	7% 1.17	596 1.23	278 1.61	875 1.33	3% 1.00	258 1.03	206 1.21	465 1.11	8% 1.00	280 0.99	27 1.00	307 0.99
WBOC-TMP	7% 1.17	595 1.23	250 1.45	845 1.29	3% 1.00	256 1.02	181 1.06	437 1.04	8% 1.00	282 1.00	28 1.04	310 1.00
ASAP-TMP	8% 1.33	559 1.15	303 1.75	862 1.31	3% 1.00	260 1.04	201 1.18	461 1.10	24% 3.00	282 1.00	346 12.81	628 2.03

Table 5-8. The amount of disk traffic with the LDB policy. The top line for each client policy is the disk utilization and the number of disk writes for the three benchmarks. The bottom line is the top line divided by the disk utilization or number of disk writes with the full-delay policy on the client and the delay-30 policy on the server.

data, descriptor, and indirect blocks are not written through to disk until the last dirty block from the file arrives from the client; the server uses the delay-30 policy in combination with this policy so that modified directories will get written through to disk periodically. The LDB policy eliminates varying amounts of disk traffic depending on the client's policy.

When clients use either the full-delay or delay-30 policy, LDB keeps the disk traffic fairly low; each indirect block only gets written to disk once and there is exactly one file descriptor block write per file. The one benchmark for which LDB is the least effective is Sort; when the delay-30 policy is used on the client, Sort generates 50% more disk writes than with the full-delay policy. This is because Sort is the one bench-

mark that has temporary files that live longer than 30 seconds; as a result, some data from temporary files ends up getting written through to the server's disk.

The client WT and WBOC policies perform much better with the LDB policy than when either ASAP or WT are used on the server. When the client uses WT or WBOC, the server only ends up writing back the data for the file when the file is closed. However, this still requires up to 2.5 times as many disk writes as when the client uses a delayed-write policy.

I could have implemented the combination of the client WT policy and the LDB policy in a different way. When the client uses write-through, each block that is written to the server is the last dirty block that the client has for the file; thus, in the most

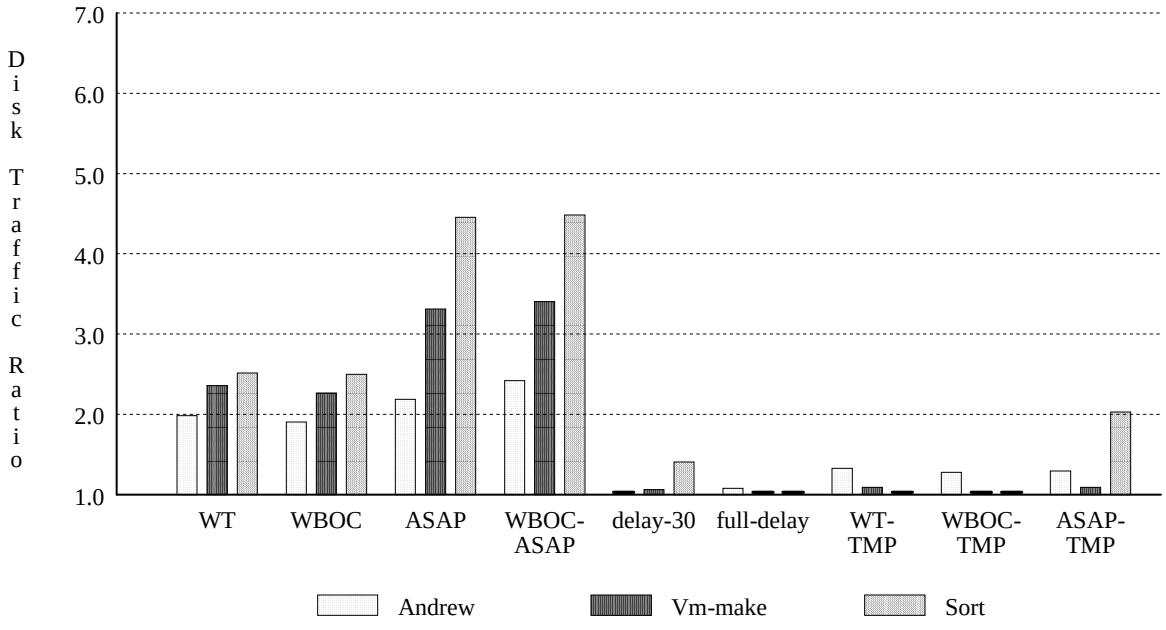

Figure 5-4. Ratio of disk writes with the LDB policy and the client policies in this figure to disk writes with full-delay on the client and delay-30 on the server.

straightforward implementation of LDB, each block that was written to the server would be marked as being the last dirty block. However, this would be no different than if the server used a write-through policy. In order to get a different data point I implemented the combination of client WT and LDB so that the file is only forced to disk when the file is closed.

The client ASAP policies do not perform very well under the LDB policy. Unless a user program can generate cache blocks faster than the operating system can write them back to the server, the client will think that each newly generated block is the last dirty block in the file. This results in up to 24 times as many descriptor and indirect block writes as when the client uses delayed-write.

The client policies that treat temporary files specially work well with the LDB policy. For both the Andrew and Vm-make benchmarks, all three of the temporary file policies require less than 40% more disk writes than when the client uses delayed-write for all files. Although both the WT-TMP and WBOC-TMP policies also perform very well on the Sort benchmark, the ASAP-TMP policy requires twice as many disk bytes as the delayed-write policy.

5.4. Client Elapsed Time

From a client's viewpoint, the most important performance measurement is the amount of time that it takes to execute the benchmark. Table 5-9 shows that, when the server uses the delay-30 policy, the elapsed time on the client is basically the same regardless of the policy used on the client. However, when the server uses a write-through policy, the client policy becomes very important (see Table 5.10 and Figure 5-

Client Elapsed Time and Server Utilization: 30-Second Delay on Server						
Client Policy	Andrew		Vm-make		Sort	
	Elapsed Time	Server Util	Elapsed Time	Server Util	Elapsed Time	Server Util
WT	279 1.04	12.98 1.17	300.08 1.04	9.33 1.14	66.10 1.14	11.22 2.01
WBOC	276 1.03	12.26 1.10	299.19 1.03	9.10 1.11	65.22 1.13	10.94 1.96
ASAP	273 1.02	12.55 1.13	296.60 1.02	9.30 1.14	62.53 1.08	11.51 2.06
WBOC-ASAP	273 1.02	12.52 1.12	296.68 1.02	9.31 1.14	62.48 1.08	11.47 2.06
delay-30	269 1.00	11.37 1.02	290.97 1.00	8.25 1.01	58.66 1.01	7.02 1.26
full-delay	268 1.00	11.13 1.00	289.81 1.00	8.18 1.00	57.92 1.00	5.58 1.00
WT-TMP	275 1.03	12.16 1.09	293.67 1.01	8.47 1.04	60.48 1.04	5.65 1.01
WBOC-TMP	274 1.02	11.55 1.04	293.71 1.01	8.23 1.01	60.52 1.04	5.63 1.01
ASAP-TMP	272 1.01	11.73 1.05	294.13 1.01	8.35 1.02	59.36 1.02	5.71 1.02

Table 5-9. The amount of time required to execute each benchmark and the percent of the server that is utilized with the delay-30 policy on the server. The top line for each client policy is the number of seconds to execute the benchmark and the percent of the server that was utilized while executing the benchmark. The bottom line is the top line divided by the execution time or server utilization in the best case (full-delay policy on the client and delay-30 policy on the server).

Client Elapsed Time and Server Utilization: WT on Server						
Client Policy	Andrew		Vm-make		Sort	
	Elapsed Time	Server Util	Elapsed Time	Server Util	Elapsed Time	Server Util
WT	424 1.58	11.03 0.99	370.13 1.28	9.13 1.12	124.45 2.15	10.11 1.81
WBOC	378 1.41	10.83 0.97	359.06 1.24	8.86 1.08	124.58 2.15	10.10 1.81
ASAP	316 1.18	12.87 1.16	312.51 1.08	10.04 1.23	65.54 1.13	17.64 3.16
WBOC-ASAP	345 1.29	12.01 1.08	329.82 1.14	9.71 1.19	87.38 1.51	14.15 2.54
delay-30	295 1.10	11.25 1.01	307.69 1.06	8.13 0.99	70.04 1.21	9.41 1.69
full-delay	292 1.09	10.79 0.97	304.88 1.05	8.09 0.99	60.53 1.05	6.69 1.20
WT-TMP	380 1.42	10.79 0.97	339.18 1.17	8.21 1.00	84.16 1.45	6.60 1.18
WBOC-TMP	343 1.28	10.56 0.95	325.96 1.12	7.93 0.97	83.77 1.45	6.57 1.18
ASAP-TMP	310 1.16	11.75 1.06	309.31 1.07	8.33 1.02	61.70 1.07	8.24 1.48

Table 5-10. The amount of time required to execute each benchmark and the percent of the server that is utilized with the write-through policy on the server. The top line for each client policy is the number of seconds to execute the benchmark and the percent of the server that was utilized while executing the benchmark. The bottom line is the top line divided by the execution time or server utilization in the best case (full-delay policy on the client and delay-30 policy on the server).

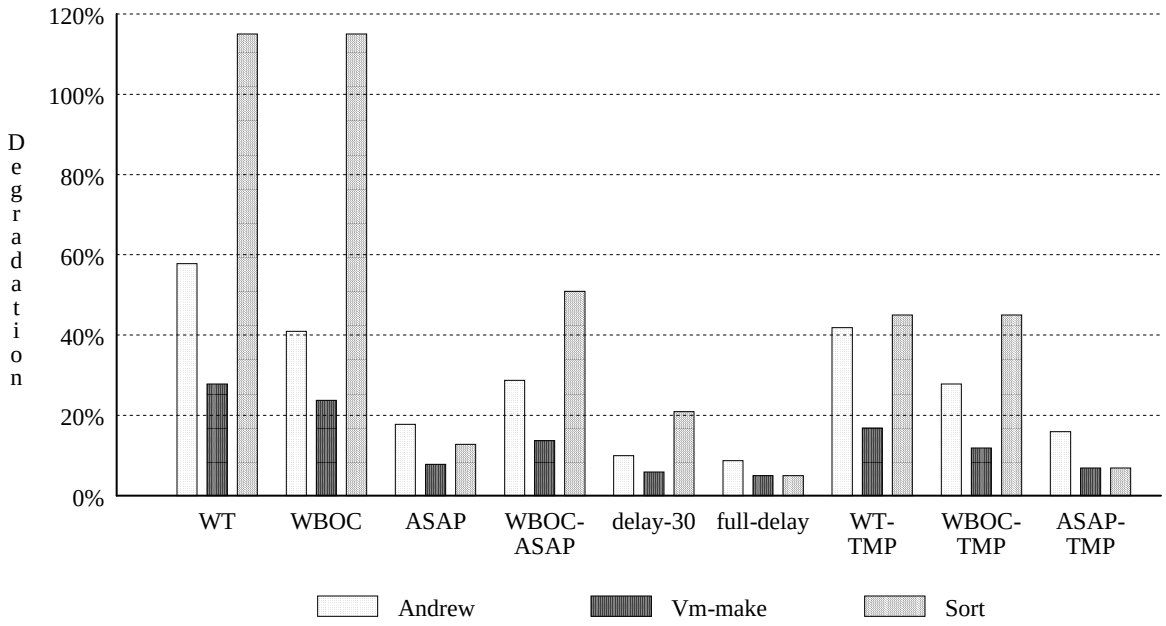

Figure 5-5. Additional elapsed time to execute each benchmark with a write-through policy on the server and the client policies in this figure relative to full-delay on the client and delay-30 on the server.

5). The extra synchronous disk writes on the server can make a client take up to twice as long to execute the benchmark as when the server uses a delay-30 policy. Even the WT-TMP and WBOC-TMP polices can slow the client down by over 40%. The only client policies that work uniformly well with server write-through are the full-delay, delay-30, ASAP and ASAP-TMP policies, which have at worst 21% degradation; these policies can handle the extra disk writes because the client programs do not have to wait for the writes to complete.

A server policy that provides reliability nearly as good as write-through yet does not slow down the client much is ASAP (see Table 5-11 and Figure 5-6). With the ASAP policy, the client slows down by less than 15% in the worst case. This performance is possible because the disk writes are asynchronous: a client does not have to

Client Elapsed Time and Server Utilization: ASAP on Server						
Client Policy	Andrew		Vm-make		Sort	
	Elapsed Time	Server Util	Elapsed Time	Server Util	Elapsed Time	Server Util
WT	282 1.05	14.42 1.30	301.27 1.04	10.40 1.27	65.71 1.13	17.05 3.06
WBOC	279 1.04	13.01 1.17	300.22 1.04	9.63 1.18	66.04 1.14	12.99 2.33
ASAP	277 1.03	13.65 1.23	298.06 1.03	10.27 1.26	61.94 1.07	17.72 3.18
WBOC-ASAP	277 1.03	13.61 1.22	298.42 1.03	10.28 1.26	62.00 1.07	17.68 3.17
delay-30	270 1.01	11.65 1.05	292.28 1.01	8.28 1.01	58.86 1.02	7.91 1.42
full-delay	268 1.00	11.47 1.03	290.27 1.00	8.20 1.00	57.98 1.00	5.90 1.06
WT-TMP	276 1.03	13.14 1.18	294.77 1.02	8.72 1.07	60.90 1.05	8.62 1.54
WBOC-TMP	274 1.02	12.06 1.08	294.32 1.02	8.33 1.02	60.80 1.05	6.01 1.08
ASAP-TMP	273 1.02	12.40 1.11	293.37 1.01	8.54 1.04	59.38 1.03	8.58 1.54

Table 5-11. The amount of time required to execute each benchmark and the percent of the server that is utilized with the ASAP policy on the server. The top line for each client policy is the number of seconds to execute the benchmark and the percent of the server that was utilized while executing the benchmark. The bottom line is the top line divided by the execution time or server utilization in the best case (full-delay policy on the client and delay-30 policy on the server).

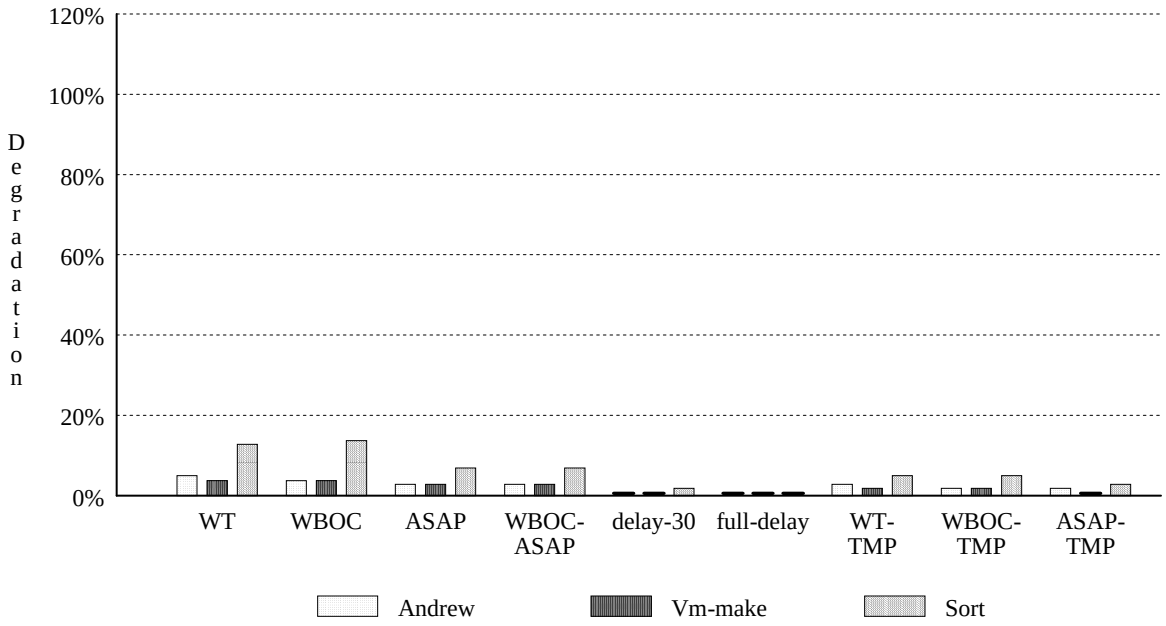

Figure 5-6. Additional elapsed time to execute each benchmark with an ASAP policy on the server and the client policies in this figure relative to full-delay on the client and delay-30 on the server.

wait for the disk write to complete before it can continue execution.

The LDB server policy provides better client performance than write-through but not nearly as good as ASAP (see Table 5-12 and Figure 5-7). Since clients have to wait for disk writes to complete when they close a file under the WBOC and WT policies, the client's performance degrades by up to 35%. However, if the client uses any of the other policies, then the degradation is 14% or less, with no noticeable degradation with the full-delay and delay-30 policies.

5.5. Server Utilization

Although the delay-30 policy on the server allows clients to use more reliable policies without suffering degradation, the more reliable policies can put a high load on the

Client Elapsed Time and Server Utilization: Last-dirty-block Policy						
Client Policy	Andrew		Vm-make		Sort	
	Elapsed Time	Server Util	Elapsed Time	Server Util	Elapsed Time	Server Util
WT	306 1.14	12.87 1.16	317.58 1.10	9.72 1.19	79.49 1.37	10.34 1.85
WBOC	302 1.13	12.09 1.09	315.66 1.09	9.26 1.13	78.58 1.36	10.67 1.91
ASAP	277 1.03	13.75 1.24	299.14 1.03	10.54 1.29	62.51 1.08	20.33 3.64
WBOC-ASAP	294 1.10	13.15 1.18	305.30 1.05	10.38 1.27	64.84 1.12	19.85 3.56
delay-30	269 1.00	11.68 1.05	291.85 1.01	8.43 1.03	59.47 1.03	8.02 1.44
full-delay	267 1.00	11.32 1.02	291.47 1.01	8.34 1.02	57.94 1.00	6.07 1.09
WT-TMP	292 1.09	12.09 1.09	302.90 1.05	8.71 1.06	65.28 1.13	5.60 1.00
WBOC-TMP	290 1.08	11.42 1.03	301.56 1.04	8.24 1.01	65.81 1.14	6.09 1.09
ASAP-TMP	275 1.03	12.27 1.10	293.59 1.01	8.60 1.05	59.29 1.02	10.90 1.95

Table 5-12. The amount of time required to execute each benchmark and the percent of the server that is utilized with the last-dirty-block policy. The top line for each client policy is the number of seconds to execute the benchmark and the percent of the server that was utilized while executing the benchmark. The bottom line is the top line divided by the execution time or server utilization in the best case (full-delay policy on the client and delay-30 policy on the server).

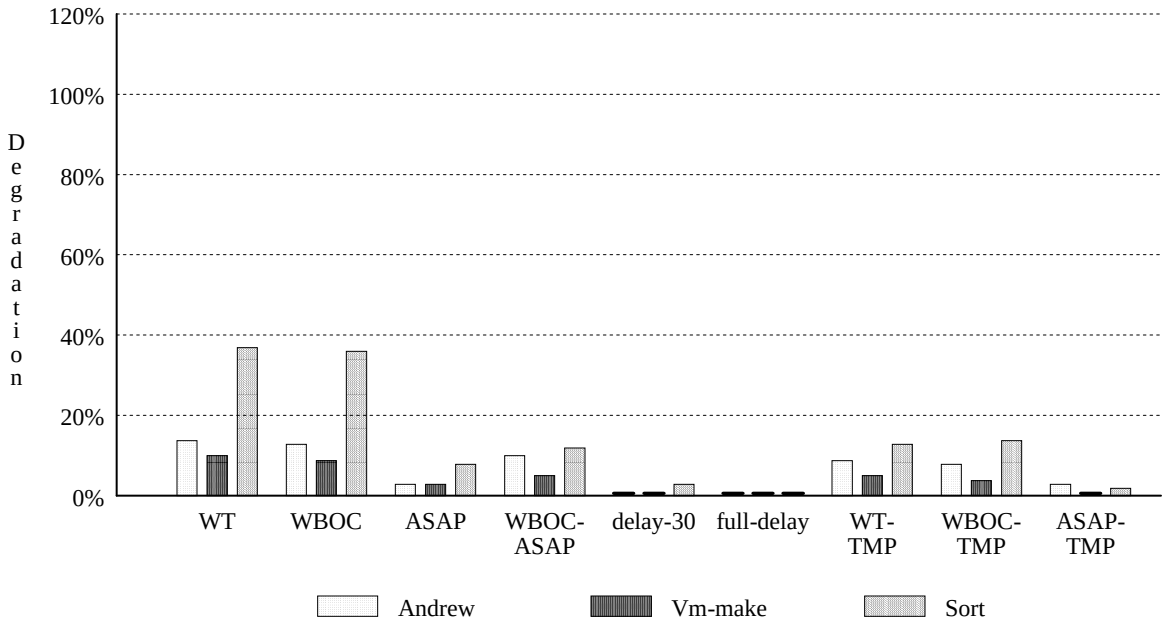

Figure 5-7. Additional elapsed time to execute each benchmark with an LDB policy on the server and the client policies in this figure relative to full-delay on the client and delay-30 on the server.

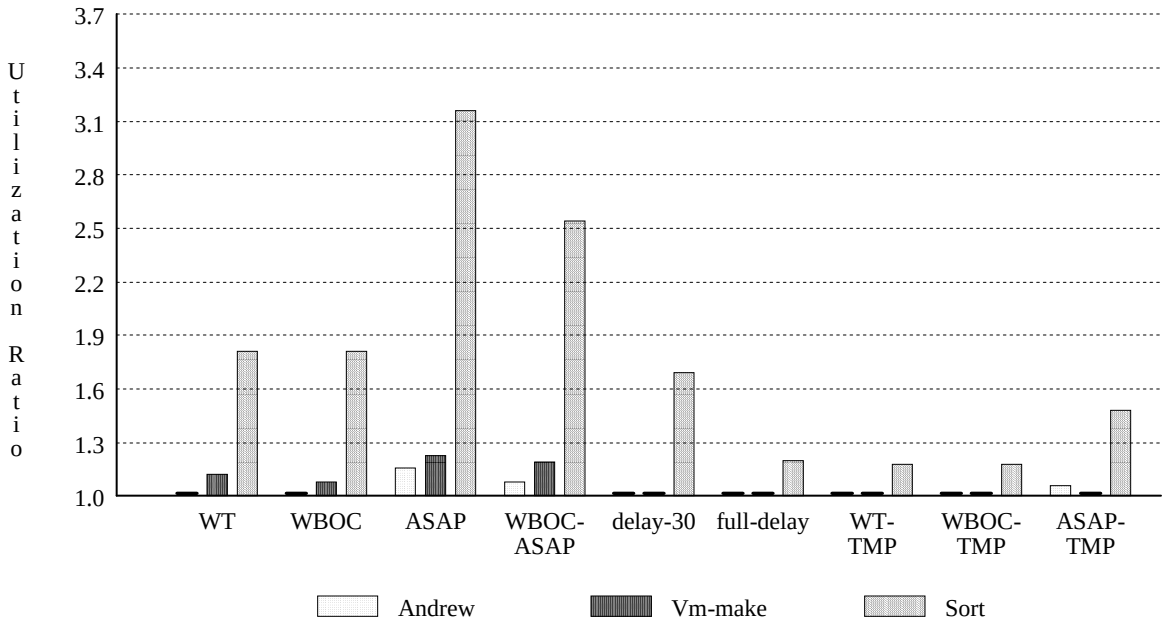

Figure 5-8. Ratio of server utilization with a write-through policy on the server and the client policies in this figure to server utilization with a full-delay policy on the client and the delay-30 policy on the server.

server. Table 5-9 presented in the previous section shows the server utilization with the delay-30 server policy when executing the 3 benchmarks with the 9 different client writing policies. With the Andrew and Vm-make benchmarks, the clients are able to use the more reliable policies without adversely effecting server utilization; the worst case is client WT, which utilizes the server 1.17 times as much as the best case utilization, which is obtained with the full-delay policy. However, with the very intensive Sort benchmark, all of the non-delayed-write polices cause the server utilization to double.

When the server uses write-through instead of delay-30, the server utilization improves for some client writing polices and gets worse for others (see Table 5-10 and Figure 5-8). If the client uses the WT or WBOC client polices, the clients slow down

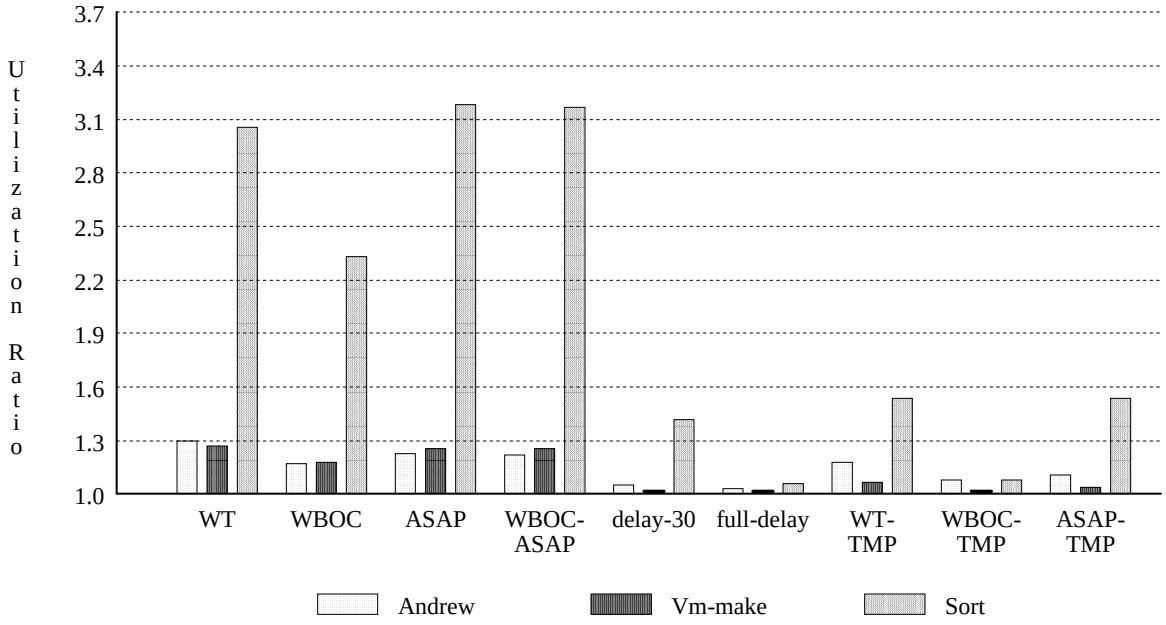

Figure 5-9. Ratio of server utilization with an ASAP policy on the server and the client policies in this figure to server utilization with a full-delay policy on the client and the delay-30 policy on the server.

so much that the server utilization drops for all three benchmarks. When the client uses either the ASAP or WBOC-ASAP policies, the server utilization increases so that for the Sort benchmark it is up to 3 times higher than the best case; for the Andrew and Vm-make benchmarks, the utilization increases slightly but is still at worst only 1.26 times the best case utilization. The reason why the client ASAP policies cause the server utilization to increase is that the server's workload is increased but the clients are not much slower than the best case elapsed time. For the same reason even the client delayed-write and temporary-file polices have slightly higher server utilization with the Sort benchmark and server write-through.

The server ASAP policy is the worst policy for server utilization (see Table 5-11 and Figure 5-9). With this policy the clients do not slow down by much, but the server

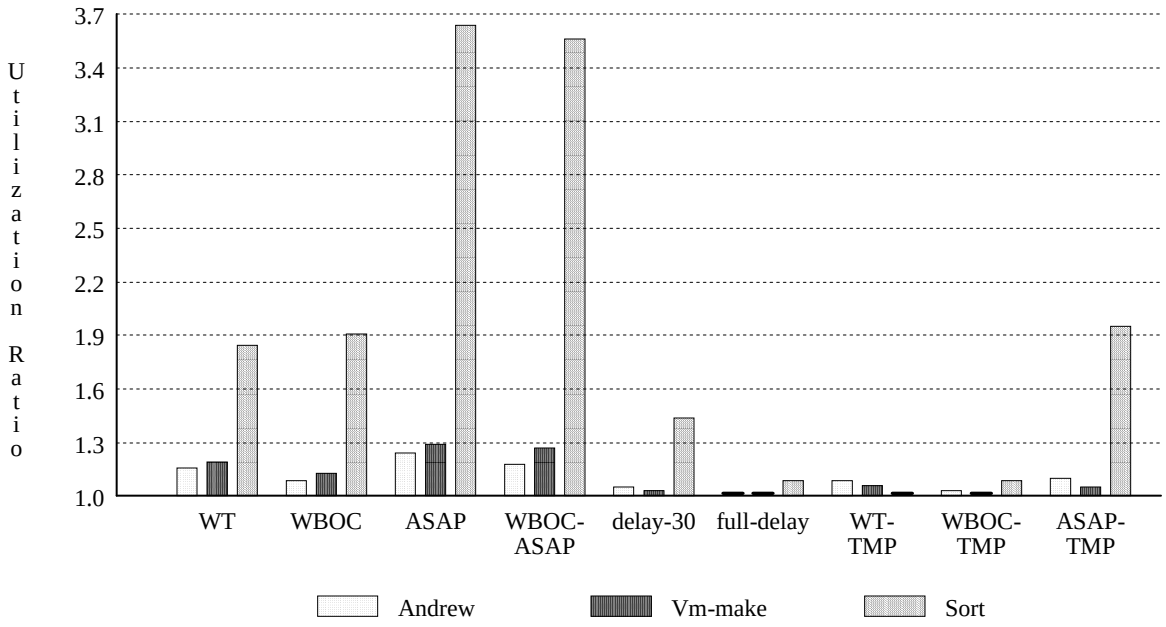

Figure 5-10. Ratio of server utilization with the LDB policy and the client policies in this figure relative to a full-delay policy on the client and the delay-30 policy on the server.

does more work than when it uses delay-30. For the Andrew and Vm-make benchmarks, the server utilization is still fairly low; in the worst case it is only 1.3 times the best case utilization. However, with the Sort benchmark, each of the policies WT, WBOC, ASAP and WBOC-ASAP have a server utilization that is between 2.5 and 3 times the best case utilization. With the delayed-write and temporary file policies, the utilization is worse for some policies and better for others; in the worst case, the utilization is 1.5 times the best case utilization.

The LDB policy is in between the server ASAP policy and the server WT policy (see Table 5-12 and Figure 5-10). For the Sort benchmark, it gives utilization as high as the server ASAP policy when the clients use either the ASAP or WBOC-ASAP policies and as low as the server WT policy if the clients use either the WT or WBOC poli-

cies. It is highest when clients use an ASAP policy because the clients force extra disk writes on each server cache write without slowing down. The utilization is lower with the WT and WBOC policies because only a single extra data block and disk block write is required for each file. One thing to note is that, although the server utilization with the client WT and WBOC policies is the same with the LDB and server WT policies, the client benchmarks complete much faster with the LDB policy.

5.6. Effect of Disk Layout on Write Performance

As mentioned before, the Sprite file system's disk reading and writing performance is not as good as that of other systems such as UNIX 4.2 BSD [MJL84]. Sprite's poor writing performance could potentially contribute to the extra client degradation, server utilization and disk utilization when more reliable server writing policies are used. There are two areas where Sprite disk writing performance could be improved. First, currently Sprite can only transfer one block per disk revolution because consecutive data blocks are not allocated in rotationally optimal locations on disk; by using a better disk layout policy, Sprite could get much higher throughput to disk. Second, Sprite does not put descriptor and indirect blocks close to the data blocks; this can require long seeks when a file descriptor or indirect block write is followed by a data block write.

The policy which would benefit most from improving the disk layout policy is LDB. When the client uses any policy but ASAP in conjunction with LDB, the server will have all of the files blocks in its cache before it has to write any of the blocks to disk. Thus, if consecutive blocks in a file were in rotationally optimal positions, the

server could write the entire file to disk very quickly. This would lower disk utilization and client degradation.

Neither the write-through or ASAP server policies would benefit much from a better layout policy. Both of these policies require that, whenever a data block is written to disk, the file descriptor and indirect blocks be written to disk as well. Thus, between every write of a data block there will have to be a seek to the descriptor or indirect block.

Reducing the seek time between data, indirect, and descriptor blocks could have a big effect on the performance of the server ASAP and write-through policies; its biggest effect would be on client elapsed time with the server write-through policy. In order to approximate the impact of the seek time I measured the writing performance with server write-through on both a very large disk partition which will cause long seeks and on a very small disk partition which will have short seeks. I measured that a client can

Client Degradation with Reduced Seek Times						
Client Policy	Andrew		Vm-make		Sort	
	Before	After	Before	After	Before	After
WT	58%	43%	28%	21%	115%	84%
WBOC	41%	30%	24%	18%	115%	84%
ASAP	18%	13%	8%	6%	13%	10%
WBOC-ASAP	29%	21%	14%	10%	51%	37%

Table 5-13. The effect of shortening seek times between file descriptor, indirect and data blocks when the server uses write-through. The *Before* column is the degradation when the benchmarks were run on the normal large disk partition (98% of disk) and the *After* column is an approximation of what the degradation would be if the benchmarks were run on a small partition (2% of the disk).

transfer 49 Kbytes per second to the small partition and 36 Kbytes per second to the large partition. This means that in the best case client degradation will only be 36/49, or 73% as high if the length of disk seeks were reduced. However, Table 5-13 shows that, even with this reduction in client degradation, clients still slow down substantially when they use either the WT or WBOC policies and the server uses write-through.

5.7. Comparison to NFS

Sun's Network File System (NFS) uses the WBOC-ASAP policy on the client and a write-through policy on the server. Table 5-10 shows that this policy will cause the Andrew benchmark to execute 29% slower than with the writing policies used on Sprite. Even if NFS is able to get the better disk performance shown in Table 5-13, the benchmark would execute 21% more slowly. In Chapter 4 I showed that Sprite executes the Andrew benchmark about 30% faster than NFS. It appears from the results in this chapter that up to two-thirds of the performance difference comes from the writing policies used in NFS.

5.8. Summary and Conclusions

In this chapter I have examined numerous client and server writing policies. As expected, the writing policies that provide the best overall performance in terms of network load, disk load, server load and client elapsed time are delayed-write policies. From a client's viewpoint, the most important performance metric is execution time degradation. If the server uses a delayed-write policy, then the client's policy will have only a small impact on the time it takes to execute a benchmark; the client delayed-write policies provide the best client performance, but even the more reliable client

policies will cause degradation of at most 14% if the server uses the delay-30 policy. The ASAP and LDB server policies also allow the client to use more reliable policies with only modest degradation. However, if the server uses a write-through policy, then the more reliable client policies can cause up to a 115% degradation in client performance.

In terms of determining the scalability of the system, the most important factors are server, disk and network load; the writing policies can have a dramatic impact on each of these factors. The amount of network traffic is independent of the server policy. The more reliable client policies can cause the network traffic to more than double over the delayed-write policies. Thus, the delayed-write client policies can allow the network to support up to twice as many clients as the other client writing policies.

Both the client and the server writing policies have a big impact on server utilization. The non-delayed-write client policies can cause the server utilization to double even when the server uses the delay-30 policy. The client policy has an even bigger impact when the server uses more reliable policies such as write-through and ASAP; for these server policies the non-delayed-write client policies can cause the server utilization to triple. If the server uses a non-delayed-write policy, even the delay-30 client policy causes server utilization to be up to 1.7 times higher. Thus, if the server and the client use delayed-write policies, then a server can support up to 3 times as many clients as when non-delayed-write policies are used.

The utilization of the disk also increases dramatically with non-delayed-write policies. When the server uses a write-through policy the amount of disk traffic will

increase by up to a factor of 7; even the client delayed-write policies will cause the disk traffic to triple with server write-through. The LDB and ASAP server policies give less disk writes than server write-through, but more than when the server uses delay-30. Thus, if the server does not use the delay-30 policy then the number of clients that can be supported by each disk will be much lower.

One potential way to give clients both high reliability and high performance is to treat temporary files specially: use a reliable policy for most files but use a full-delay policy for temporary files. For the three benchmarks, all of which use temporary files extensively, treating temporary files specially is reasonably effective. With the delay-temp-files policies, the network traffic and server utilization are close to the delay-30 and full-delay client policies. However, when the server uses more reliable writing policies, the benchmarks can execute up to 45% more slowly than the best client policy, and the disk can be up to twice as utilized. The one benchmark for which delaying temporary files works best is Sort, which writes all of its output except for the final result to temporary files. Thus, using full-delay on temporary files and a more reliable policy on other files shows some potential, but it is not able to totally insulate the client from more reliable server writing policies. If other files besides those in the /tmp directory were considered temporary files (such as object files), then special casing of temporary files would be more effective.

Server write-through is by far the worst server policy in terms of performance. I have presented two alternatives to server write-through, which provide nearly as good reliability, yet with better performance. Both the ASAP and LDB policies work very well with the full-delay and 30-second delay client policies; they give low disk and

server utilization, while allowing the clients to execute without suffering degradation. The policies which delay temporary files also work reasonably well with the ASAP and LDB policies, although there is higher disk utilization and client degradation than when the client uses the full-delay and delay-30 policies. The ASAP and LDB policies work better than write-through with the 4 non-delayed-write client policies. However, LDB can still cause serious client degradation, and ASAP can cause very high disk and server utilization.

This chapter has shown that the writing policy can have a large impact on both client and server performance. The best policies for performance are the worst for reliability, and the best policies for reliability are the worst for performance. Thus, the writing policy to use for clients and servers must be a compromise between performance and reliability. The choice of the writing policy will become even more important in the future, as CPU speeds increase dramatically but disk speeds do not; any policy that requires application programs to wait for the disk will cause serious performance degradation.

CHAPTER 6

Variable-Sized Caches

6.1. Introduction

I have shown that file data caches are very effective in providing high performance to diskless clients. In order to get the maximum benefit from client caching, it is desirable to let each client cache be as large as possible. For example, applications that do not 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. Therefore a mechanism is needed that lets each file cache grow and shrink dynamically in response to changing demands on the machine's virtual memory system and file system.

This chapter looks at approaches to providing variable-sized file system caches. It is organized as follows: Section 6.2 summarizes previous work in this area; Section 6.3 describes the approach that I implemented in Sprite; Section 6.4 analyzes the performance of the Sprite mechanism; Section 6.5 measures the effect on performance of modifications to the Sprite algorithm; Section 6.6 compares the performance of the Sprite approach to other approaches; and Section 6.7 gives a summary and offers some conclusions.

6.2. Previous Work

The approach that has been commonly used to provide variable-size file system caches is to combine the virtual memory and file systems together; this is generally called the *mapped-file* approach. To access a file, it is first mapped into a process's virtual address space and then read and written just like virtual memory. This approach eliminates the file cache entirely; the standard page replacement mechanisms automatically balance physical memory usage between file and program information. Mapped files were first used in Multics [BCD72, DaD68] and TENEX [BBM72, Mur72]. More recently they have been implemented in Pilot [Red80], Accent [RaR81, RaF86], Apollo [LLH85, Lea83] and Mach [Ras87].

Mapped files present a much different interface than systems such as UNIX that keep the file system and virtual memory system separate. Under the UNIX approach, users use system calls such as *read* and *write* to access file data. These system calls copy data between the file data cache and the virtual address space of user processes. By using mapping techniques, the mapped-file approach can eliminate many of the copy operations required under the UNIX approach.

The main problem with the mapped-file approach is that it constrains the number of options available for caching and cache consistency; since all reads and writes happen directly to a client's memory, all clients must be allowed to cache files in their memory. This would make it impossible to use Sprite's simple cache consistency algorithm, which requires caches to be disabled under some conditions.

One scheme that keeps mapped file caches consistent without requiring users to lock their files is one that has been implemented by Kai Li [Li86]. His scheme provides cache consistency at the page level. It is a complex scheme in which each page is ‘‘owned’’ by a workstation. Whenever a workstation wishes to read a page that is not already in its memory, a copy of the page is fetched from the page’s owner. In order to modify a page, a copy of the page must be acquired from the owner, and then the workstation that is modifying the page becomes the owner of the page. Acquiring ownership causes the page to be removed from all other workstation’s memories.

Another potential problem with mapped files is that the hardware may make mapping difficult. Some newer workstations use a virtually addressed hardware cache [Hil86, Kel86, SSS85]. These caches do not support synonyms – multiple virtual addresses pointing to the same physical address. Thus, if one file system page is mapped into two different processes’ virtual address spaces at different virtual addresses and one process is modifying the page, the hardware will not guarantee that the two processes will see a consistent view of the data in the page. Actually, this presents a consistency problem similar to the file system consistency problem mentioned earlier, including problems related to both concurrent and sequential sharing. The result is that, on certain hardware, mapped files may introduce additional complexity and overhead.

A third problem with the mapped-file approach is that it treats virtual memory and file system data in the same way. Unfortunately, the access patterns of the two types of data may be entirely different. For example, file accesses are typically sequential and while virtual memory accesses are not. Therefore, it may make sense to use different

replacement strategies for the two types of data.

6.3. Sprite Mechanism

The mapped-file approach that I just described has the nice properties that it provides a single mechanism for accessing file and virtual memory data, and it eliminates copy operations. However, because of the potential problems with mapped files, I decided to investigate alternative mechanisms for providing variable-sized caches. The mechanism that I developed allows the file system cache to vary in size by having the virtual memory system and the file system modules negotiate over physical memory usage. The mechanism not only provides variable-sized caches, but it also allows Sprite to use a simple cache consistency mechanism, it works well on machines with virtually-addressed hardware caches, and it allows Sprite to treat virtual memory and file pages differently if that should become necessary or desirable. It has the disadvantage that it requires more page copying for I/O than the mapped-file approach.

In the Sprite mechanism, 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 [Nel86]. The file system keeps its cache blocks in perfect LRU order since all block accesses are made 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. This LRU time comparison is done using a simple procedural interface between the two modules.

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 [Nel86]. 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. The page is copied instead of remapped because, as explained below, there may be multiple file system blocks per page.

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 extended main memories 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 some sort of average? 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 the Sun-3 implementation of Sprite, which has 8-Kbyte pages but 4-Kbyte file blocks, I 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.

I also considered more centralized approaches to trading off physical memory between the virtual memory page pool and the file cache. One 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. I 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 the distributed mechanism, while leaving each module free to age its own pages in the most convenient way. The Sprite approach also permits the relative aging rates to be adjusted for virtual memory and file pages, which we have found desirable. The effect of this adjustment is discussed in Section 6.5.

6.4. Variable-Size Cache Performance

The Sprite variable-size cache mechanism will work very well if only file-intensive programs are run, because the cache will be allowed to grow very large.

Likewise, if users run purely VM-intensive programs, the cache will become small and let most of physical memory be used by the virtual memory system. This section looks at how well the Sprite variable-size cache mechanism performs when users run both file- and virtual-memory-intensive programs. I will use the results from a benchmark that is both file- and virtual-memory-intensive to answer the following questions:

1) How well do fixed-size caches perform with the benchmark?

2) How do fixed-size and variable-size caches compare?

3) What is the effect of changes in physical memory size?

The benchmark that I used is an edit-compile-debug benchmark that runs under the X11 window system on Sprite (see Table 6-1). This benchmark represents work that is commonly done on Sprite, and is both VM and FS intensive. In order to

Component	Description	FS I/O	VM Image Size
Edit	Run window-based editor on 2500 line file.	70 Kbytes	560 Kbytes
Compile	Compile VM Module	800 Kbytes	1 Mbyte
Link	Link the kernel	8 Mbytes	3 Mbytes
Debug	Run kernel debugger	4 Mbytes	8.5 Mbytes
Environment	The X window system plus several typescript windows and tools.	--	5 Mbytes

Table 6-1. This table describes each of the components of the edit-compile-debug benchmark. The first two columns describe the component of the benchmark. The third column gives the amount of bytes read and written by each benchmark step. The last column gives the size of the largest virtual memory image of the step. The last row is not a step in the benchmark but rather shows the total amount of memory required by the basic environment in which the benchmark is running.

facilitate running the benchmark, I modified Sprite so that I could inject mouse events into the X11 input stream. Using this feature, I was able to move the mouse around under program control and enter commands in various windows; basically, I was able to simulate under program control the actions of a normal user of the window system.

The benchmark was run on a Sun-3/75 client with 16 Mbytes of memory, and the server was a Sun-3/180 with 16 Mbytes of memory. The server used an 8-Mbyte cache. I varied both the amount of physical memory available on the client and whether or not the client was using a fixed-size cache or the Sprite variable-size cache mechanism. Each benchmark consisted of two runs through the edit-compile-link-debug loop, where the benchmark components were run in one of three windows. Each data point was taken from the average of three runs of the benchmark. In this section I will only present the most important results from the benchmark; see Appendix C for more detailed results including standard deviations.

6.4.1. Variable vs. Fixed-Size Caches

The results from Chapter 4 suggest that a large fixed-size cache will provide the best performance for file-intensive programs. However, for the edit-compile-debug benchmark, the smallest fixed-size cache is best. Figure 6-1 gives the elapsed time and server utilization for the benchmark as a function of the amount of physical memory available on the client and the size of its file cache. A cache of 0.5 Mbytes provides the lowest elapsed time, and a cache from 0.5 Mbytes to 1 Mbyte gives the lowest server utilization for the benchmark; note that this benchmark is so virtual-memory-intensive that even with the largest physical memory the smallest-sized cache is best.

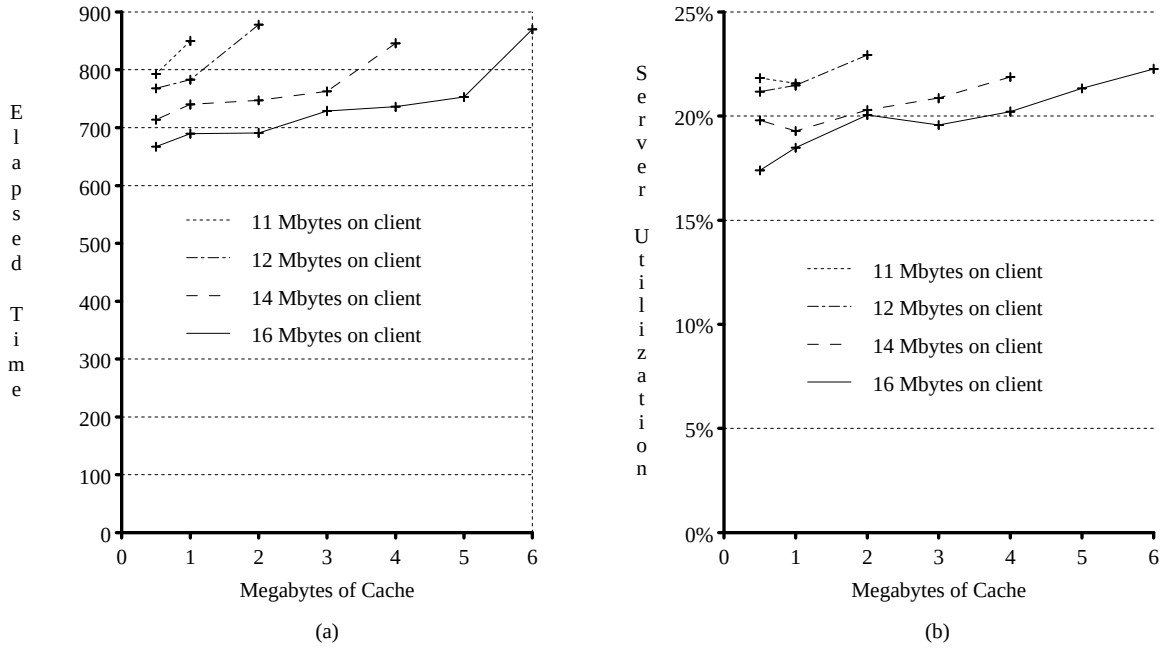

Figure 6-1. Elapsed time and server utilization for the edit-compile-debug benchmark with fixed-size caches as a function of client physical memory size. In both graphs the X-axis is the size of the file cache. In graph (a) the Y-axis is the number of seconds to execute the benchmark and in graph (b) the Y-axis is the percent of the server's CPU that was utilized while the client was executing the benchmark. The system thrashed whenever the amount of physical memory left for the virtual memory system dropped below 10 Mbytes. I did not run the benchmark for points where thrashing occurred (since elapsed time more than doubles), which explains why some curves have fewer data points than others.

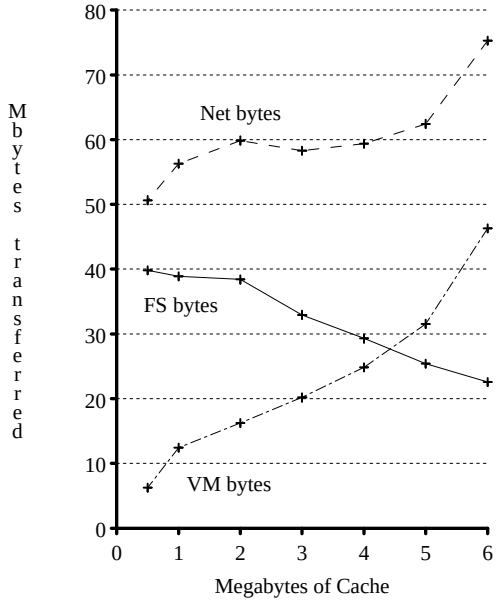

Figure 6-2. This graph gives the number of Mbytes transferred across the network for the edit-compile-debug benchmark with fixed-size caches and 16 Mbytes of memory on the client. The X-axis is the size of the cache and the Y-axis is the number of Mbytes transferred.

Figure 6-2 clearly shows why the smallest cache is best for this benchmark (graphs of network bytes transferred for the other four memory sizes will yield similar results - see Appendix C). As the cache grows in size, the number of file system bytes transferred drops. However, because the amount of physical memory available for virtual memory decreases with the increased file system cache, the number of page faults and hence virtual memory system bytes transferred increases. This causes a net increase in the number of network bytes transferred and a corresponding increase in client degradation and server utilization.

The results with fixed-size caches clearly demonstrate that there is no one cache size that will yield the best performance for both the file intensive programs from Chapter 4 and the file and virtual-memory intensive benchmark in this chapter. For-

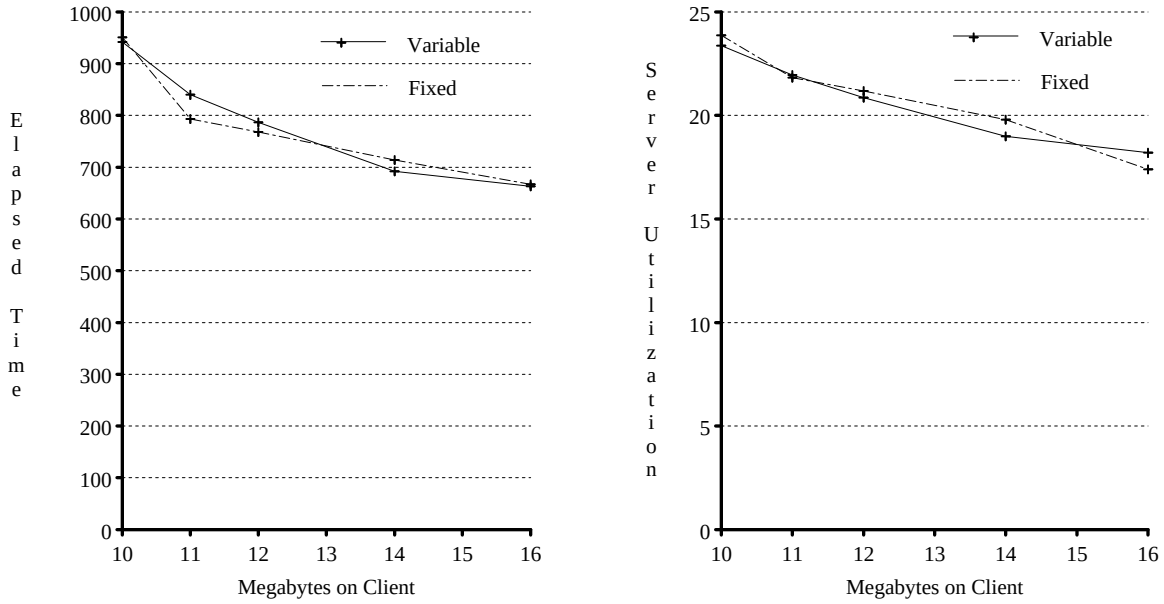

Figure 6-3. Elapsed time and server utilization for the edit-compile-debug benchmark with a variable-sized cache and with the smallest fixed-size cache as a function of physical memory size. In both graphs the X-axis is the amount of cache. In graph (a) the Y-axis is the number of seconds to execute the benchmark and in graph (b) the Y-axis is the percent of the server's CPU that was utilized while the client was executing the benchmark.

tunately, the Sprite variable-size cache mechanism works well for both types of benchmarks. Figure 6-3 shows that, in terms of elapsed time and server utilization, the variable-size and fixed-size cache mechanisms provide nearly identical performance. The reason why the performance is similar is clearly demonstrated in Figure 6-4, which gives the amount of network traffic. The variable-sized cache gives consistently fewer file system bytes transferred than a fixed-size cache, and the fixed-sized cache gives fewer virtual memory bytes transferred. However, in terms of overall net bytes transferred, the variable-size is slightly better than the best fixed-size cache. Thus, the poorer virtual memory performance for the variable-size cache is more than offset by the much better file system performance.

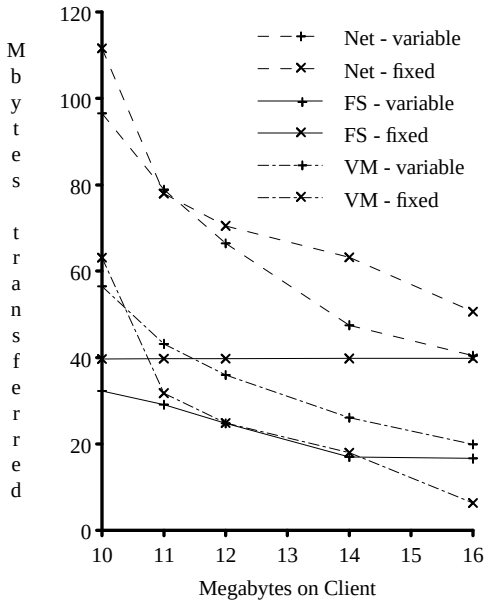

Figure 6-4. This graph gives the number of Mbytes transferred across the network with variable-size and smallest-fixed-size caches and 16 Mbytes of memory on the client. The X-axis is the amount of cache and the Y-axis is the number of Mbytes transferred.

The measurements from Chapter 4 (see Figure 4-1) and of the file- and virtual-memory intensive benchmark show that the Sprite variable-size cache mechanism is uniformly better than any fixed-size cache. When file-intensive benchmarks are run, the variable-size cache lets the cache get as large as is necessary. However, even when file and virtual memory activities are intermixed, the variable-size cache provides performance that is at least as good as the performance possible with the optimal fixed-size cache.

6.4.2. Negotiation Activity

The edit-compile-debug benchmark shifts between file- and virtual-memory-intensive programs. This requires that there be constant shifts in the allocation of

Client Mem (Mbytes)	Min Cache Size (Mbytes)	Max Cache Size (Mbytes)	FS Asks VM		VM Asks FS	
			Num	Satisfied	Num	Satisfied
10	0.25	5.6	8125	1810	2942	1846
11	0.25	6.4	7105	1889	2610	1967
12	0.25	6.9	5840	1964	2555	2075
14	0.25	8.7	4012	1957	2669	2162
16	0.34	8.8	3652	1937	2629	2229

Table 6-2. Traffic between the virtual memory system and the file system. The first column gives the amount of physical memory available on the client. The second and third columns give the minimum and maximum cache sizes during the benchmark. The fourth and fifth columns are the number of times that the file system asked the virtual memory system for the access time of its oldest page and the number of times that it was able to get a page from the virtual memory system. The sixth and seventh columns are the same as the previous two, except that they are the number of times the virtual memory system asked the file system for memory.

physical memory between the file system and the virtual memory system (see Table 6-2). The minimum and maximum cache size columns from Table 6-2 show that the file cache varied widely in size during the life of the benchmark, going from the minimum possible size (0.25 Mbytes) up to over half the amount of physical memory available. As the amount of physical memory increased, the maximum size of the cache increased as well; the variable-size cache mechanism allowed the file system to take advantage of the extra physical memory.

Table 6-2 also shows the amount of negotiation that went on between the virtual memory system and the file system. As the amount of physical memory increased, the number of times that the file system attempted to get memory from the virtual memory system dropped dramatically; however, the number of times that the file system was successful in stealing a page from the virtual memory system remained fairly constant across all memory sizes. In contrast, the number of requests for memory made by the

virtual memory system to the file system remained reasonably constant for all memory sizes, but the virtual memory system was more successful in taking pages from the file system as the amount of memory increased.

Table 6-2 suggests that the virtual memory system is much less elastic in its needs than the file system, at least for this benchmark; I hypothesize that this is true in general. The low success rate that the file system has when asking the virtual memory system for memory implies that the pages in the virtual memory system are being more actively used than those in the file system. Thus, the virtual memory system has fairly strict memory needs regardless of the physical memory size, and it actively uses the pages that it has. The file system, on the other hand, because it caches files after they are no longer being used, will grow to fill the available memory. Since the file system does not actively use many of its cached pages, its pages are the best candidates for recycling.

6.5. Penalizing the File System

The Sprite variable-size cache mechanism that I have described so far treats virtual memory and file system data the same; it is basically a global LRU mechanism where all pages are ordered by their LRU times. However, the two types of data are actually quite different. The sequential nature of file accesses [Ous85] means that a low file hit ratio should have a much smaller impact on system performance than a low virtual-memory hit ratio. Also, the level of interactive response relies almost entirely on virtual memory system performance, not on the performance of the file system. In this section I will investigate the effect on both overall and interactive performance of

giving the virtual memory system priority over the file system.

The method that I developed to bias against the file system involves adding a fixed number of seconds to the reference time of each virtual memory page. This makes each virtual memory page appear to have been referenced more recently than it actually was. For example, if 5 minutes is added to the reference time of each virtual memory page, then the file system will not be able to take any page from the virtual memory system that has been referenced within 5 minutes of the oldest file system page.

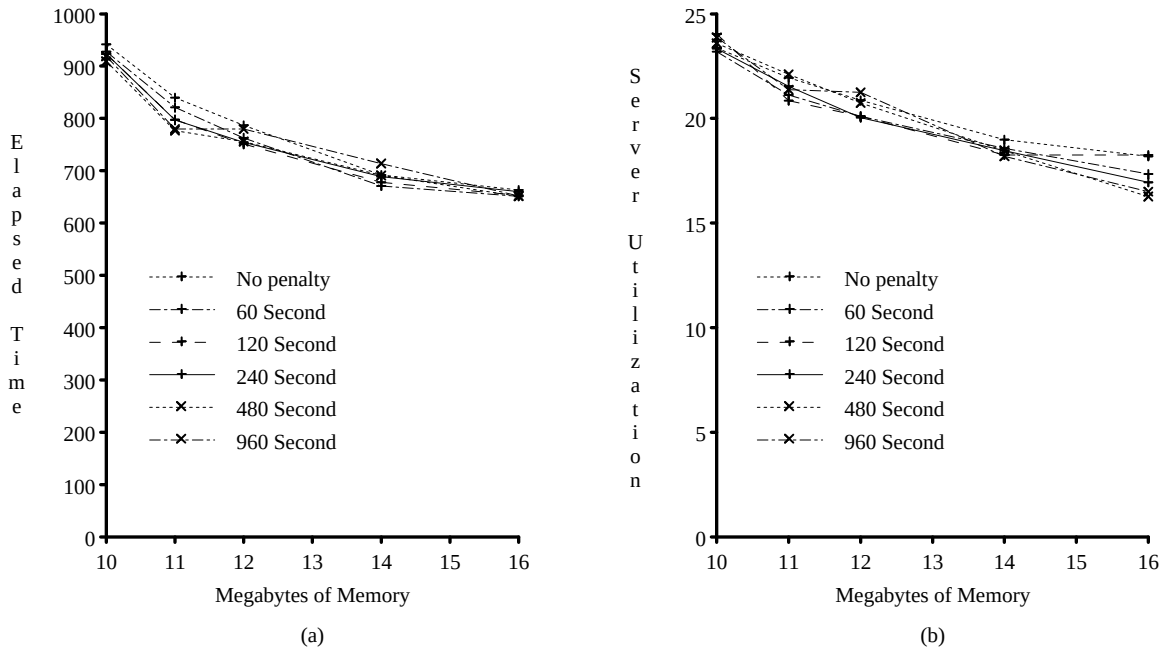

Figure 6-5. Elapsed time and server utilization with various penalties as a function of client physical memory size. In both graphs the X-axis is client memory size. In graph (a) the Y-axis is the number of seconds to execute the benchmark and in graph (b) the Y-axis is the percent of the server's CPU that was utilized while the client was executing the benchmark.

After implementing this simple mechanism for penalizing the file system, I wanted to see what effect it had on system performance. I first measured its effect using the same benchmark and configuration that I used in the previous section. I tried penalizing the file system from 60 seconds up to 960 seconds (longer than the life of the benchmark). In this section I will present only the most important results from the benchmark; see Appendix C for detailed results.

The results of this benchmark indicate that penalizing the file system has little or no effect on overall performance. Figure 6-5 shows that, regardless of the penalty, the elapsed time and server utilization are about the same. Figure 6-6 shows why the penalty has no effect. As the penalty is made larger, the virtual memory performance gets better and the file system performance worse. The result is that overall performance is about the same regardless of the penalty.

The interactive component of the edit-compile-debug benchmark is small; most of the time is spent in debugger initialization and in the compiler and linker. As a result, it cannot be used to provide a good measurement of the effects of the file system penalty on interactive response. In order to look at the impact of penalizing the file system on interactive response, I developed a benchmark which simulates concurrent interactive and file system activity; I will call this benchmark the IFS benchmark. The interactive component of the benchmark is a program which periodically touches many pages in its virtual address space. This simulates a user who is interacting with a program. Each time a user interacts with a program, the program must have its code, heap and stack pages memory-resident in order to give good interactive response. In fact, if the user is interacting with a program under a window system such as X11, then several programs

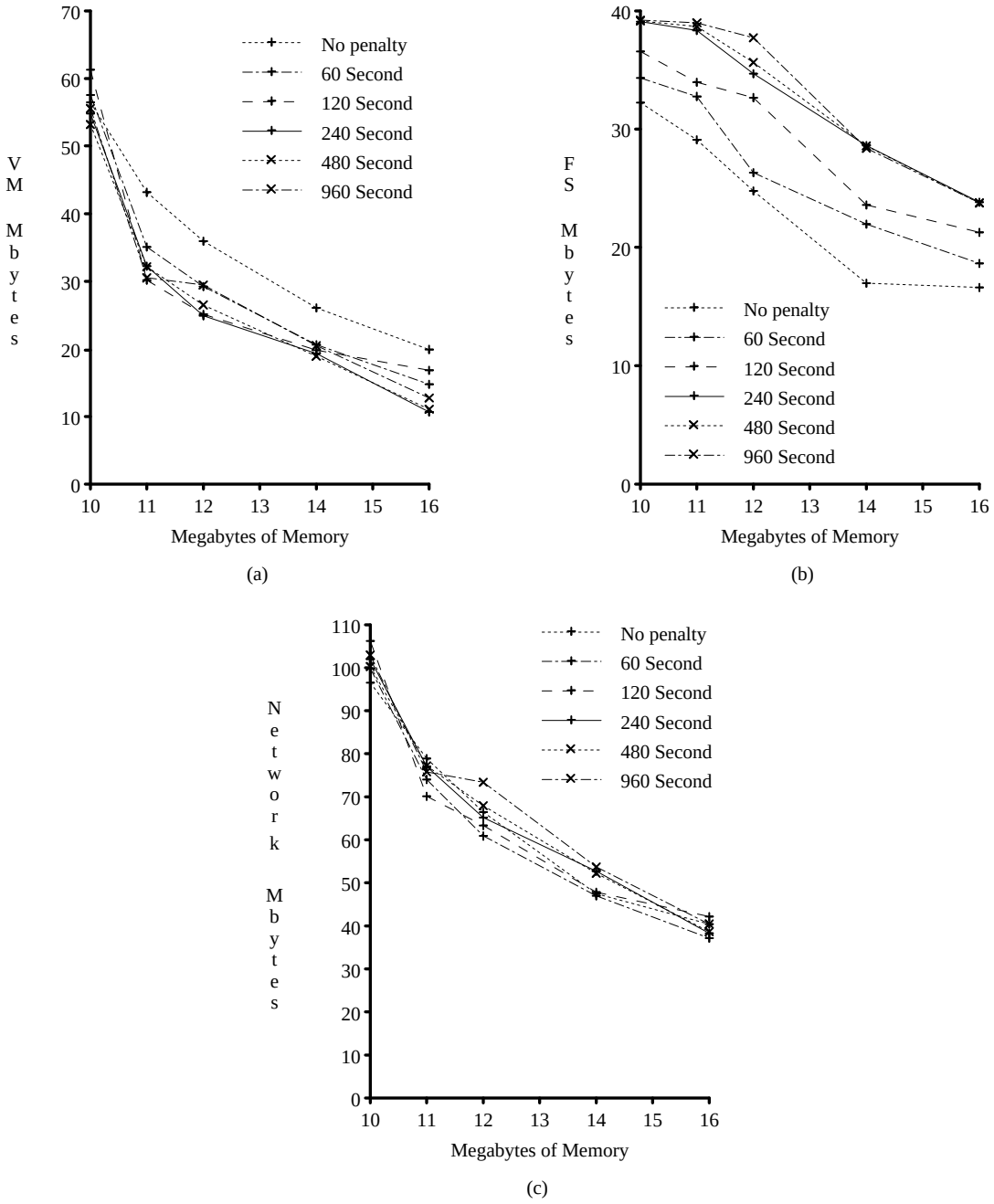

Figure 6-6. These graph gives the number of Mbytes transferred across the network with various penalties as a function of client memory size. In all three graphs the X-axis is the amount client memory and the Y-axis is the total number of Mbytes transferred during the benchmark. Graph (a) is virtual memory traffic, (b) is file system traffic and (c) is total network traffic.

may have to be memory-resident in order for the user to get good interactive response.

The file system component of the IFS benchmark is the Sort benchmark that was described in Chapter 4. I chose Sort because it is the benchmark that is most sensitive to changes in the file system cache size; of all benchmarks, its performance is most likely to degrade if the file cache size is reduced. The Sort program is run concurrently with the interactive program to simulate a file system program that attempts to grow its cache by stealing memory from an interactive program. Penalizing the file system should prevent Sort from stealing memory away from the interactive program, but it may cause the Sort program to degrade in performance.

The configuration that I used was an 8-Mbyte Sun-3/75 client and a 16-Mbyte Sun-3/180 server. 1.3 Mbytes of the 8 Mbytes on the client are used by the kernel, which leaves 6.7 Mbytes for user processes. I made the interactive program use 5.7 Mbytes of memory and left at most 1 Mbyte for Sort and the file system cache. I left only 1 Mbyte for Sort so that Sort will contend with the virtual memory system for memory.

The percentage of memory that the interactive program dirties each time it touches the memory in its address space may impact the performance of the IFS benchmark. In order to approximate the percentage of memory that an average program dirties, I measured the amount of dirty memory on 5 workstations running Sprite. Between 40 and 60 percent of the memory that was being used by user processes was dirty on these machines. Because of this result, I made the interactive component of the IFS benchmark dirty half of the memory that it touches.

Sleep Interval	No penalty					Penalty				
	Response Time			Sort Time		Response Time			Sort Time	
	Min	Max	Avg	Time	Deg	Min	Max	Avg	Time	Deg
1	0.0	4.7	0.1	79.6	33%	0.0	0.4	0.03	74.8	25%
5	0.0	4.5	0.8	83.8	40%	0.0	0.1	0.02	72.8	21%
10	1.9	13.3	5.9	103.4	72%	0.0	0.1	0.01	72.1	20%
30	12.5	22.0	15.8	96.3	61%	0.0	0.1	0.03	74.0	23%

Table 6-3. Response time and elapsed time for the IFS benchmark. Each data point is the average of the results from three runs of the benchmark. The first column gives the number of seconds that the interactive benchmark slept before touching all of its memory. Columns 2 through 6 give the results when the file system was not penalized. Columns 2 through 4 give the minimum, maximum and average number of seconds it took the interactive benchmark to touch all of its memory when it awoke from its sleep. Columns 5 and 6 give the total number of seconds it took to execute the Sort benchmark, and the amount of degradation relative to the best case given in Chapter 4 (60 seconds). The last five columns are the results when the file system was penalized by 120 seconds.

Tables 6-3 and 6-4 show the impact of the file system penalty on the performance of the IFS benchmark. Table 6-3 shows that, when the file system is not penalized, the response time has a high variance. Sometimes it is instantaneous and other times it can take up to 22 seconds. The response time gets worse when the interactive program touches memory less frequently. Short sleep intervals correspond, for example, to temporary pauses in an editing session. Longer sleep intervals correspond, for example, to windows that have been idle because the user was working in a different window. Longer sleep intervals allow the sort program to steal more memory (see Table 6-4). This causes the interactive program to wait for pages to get faulted in from the file server.

Table 6-3 shows that, when the file system is penalized, the interactive response is excellent. The 120-second penalty prevents the file system from taking any memory away from the virtual memory system. Thus, regardless of the amount of time that the

Sleep Interval	No penalty					Penalty				
	Faults		Page Outs	Cache Size		Faults		Page Outs	Cache Size	
	Total	Swap		Min	Max	Total	Swap		Min	Max
1	1217	173	456	152	784	783	1	4	64	178
5	1261	437	509	157	842	781	0	0	64	168
10	2481	1605	1177	146	1226	781	0	0	64	168
30	2097	1250	983	141	2533	782	0	0	64	168

Table 6-4. Cache size and page fault behavior for the IFS benchmark. Each data point is the average of the results from three runs of the benchmark. The first column gives the number of seconds that the interactive benchmark slept before touching all of its memory. Columns 2 through 6 give the results when the file system was not penalized. Column 2 is the total number of page faults that occurred, Column 3 is the number of faults from swap space, Column 4 is the number of pages that were written to swap space and Columns 5 and 6 give the minimum and maximum amount of memory in Kbytes that was resident in the cache during the benchmark. The last five columns are the results when the file system was penalized by 120 seconds.

interactive program pauses between successive touching of its memory, the response time is the same.

Surprising, the file system penalty actually improves the execution time of the Sort benchmark (see Table 6-3). Without the penalty, the benchmark takes up to 72% longer to execute than the best case given in Chapter 4. The performance degrades because the CPU is busy trying to fault in pages for the interactive benchmark; if the interactive benchmark is memory resident, then it utilizes very little of the CPU. When the file system is penalized, Sort takes only 25% longer than the best case. This degradation is nearly identical to the degradation shown in Chapter 4 when Sort was run using only a small cache.

As I mentioned earlier, I had the interactive component of the IFS benchmark dirty half of its memory. In order to determine the effect of the amount of dirty memory, I ran the IFS benchmark where it only dirtied 10% of its memory (see Table 6-5). A comparison of Tables 6-3, 6-4 and 6-5 shows that the amount of dirty memory

has only a very minor impact on the performance of the IFS benchmark.

The results of the benchmarks in this section show that penalizing the file system can improve interactive response without degrading overall system performance. In some cases, it can even make the performance of both file- and virtual-memory intensive programs better. However, it is not clear what the optimal penalty should be. The penalty should be large enough so that idle user programs that will be used in the near future will not get removed from memory, but not so large that the performance of the file system is degraded unnecessarily. The best value for the penalty will depend on the behavior of the users of the system. In Sprite we normally set the penalty to 20 minutes. This means that an interactive program's pages will not be reclaimed by the file cache until the program has been idle for 20 minutes.

Sleep Interval	Response Time			Sort Time		Faults		Page Outs	Cache Size	
	Min	Max	Avg	Time	Deg	Total	Swap		Min	Max
1	0.0	7.3	0.1	77.3	29%	1111	157	353	186	845
5	0.0	5.4	0.8	80.7	35%	1206	399	446	186	925
10	0.0	11.8	7.1	100.7	68%	2431	1576	890	178	1245
30	12.3	23.0	17.7	89.8	50%	1889	1088	798	128	2621

Table 6-5. Results from the IFS benchmark when only 10% of memory was dirtied and no penalty was used. Each data point is the average of the results from three runs of the benchmark. The first column gives the number of seconds that the interactive benchmark slept before touching all of its memory. Columns 2 through 4 give the minimum, maximum and average number of seconds that it took the interactive benchmark to touch its memory when it awoke from its sleep. Columns 5 and 6 give the total number of seconds that it took to execute the sort benchmark and the amount of degradation relative to the best case given in Chapter 4 (60 seconds). Column 7 is the total number of page faults that occurred, Column 8 is the number of faults from swap space, Column 9 is the number of pages that were written to swap space and Columns 10 and 11 give the minimum and maximum amount of memory in Kbytes that was resident in the cache during the benchmark.

Benchmark	Megabytes			Seconds			Degradation
	Read	Written	Total	Read	Written	Total	
Vm-make	9.20	2.39	11.59	3.31	1.14	4.45	1.5%
Andrew	7.62	3.12	10.74	2.74	1.48	4.22	1.6%
Sort	2.70	2.70	5.40	0.97	1.28	2.25	4.0%
Diff	2.00	0.00	2.00	0.72	0.0	0.72	19.1%
Ditroff	0.69	0.79	1.48	0.25	0.38	0.63	0.5%

Table 6-6. Cost of not using mapped files. The first column identifies the benchmark, the second column the total number of Mbytes read and written during the life of the benchmark, the third column the approximate number of seconds spent reading and writing the data and the last column the degradation of the benchmark compared to free reads and writes (no copying). The time spent copying was computed by taking the amount of data that was read and written, dividing it into 4 Kbyte blocks (the file system block size), and then multiplying it by the time to read and write 4 Kbyte blocks. On Sprite, 4-Kbyte data blocks can be read and written at rates of 2,912,711 bytes/second and 2,207,528 bytes/second respectively, or approximately 350us per Kbyte read and 463us per Kbyte written. Note that the reading and writing speeds given here are less than the ones given in Table 4-2 in Chapter 4. This is because the speeds in Chapter 4 were measured with larger blocks to get the maximum possible throughput.

6.6. Comparison to Mapped Files

One of the disadvantages of the Sprite variable-size cache mechanism is that it requires that data be copied between the user and kernel virtual address spaces during I/O. For example, when a user reads data from a file, the data is copied from the file system cache into a buffer in the user's address space. Mapped files eliminate these copy operations at the expense of an extra mapping cost. Table 6-6 contains the approximate performance penalty for using the Sprite mechanism instead of mapped files on the 5 benchmarks from Chapter 4. The results in Table 6-6 are a worst-case approximation of the Sprite penalty because they were calculated under the assumption that reads and writes are free under the mapped file scheme. The client degradation of Sprite's scheme in comparison to a mapped-file scheme ranges from 0.5 to 19.1 percent. However, except for the highly input-intensive Diff benchmark, the highest

degradation is only 4.0% and the average is only 1.9%. Thus, the extra copying with the Sprite mechanism has only a small effect, except for highly I/O-intensive benchmarks such as Diff.

6.7. Summary and Conclusions

In this Chapter I have demonstrated the effectiveness of variable-size file caches. Variable-size caches allow the amount of file data to grow for file-intensive programs, yet they work just as well as fixed-size caches for mixtures of file- and virtual-memory intensive programs; there is no fixed-size cache that can out-perform a variable-size cache.

I have also shown that a simple variable-size cache can be built that is not based on the mapped-file paradigm. The Sprite variable-size cache mechanism works by having the virtual memory system and file system negotiate over the use of physical memory. This allows Sprite to use a simple cache consistency mechanism and incurs no extra overhead on machines that make mapping expensive. The extra copying cost required in Sprite over mapped file schemes is small and does not noticeably degrade client performance over mapped-file schemes; if the mapping cost is high enough, then the Sprite mechanism will outperform mapped-file schemes.

The Sprite variable-size cache mechanism allows the file system to be penalized so that it will be more difficult for the file system to take memory from the virtual memory system. The use of the penalty appears to be effective in improving interactive response without degrading file system performance. The optimal value of the penalty is not yet clear; how much to penalize the file system will depend on the behavior of the

users of the system.

CHAPTER 7

Copy-on-Write For Sprite

7.1. Introduction

In systems that create new processes by forking, one of the major costs of process creation is copying the address space from the parent to the newly created child. A common method of improving the performance of process creation is by using copy-on-write: pages in the address space are initially shared by the parent and child; a page is not actually copied until one of the processes attempts to modify it. Copy-on-write saves not only copying of pages in memory, but also copying of pages that are on backing store. Copy-on-write has been implemented in several systems, with the earliest being TENEX [BBM72, Mur72] and one of the most recent being Mach [Ras87].

This chapter describes a simple copy-on-write mechanism that I have implemented as part of Sprite. It differs from other copy-on-write mechanisms in that it is actually a combination of copy-on-write (COW) and copy-on-reference (COR); for each page that is involved in copy-on-write activity, one segment has it copy-on-write and all other segments that reference it have it copy-on-reference. I chose the COW-COR mechanism for two reasons: virtually-addressed caches and simplicity. The SPUR hardware [Hil86], which is one of Sprite's target machines, uses virtually-addressed caches that do not provide efficient support for copy-on-write; expensive cache flushing operations are required in order to implement copy-on-write on a SPUR.

As I will explain later, the Sprite COW-COR scheme can be implemented on architectures such as a SPUR with less cache flushing overhead than a pure copy-on-write scheme.

The other major reason for using the Sprite scheme was simplicity. One of the major complexities of copy-on-write is handling the tree of descendants that results from a single parent. In the Sprite scheme, this potentially-complex tree structure is represented by a simple linear list. This simplification and others made the addition of copy-on-write to Sprite an easy task; the implementation was completed in less than one man-week.

In order to compare the Sprite COW-COR scheme to copy-on-fork schemes, I measured the performance of the Sprite COW-COR scheme by running benchmark programs against Sprite and by monitoring normal use of the system. The measurements indicate that the COW-COR mechanism can potentially improve fork performance over copy-on-fork schemes from 10 to 100 times depending on whether pages are resident in memory or on backing store. However, during normal use, the COW-COR mechanism provides a much smaller benefit: less than 30 percent of page copy operations are eliminated. Also, the 70% of the pages that are copied are copied at the expense of extra page faults. With the Sprite implementation, the overhead of handling the additional page faults results in worse overall performance than copy-on-fork; a more optimized implementation could provide more than a 20% improvement in performance over copy-on-fork. A pure copy-on-write scheme would eliminate 10 to 20 percent of the page copy operations required under COW-COR, and would provide up to a 20% improvement in fork performance over COW-COR. However, because of extra cache-

flushing overhead on machines with virtually-addressed caches, copy-on-write may have worse overall performance than COW-COR on these types of machines.

The rest of this Chapter is organized as follows: Section 7.2 gives a brief overview of the Sprite virtual memory system; Section 7.3 discusses previous work and Mach in particular; Section 7.4 describes the Sprite copy-on-write mechanism; Section 7.5 compares the Sprite scheme to a pure copy-on-write scheme; and Section 7.6 gives measurements of the performance of the Sprite scheme.

7.2. Sprite Virtual Memory

A Sprite process's virtual address space is divided up into three segments: code, heap and stack. Each segment has its own page table that describes the segment's virtual address space, and each segment has its own file that is used for backing store. Segments can be shared by different processes. When a process is forked using a copy-on-fork mechanism, a) the child will share the parent's code segment read-only, b) the child is given a copy of the stack segment, and c) the heap segment is either write-shared or a copy of it is given to the child. A copy-on-write mechanism has the potential of saving the actual copying of pages in the stack and heap segments.

The most common scenario where copy-on-write may be helpful is the fork-exec sequence. This is the case where a parent creates a child with the *fork* system call and then the child immediately replaces the address space that it shares with its parent with a new address space with the *exec* system call. This happens, for example, in the UNIX shells for each command executed. If neither the parent nor the child modify many pages between the *fork* and the *exec*, then copy-on-write may be able to save many page

copy operations.

7.3. Previous Work

The original idea of copy-on-write emerged over 15 years ago with TENEX [BBM72, Mur72]. Since then it has been implemented in several systems [Akh87, GMS87, Ras87, SCC86]. The Mach operating system [Ras87] is one of the most recent systems to implement copy-on-write, and is one of the few whose implementation of copy-on-write has been published in detail. Copy-on-write is an integral part of Mach; it is the basis for both efficient message transmission and efficient process creation. This section briefly describes the Mach implementation of copy-on-write as it pertains to process creation.

A Mach process's address space is defined by an *address map* which is a linked list of references to memory objects. When a process forks, the memory objects are ''copied'' using copy-on-write. This is done by making the address maps of the parent and child point to the same memory objects. When a page in the copied memory object is written, a new page is given to the process that wrote the page. In order to hold new pages that are copied because of a copy-on-write fault, Mach creates an object called a *shadow object*. Those pages that are modified are copied to the shadow object and unmodified pages are kept in the original object; pages only have to be copied if they are modified.

The complexity that arises in the Mach scheme is that a shadow object may itself be shadowed as a result of a copy-on-write copy operation. This can result in an entire chain of shadow objects being created (see Figure 7-1). In order to satisfy a page fault,

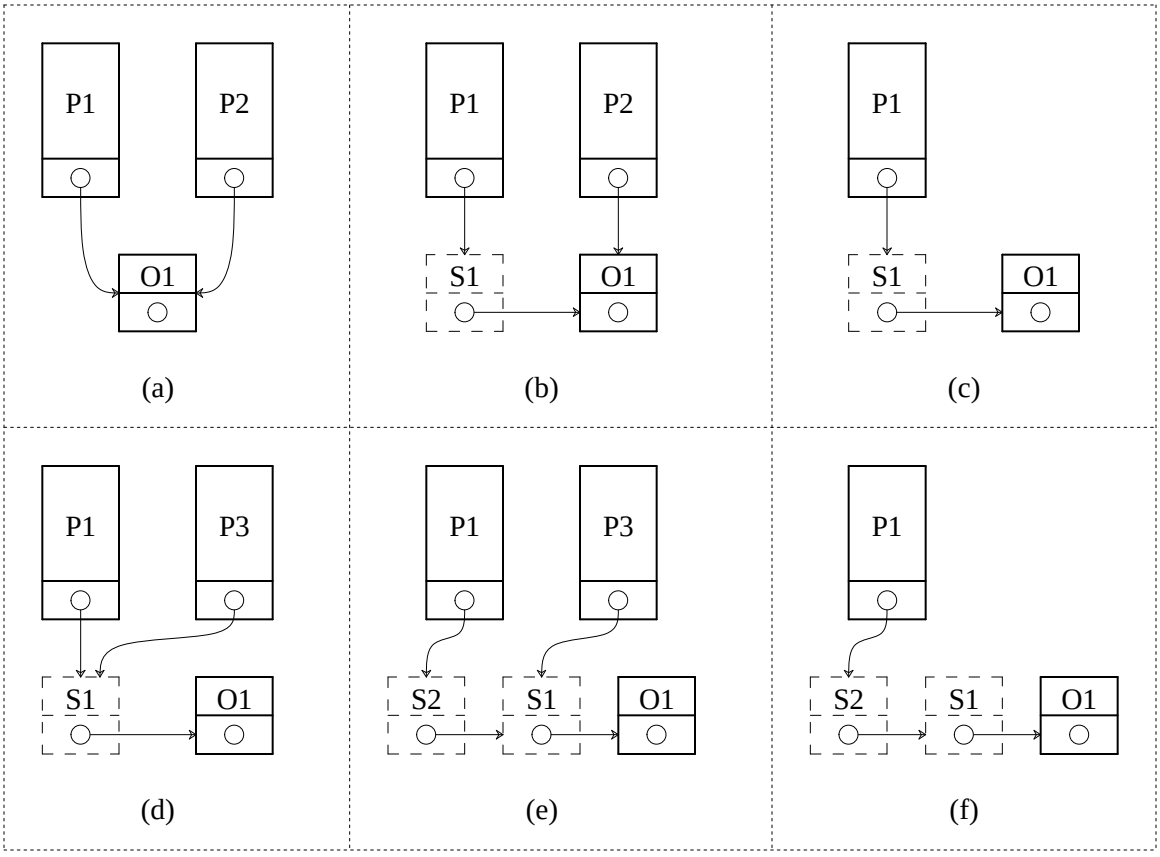

Figure 7-1. Mach copy-on-write. In (a) process P1 forks creating process P2. They both share object O1. In (b) P1 modifies a page and gets a shadow object to hold the modified page. P1's address map points to the shadow object which in turn points to the original object. In (c) P2 exits leaving P1 with the chain of two objects. In (d) P1 forks P3. They both share the object O1 and the shadow object S1. In (e) P1 modifies a page in either O1 or S1 and gets a new shadow object S2. Now P2 has a chain of 3 objects: two shadow objects and the original object. When P3 exits in (f), P1 is still left with the 3 objects. However, by recognizing that the shadow objects completely overlap the original object, the extraneous shadow objects can be eliminated.

the list of shadow objects and then possibly the original object need to be searched to find the data for the page. Much of the complexity involved in Mach memory management is involved in preventing long chains of shadow objects [Ras87]. In particular, the extraneous shadow objects shown in Figure 7-1 that are left over after a child exits can be eliminated by moving the pages in the shadow objects into the original object.

7.4. Sprite COW-COR

The Sprite copy-on-write scheme was designed with a more restrictive set of goals than Mach's. The goals behind the Sprite design were:

- To make process creation efficient in a UNIX-like environment.
- To be able to run as efficiently as possible on machines such as a SPUR that have virtually-addressed caches.
- To yield as simple an implementation as possible.

In particular, Sprite's copy-on-write scheme does not participate in the implementation of message communication; this simplified the design constraints in comparison to Mach.

7.4.1. Overview

Sprite uses a combination of copy-on-write and copy-on-reference, as illustrated in Figure 7-2. For each page that is involved in copy-on-write activity, one segment (called the *master segment*) has the page marked copy-on-write and all other segments that reference the page (called *slave segments*) have it marked copy-on-reference. When a process forks, the segment in the parent process becomes the master segment and the segment in the child becomes the slave segment. All pages in the master segment are marked copy-on-write and made read-only. All pages in the slave segment are marked copy-on-reference and made inaccessible.

A copy-on-reference fault occurs when a copy-on-reference page is referenced. When the fault occurs, the master copy-on-write page is located, and a copy is made for

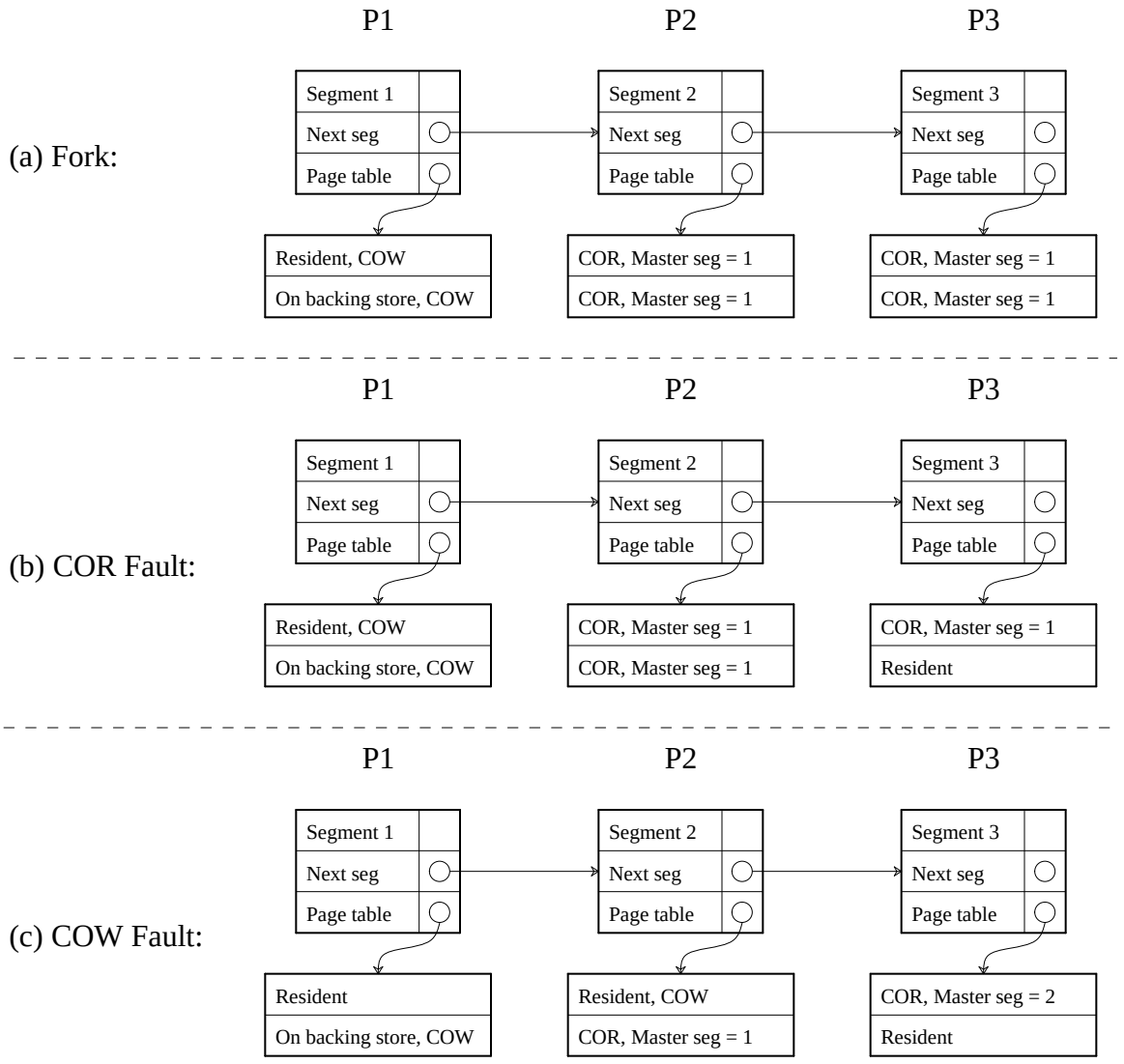

Figure 7-2. Sprite copy-on-write. In (a) the process (P1) that owns segment 1 forks two children and creates two copy-on-reference copies, segments 2 and 3, which are owned by processes P2 and P3 respectively. The page table entry (PTE) for each of the COR pages names the segment with the COW copy. In (b) P3 references the second page in segment 3 and a copy of the page is loaded into segment 3 from S1's swap file. The copy is made readable and writable. In (c) P1 modifies the first page in segment 1 and gives a new COW copy to segment 2. Segment 3's PTE is updated to point to segment 2 and segment 1's page is made readable and writable.

the slave segment. In order to allow the master copy of the page to be easily located, the page table entry for each copy-on-reference page names the master segment for the page (as shown below, different pages may reference different master segments).

When a process attempts to modify a copy-on-write page (call it A), a copy-on-write fault occurs. A copy-on-write fault is more complex than a copy-on-reference fault because a new copy-on-write master segment for the page must be found so that the master segment can modify its copy of the page. This new copy-on-write master must be one of the slave segments. In order to allow a slave segment to be easily located, the master segment and each of its slave segments are linked together in a list; a master can have multiple slave segments if a parent forks multiple children. The new master segment is found by searching the list of segments for a slave segment that contains a page that is copy-on-reference off of A. This slave segment is given a copy-on-write copy of A (call it B). All of the remaining segments that have pages that were copy-on-reference off of A must now be changed to reference B as their master. This is done by searching the list and updating the page table entries of each segment that was copy-on-reference off of A to point to the new master segment.

When a segment is deleted because a process exits or execs, copy-on-write dependencies in the deleted segment need to be eliminated. Pages that are copy-on-write must be copied to another segment. Each copy-on-write page (call it A) in the deleted segment is copied to another segment that contains a page (call it B) that used to be copy-on-reference off of A. If A is resident in memory, this is done by remapping the page in A onto B. Otherwise, the backing store for A is copied to B's backing store. Copy-on-reference pages in the deleted segment are ignored; this may cause extraneous copy-on-write page faults and is discussed below. Once all copy-on-write dependencies are eliminated, the segment is deleted from the linked list.

7.4.2. Trees of Descendants

The previous section only mentioned COW-COR for a single parent with multiple children. However, if processes with copy-on-write slave segments fork, then a tree of copy-on-write and copy-on-reference relationships will result. Rather than build a tree-like data structure to represent the relationships, Sprite puts all of the related segments in the same linked list. This can be done because the page table entry for each copy-on-reference page names the segment that contains the master copy. This provides a simpler implementation and is based on the assumption that the lists will rarely contain more than a few segments. The lists should be short because in a UNIX-like environment processes normally replace their address space by calling the *exec* system call soon after they are created; the benchmark results in Section 7.6.2 validate this assumption.

One difference between the Sprite and Mach mechanisms is that, when a page fault of any type occurs, the location of the master copy of the page is immediately known; no chain of objects needs to be traversed. However, Sprite does need to traverse its linked list of segments for other reasons, as described above and below.

7.4.3. Eliminating Extra Copy-on-Write Faults

After a segment is deleted, or a copy-on-write fault or a copy-on-reference fault is handled, there can be pages marked copy-on-write for which there is no longer a corresponding copy-on-reference page. The easiest method of handling this problem is to cleanup extraneous copy-on-write pages when they are faulted on. However, because a copy-on-write fault is fairly expensive, the Sprite implementation of COW-

COR checks for the common causes of extra faults and eliminates them. For example, after a copy-on-reference fault is handled on a page, the master may be the only segment that references the COW copy of the page; this case is detected and the page is made writeable by the master.

Another common cause of extra page faults is segment deletion. After a copy-on-reference segment is deleted, the master may be the only segment left in the list of copy-on-write and copy-on-reference segments. Since, as explained above, copy-on-reference pages in a deleted segment are ignored, this potentially leaves copy-on-write pages for which there is no copy-on-reference page. However, this case is detected and, when there are no longer any slaves off of a master, all of the pages in the master are made writeable.

7.4.4. Backing Store

The backing store for each copy-on-write page is the master segment's backing store file. When a copy-on-reference fault occurs for a page that is on backing store and not resident in memory, the page is read from the master segment's backing store file. Copy-on-write faults can only occur to pages that are memory resident. If a process attempts to modify a copy-on-write page that is not memory resident, then a normal page fault occurs instead of a copy-on-write fault. Once the faulting process continues, it will try to modify the now-memory-resident page and a true copy-on-write fault will occur. This second fault could be eliminated by slightly complicating the implementation, but the cost of the extra fault is very small in comparison to the cost of loading a page from backing store.

7.5. Comparison of Sprite Scheme and Shadow Objects

Besides using a combination of copy-on-write and copy-on-reference instead of pure copy-on-write, the major difference between Sprite and Mach is that Sprite does not use shadow objects. The method that Sprite uses to implement COW-COR could also be used to implement a pure copy-on-write scheme. The difference would be that, when a process forks, each memory resident page would be marked copy-on-write in both the parent and the child segment's page tables, instead of copy-on-write in the parent and copy-on-reference in the child. For pages that are only resident on backing store, the page table entry of the child would be used to point to the parent segment since the parent has the swap file.

The main advantage of the Sprite method of implementation of copy-on-write is that it eliminates the potential to create chains of extraneous objects. For example, Figure 7-3 shows what happens under the Sprite scheme when a parent forks a child, the parent modifies a page, and then the child exits. The result is that, after the child exits, the Sprite scheme automatically cleans up the list; there are no extra structures to maintain or collapse.

The disadvantage of the Sprite scheme is that it can require extra copying of pages when a parent exits before its child exits. With shadow objects, no copy operations are required when a process exits, because shadow objects can exist even after the process that created them has exited. However, under the Sprite scheme, when a segment is deleted, all copy-on-write pages must be copied to another segment. In a normal UNIX environment parents usually wait for their children to exit, so in practice the Sprite

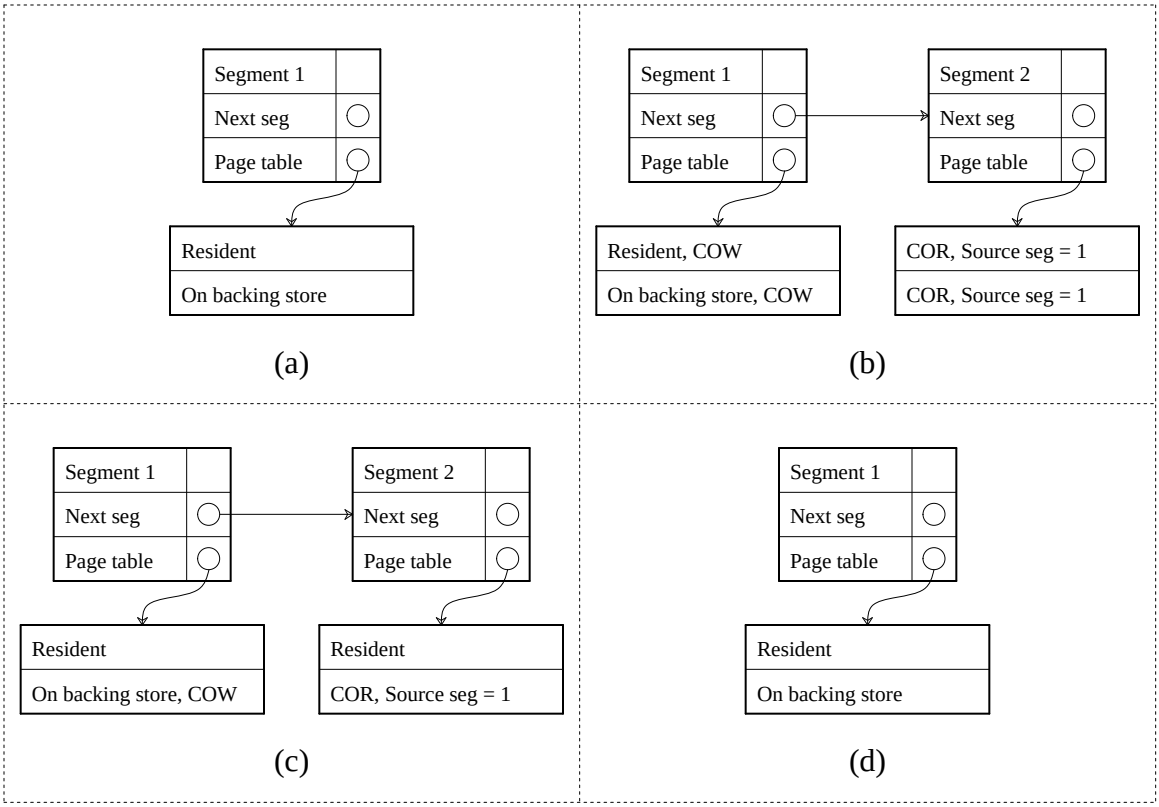

Figure 7-3. In (b) the process (call it P1) that owns segment 1 forks a child and creates a copy-on-reference copy, segment 2, which is owned by process P2. In (c) P1 modifies one of the copy-on-write pages in segment 1 and gives a copy of the page to segment 2. In (d) P2 exits causing segment 2 to be deleted. When segment 2 is deleted, it is removed from the list and the lone copy-on-write page left in segment 1 is made readable and writable. The result is that the state of segment 1 is restored back to how it was in (a); that is, the state before segment 1 was created.

scheme should perform as well as the shadow object scheme.

7.6. Copy-on-Write Performance

I ran benchmark programs and measured normal use of Sprite in order to answer several questions about the performance of the Sprite COW-COR scheme:

- What is the maximum potential benefit from COW-COR, compared to no copy-on-write mechanism at all?

- What is the actual benefit from COW-COR during normal use, compared to no copy-on-write mechanism at all?
- How does COW-COR compare to a pure copy-on-write scheme?
- How much more efficiently can COW-COR be implemented on a SPUR than a pure copy-on-write scheme?

The benchmark programs are UNIX programs that have been converted to run on Sprite, and the results obtained from the measurements of Sprite should be applicable to any UNIX-like operating system. The measurements were taken on a Sun-3/75 workstation with 16 Mbytes of memory, 8-Kbyte pages and about 2 MIPS processing power. The Sun-3/75 does not have a CPU cache.

7.6.1. Raw Performance

I used a simple benchmark to determine the maximum benefit attainable from COW-COR during process creation. This benchmark forks a child and then waits for the child to exit. The amount of memory that the parent has resident in memory or on backing store when it does the fork can be varied. It is an optimistic measurement of the benefit of copy-on-write because none of the pages are referenced or modified by the parent or the child. Table 7-1 gives the results.

There are two interesting results from this benchmark. First, forks are substantially faster under the COW-COR scheme than they are with copy-on-fork schemes: more than 10 times faster for processes with large amounts of resident memory and more than 100 times faster for processes with large amounts of memory on backing

	COW-COR		Copy-on-Fork	
Kbytes	Mem-res	Backing Store	Mem-res	Backing Store
0	22.8ms	22.8ms	22.5ms	22.4ms
64	24.7	24.0	59.7	171.7
128	25.8	24.3	79.6	265.0
256	28.0	24.6	119.4	457.6
512	32.3	25.3	199.0	850.8
1024	41.1	26.8	358.6	1635.1
2048	58.7	29.7	677.2	3209.0

Table 7-1. Raw Sprite COW-COR performance. This table gives the time in milliseconds required per execution of a *fork* and *wait* call in the parent and an *exit* call in the child as a function of segment size. These measurements were taken on a Sun-3/75. The first column gives the number of Kbytes that were either memory resident or on backing store when the parent forked. The second and third columns are the performance with COW-COR and the fourth and fifth columns are without COW-COR (i.e., all of the data had to be copied at fork time). "Mem-res" means that all of the bytes were memory resident and "Backing Store" means that all of the bytes were on backing store.

store. Thus, as expected, if processes with large amounts of memory fork and do not reference many pages, copy-on-write can substantially improve fork performance. Second, it is slower to fork a process when all of its pages are memory resident than when all of its pages are on backing store. This is because the hardware protection must be changed to make memory resident pages copy-on-write.

7.6.2. Realistic Performance

The benchmark described in the previous section gave a best-case scenario for the COW-COR mechanism: a large process forks and does not reference any of its memory. In order to make a more realistic determination of the benefits of the COW-COR mechanism over traditional copy-on-fork schemes, I measured a file system benchmark program, an edit-compile-debug benchmark and several days' work of two different Sprite designers. Table 7-2 describes the benchmarks and Table 7-3 gives the

results.

One interesting result from Table 7-3 is that the number of times pages were marked copy-on-write was about the same as the number of times pages were marked copy-on-reference. This implies that in general there is only one segment that has any

Benchmark	Description
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 (same as used in Chapters 4 and 5).
ECD	An edit-compile-debug benchmark run under the X11 window system (same as used in Chapter 6).
User-A	Several days' work of a Sprite system designer using Emacs under the X11 window system. Work involved editing, compiling and other miscellaneous activities.
User-B	Several days' work of a Sprite system designer using typescript windows and a window-based editor under the X11 window system. Work involved editing, compiling, debugging and other miscellaneous activities.

Table 7-2. Sprite COW-COR benchmarks.

given page mapped COR; if multiple segments had pages mapped COR, then there would have to be more copy-on-reference pages than copy-on-write pages. Therefore, the COW-COR lists should normally contain only two segments and the extra overhead required to traverse the list on copy-on-write and copy-on-reference faults should be small.

Perhaps the most interesting result in Table 7-3 is that, under normal use, COW-COR saves less than 30% of the page copy operations that would be required under a copy-on-fork scheme. Furthermore, copy-on-write schemes require additional page faults that would not occur otherwise; as the cost of a page fault increases, the benefits of COW-COR will diminish. I determined from measurements of Sprite that a page fault takes 1.1 milliseconds on a Sun-3/75 workstation. In addition, from Table 7-1 it can be calculated that the cost of a copy operation is approximately 2.5 milliseconds.

	COW Pages	COR Pages	Faults			Copies Saved
			COW	COR	% of Total	
Andrew	2846	2846	1%	71%	26%	28%
ECD	1430	1448	5%	72%	15%	22%
User-A	38771	40231	8%	63%	30%	28%
User-B	109965	112257	6%	66%	23%	27%

Table 7-3. Sprite COW-COR performance under more realistic conditions. This table gives Sprite COW-COR statistics for the two benchmarks and the two measurements of user activity. The second and third columns are the number of times a page was marked copy-on-write and copy-on-reference by processes forking. Columns four and five are the percentages of copy-on-write and copy-on-reference pages that actually generated faults. The sixth column gives the percentage of the total number of faults taken during the benchmark that were copy-on-write and copy-on-reference faults. Finally, the last column indicates how many page copies were saved by COW-COR relative to a copy-on-fork scheme.

Thus, in Sprite a page fault costs nearly half as much as a copy operation. Figure 7-4 shows that, with this fault cost, COW-COR provides slightly worse performance than copy-on-fork.

The fault cost in Sprite is much higher than the fault cost in the Mach operating system [Ras88]. In Mach the page fault cost is less than 10% of the copying cost. If Sprite were able to attain the same low fault cost as Mach, forks would be 15 to 20 percent faster with COW-COR than with copy-on-fork. Thus, with a highly optimized page fault handler, COW-COR can provide a moderate performance improvement over copy-on-fork schemes.

7.6.3. COW-COR vs. Pure Copy-on-Write

Nearly all of the faults that occurred during the benchmarks and normal use were copy-on-reference faults. If a pure copy-on-write mechanism could eliminate these faults, then it would provide much better performance than the COW-COR scheme. However, for the two benchmarks and normal use, between 80 and 90 percent of those pages that were copied because of copy-on-reference faults were eventually modified (see Table 7-4). Thus, a pure copy-on-write scheme has only a small advantage over the COW-COR scheme: only between 10 and 20 percent of the page copy operations required under COW-COR would be eliminated.

Figure 7-4 shows the performance improvements possible on a Sun-3 with a pure copy-on-write scheme. With the high Sprite fault cost, a pure copy-on-write scheme provides a 5 to 20 percent improvement over copy-on-fork schemes and a 10 to 20 percent improvement over the Sprite COW-COR scheme. With the low Mach fault cost,

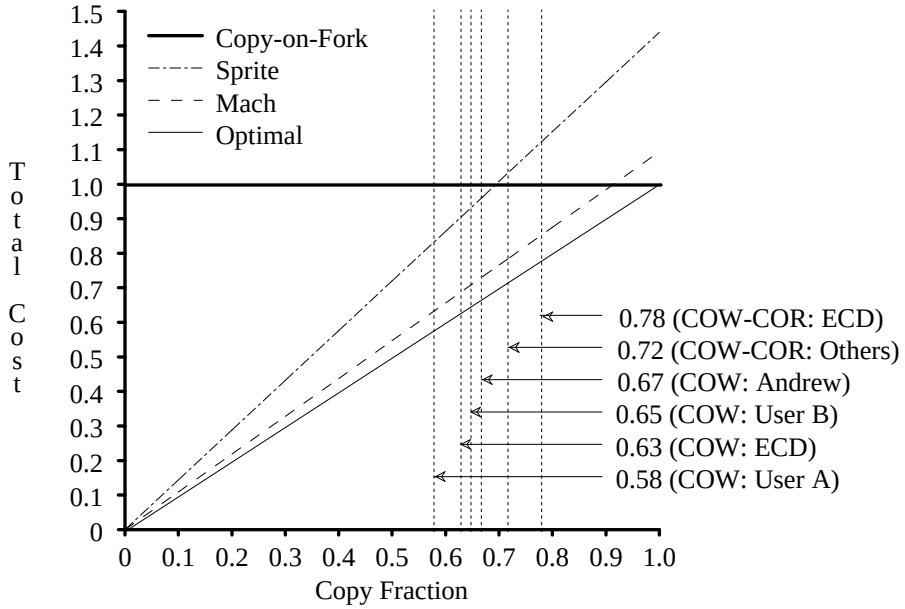

Figure 7-4. Total cost of handling fork-related page copying on Sun-3's as a function of the fraction of pages copied because of copy-on-write or copy-on-reference faults. A total cost of 1.0 corresponds to the cost of copying all pages at fork time. The optimal line represents the cost when the time required for each copy-on-write or copy-on-reference fault is 0, the Mach line when each fault is 0.234 milliseconds (9% of the cost of copying a page), and the Sprite line when each fault is 1.1 milliseconds (44% of the cost of copying a page). The 2 rightmost vertical lines correspond to the fraction of pages copied with the COW-COR mechanism for the benchmarks and the 4 leftmost vertical lines represent the fraction of pages that would have been copied with a pure copy-on-write mechanism. The vertical line marked ''Others'' is the copy fraction for Andrew, User A and User B.

copy-on-write provides fairly substantial improvements over copy-on-fork schemes. Thus, with an optimized fault handler, copy-on-write reduces the fork cost by 30 to 40 percent over copy-on-fork schemes and by 10 to 20 percent over optimized COW-COR schemes.

7.6.4. Cost of Virtually Addressed Caches

As mentioned earlier, one of the potential advantages of the Sprite COW-COR scheme over a pure copy-on-write scheme is that it may reduce overhead on

	COW Faults	COR Faults	COR Modified	Pure-COW Faults
Andrew	1%	71%	93%	67%
ECD	5%	72%	81%	63%
User-A	8%	63%	79%	58%
User-B	6%	66%	90%	65%

Table 7-4. COW-COR vs. Copy-on-Write. This table gives the number of page faults that would occur under a pure copy-on-write scheme for the two benchmarks and the two measurements of user activity. The second and third columns show the percentage of copy-on-write and copy-on-reference pages that actually generated faults. The fourth column gives the percentage of those pages that were copied because of copy-on-reference faults that were eventually modified; all of the copy-on-reference pages that were eventually modified would have had to be copied under a pure copy-on-write scheme. The last column is the percentage of pages that would have been copied under a pure copy-on-write scheme; it is the second column added to the product of the third and fourth columns.

architectures with virtually addressed caches, such as the Sun-3 [SSS85], Sun-4 [Kel86] and SPUR [Hil86] architectures. In these machines, protection bits are stored along with the data in individual cache lines. To change the protection on a page, the operating system must first modify the page table entry, then flush all of the page's lines from the cache. When the lines are re-loaded into the cache, their protection bits will be set from the new page table entry.

When a process forks, all of its pages will have to be flushed from the cache in order to mark them read-only. This flush must occur in either a pure copy-on-write scheme or in Sprite's COW-COR scheme. In addition, whenever a copy-on-write page is made writable again, it will have to be flushed from the cache again. Once again, this will occur in both schemes. However, Sprite's mechanism allows a copy-on-reference page to be made accessible without any cache flushes: since the page was not previously accessible, there will be no data from it in the cache. On average, Sprite's

COW-COR mechanism will require 2 flushes per page (one at the time of the fork and another one later, when the parent's page eventually becomes writable again), while a pure copy-on-write scheme will require about 2.6 on the average (one at the time of the fork, another one when the parent's page becomes writable again, and a third one on the 58 to 67 percent of the pages that resulted in copy-on-write faults in the child).

The actual number of cache flushes required will be smaller on architectures such as the SPUR and the Sun-4, that use direct-mapped caches. With direct-mapped caches, the act of copying a page between virtual addresses that have the same offset within the cache will flush the source of the copy from the cache (the destination data will replace the source data in the cache because they will both map to the same address in the cache). Under COW-COR, over 70% of pages are copied. As a result, an average of only 1.3 cache flushes per page will be required with a direct-mapped cache (one at the time of the fork and another one on the 30% of the parent's pages that remain in the cache). A pure copy-on-write scheme copies over 60% of the pages. This will give an average of 2.0 flushes per page (one at the time of the fork, another one on the 40% of the parent's pages that remain in the cache, and a third one on the 58 to 67 percent of the pages that resulted in copy-on-write faults in the child).

Although Sprite's mechanism reduces cache flushing relative to pure copy-on-write schemes, the overhead may still be quite high. For example, if the cost of flushing a page is half as great as the cost of copying it, then any copy-on-write scheme will be at least as expensive as a copy-on-fork mechanism, even if none of the copied pages are ever accessed (unless the pages are on backing store). Since the Sprite COW-COR mechanism has not yet been ported to a machine with a virtually addressed cache, I

have no measurement of the impact of cache flushing on fork performance. However, I can estimate the impact of cache flushing on fork performance for the SPUR and Sun-4 architectures. Because the actual performance on a SPUR and a Sun-4 is dependent on numerous variables, including the cache miss ratio and the percentage of data that is modified, it is impossible to derive the exact fork cost without actually measuring it. However, the worst and best case performance can be easily calculated.

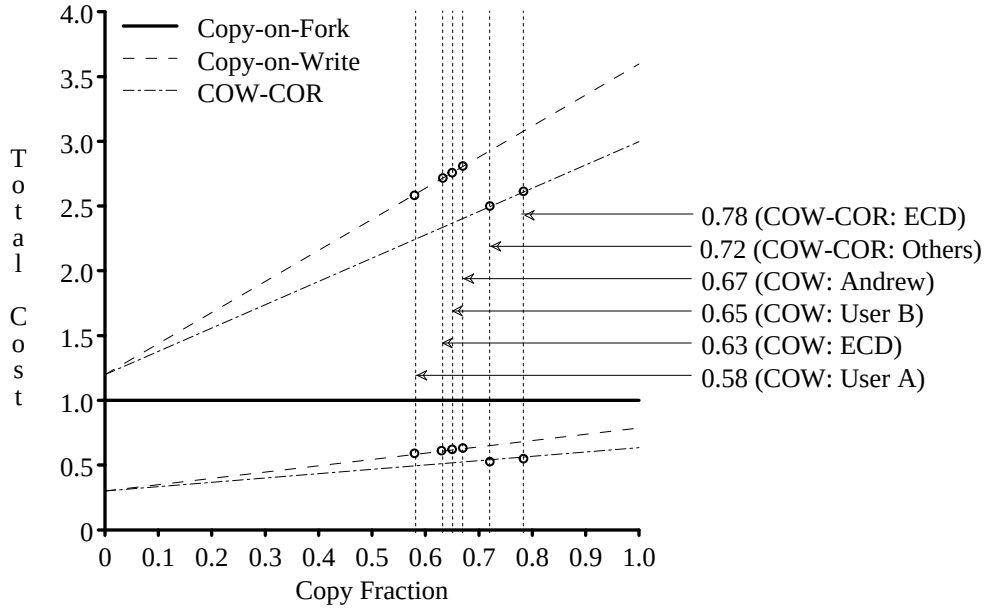

Figure 7-5. Total estimated cost of fork-related page copies on a SPUR, as a function of the fraction of page copies and cache flushes because of copy-on-write or copy-on-reference faults. The attributes of the SPUR architecture that were used to compute the curves in the graph are given in Table 7-5. The lower lines of the graph are best-case scenarios for copy-on-write and COW-COR and the upper two lines worst-case scenarios. The best-case combines the lowest possible flush cost and copy cost for copy-on-write and COW-COR and the highest possible copy cost for copy-on-fork. The worst-case is when all of the data for each page is present in the cache and clean at fork time and when the highest possible flush and copy costs occur for copy-on-write and COW-COR at other times. A cost of 1.0 corresponds to the cost of copying every page at fork time (copy-on-fork). The 6 vertical lines are the copy fractions for the 4 benchmarks. The vertical line marked ‘‘Others’’ is the copy fraction for Andrew, User A and User B.

Figure 7-5 gives the worst case and best case performance of copy-on-write and COW-COR on a SPUR. The computation of the performance is a complex one that involves the percentage of data that is resident in the cache during copy and flush operations, and the percent of cache memory that is dirty (see Table 7-5 for the SPUR attributes that were used in the computation, and Figure 7-5 for more explanation of the computation). In both the best and the worst case, COW-COR is strictly better than pure copy-on-write; this shows that COW-COR may be a reasonable alternative to pure

SPUR Attributes		
Page Size		4096 Bytes
Cache Line Size		32 Bytes
Copy cost per cache line	Copy data	12 Cycles
	Cache read miss	23 Cycles
	Write-back data	22 Cycles
	Minimum cost	35 Cycles
	Maximum cost	80 Cycles
Flush Cost	Read tags	12 Cycles
	Flush clean line	9 Cycles
	Flush dirty line	25 Cycles
Page Fault Cost		500 Cycles

Table 7-5. Attributes of the SPUR architecture [Woo88]. When a cache line is copied at fork-time or because of a copy-on-write or copy-on-reference fault, the destination of the copy will not be present in the cache. As a result the SPUR hardware will consider this a cache miss and fetch the destination from memory even though it is being totally overwritten. Hence the minimum copy cost is 35 cycles (12 for the copy and 23 to fetch the destination cache line). The maximum cost occurs when the source of the copy is not present in the cache (an additional 23 cycles) and the cache line that is being replaced must be written back (an additional 22 cycles). The cost of a cache flush ranges from 12 cycles if the line being flushed is not in the cache, to 37 cycles if the line being flushed is dirty. The fault cost is extrapolated from the Mach fault cost of approximately 500 instructions on a Sun-3 (234 microseconds on a 2 MIP machine). Since each instruction on a SPUR takes one cycle, the fault cost is assumed to be 500 cycles. This fault cost is merely a rough estimate and will be higher when cache behavior is taken into account.

copy-on-write for architectures like a SPUR with virtually-addressed caches. In addition, in the best case, COW-COR and pure copy-on-write can cut the fork cost by a factor of 2 over copy-on-fork schemes, but in the worst case the fork cost is up to three times higher with COW-COR or pure copy-on-write. Thus, although COW-COR and copy-on-write can potentially give a substantial performance improvement over copy-on-fork schemes, they can potentially give an even more substantial performance degradation.

The Sun-4 architecture has a much lower cache flushing cost than a SPUR (see Tables 7-5 and 7-6). Figure 7-6 gives the worst case and best case performance of copy-on-write and COW-COR on a Sun-4. Because of the lower cache flushing cost on a Sun-4, COW-COR does not perform as well in comparison to pure copy-on-write schemes as it does on a SPUR; COW-COR is slightly better than copy-on-write in some cases and slightly worse in others. In addition, the degradation of both COW-COR and copy-on-write in relation to pure copy-on-fork schemes is not as severe in the worst case. However, even with the low cache flushing cost, the fork cost can still be up to twice as high as copy-on-fork schemes.

The main conclusion that can be drawn from this discussion is that copy-on-write mechanisms may not be worthwhile in architectures with virtually-addressed caches. The actual performance advantages of copy-on-write will depend on program behavior, the flush and copy costs of the architecture, and the actual implementation of the copy-on-write mechanism. Although Figures 7-5 and 7-6 were derived for specific architectures and the Sprite implementation, a similar graph could be easily drawn to determine

Sun-4 Attributes		
Page Size		8192 Bytes
Cache Line Size		16 Bytes
Copy cost per cache line	Copy data	10 Cycles
	Cache read miss	10 Cycles
	Write-back data	8 Cycles
	Minimum cost	20 Cycles
	Maximum cost	38 Cycles
Flush Cost	Flush clean line	3 Cycles
	Flush non-consecutive dirty lines	3 Cycles
	Flush consecutive dirty lines	8 Cycles
Page Fault Cost		500 Cycles

Table 7-6. Attributes of the Sun-4 architecture [Kel88]. When a cache line is copied at fork time or because of a copy-on-write or copy-on-reference fault, the destination of the copy will not be present in the cache. As a result the Sun-4 hardware will consider this a cache miss and fetch the destination from memory even though it is being totally overwritten. Hence the minimum copy cost is 20 cycles (10 for the copy and 10 to fetch the destination cache line). The maximum cost occurs when the source of the copy is not present in the cache (an additional 10 cycles) and the cache line that is being replaced must be written back (an additional 8 cycles). The cost of a cache flush ranges from 3 cycles if the line being flushed is not in the cache, to 8 cycles if consecutive dirty lines are being flushed; it only takes 3 cycles to flush a dirty cache line as long as the next line in the cache is clean. The fault cost is extrapolated from the Mach fault cost of approximately 500 instructions on a Sun-3 (234 microseconds on a 2 MIP machine). Since each instruction on a Sun-4 takes one cycle, the fault cost is assumed to be 500 cycles. This fault cost is merely a rough estimate and will be higher when cache behavior is taken into account.

the potential benefit of copy-on-write mechanisms for any architecture and implementation.

7.6.5. Effect of Page Size

One reason why I measured only a small benefit from COW-COR under normal use may be that the measurements were made on a machine with large pages (8 Kbytes). If the page size were smaller, several changes in COW-COR behavior would

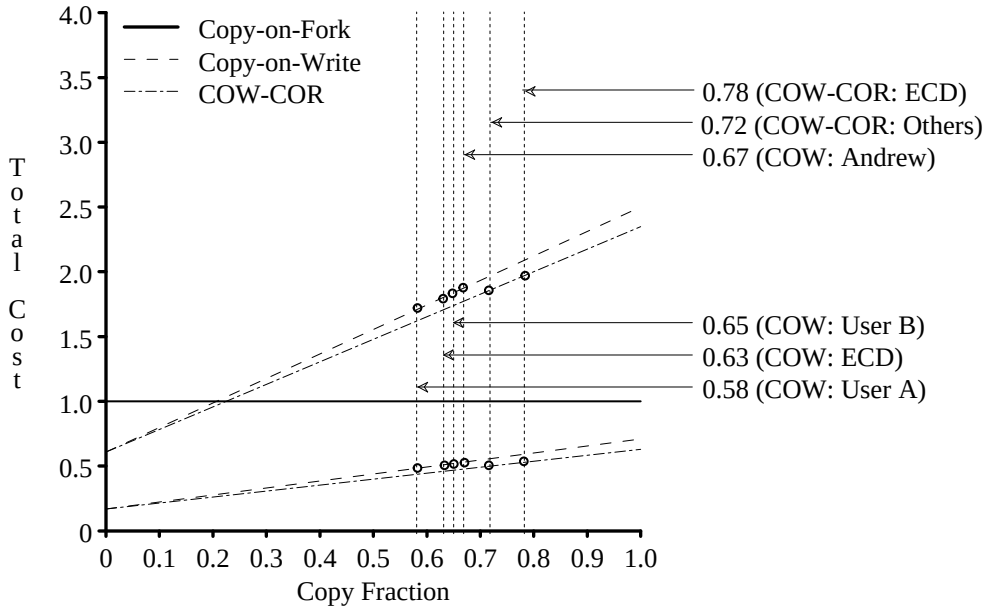

Figure 7-6. Total estimated cost of fork-related page copies on a Sun-4, as a function of the fraction of page copies and cache flushes because of copy-on-write or copy-on-reference faults. The attributes of the Sun-4 architecture that were used to compute the curves in the graph are given in Table 7-6. The lower lines of the graph are best-case scenarios for copy-on-write and COW-COR and the upper two lines worst-case scenarios. The best-case combines the lowest possible flush cost and copy cost for copy-on-write and COW-COR and the highest possible copy cost for copy-on-fork. The worst-case is when all of the data for each page is present in the cache and clean at fork time and when the highest possible flush and copy costs occur for copy-on-write and COW-COR at other times. A cost of 1.0 corresponds to the cost of copying every page at fork time. The 6 vertical lines are the copy fractions for the 4 benchmarks. The vertical line marked ‘‘Others’’ is the copy fraction for Andrew, User A and User B.

occur:

- The total number of pages that are marked copy-on-write and copy-on-reference would increase.
- The total number of copy-on-write and copy-on-reference faults would increase.
- The percentage of copy-on-write and copy-on-reference pages that are faulted on out of the total number of copy-on-write and copy-on-reference pages would

decrease.

- A pure copy-on-write scheme might improve relative to a COW-COR scheme because the percentage of pages that are copied on reference and then are later modified might decrease.

The only machine that I could perform the measurements on was a Sun-3 which has a fixed page size of 8 Kbytes. Although Sprite runs on Sun-2's, they do not have enough memory to allow me to measure either the ECD benchmark or normal use. However, since the trend in hardware is towards large page sizes, the results that I measured on the Sun-3 architecture should be applicable to most machines that will be built in the future.

7.6.6. Effect on System Performance

In addition to just affecting fork performance, copy-on-write mechanisms can also potentially affect overall system performance relative to copy-on-fork schemes. First of all, by making forks faster, copy-on-write will improve overall system response time. However, since fork time may only account for a small portion of the execution time of a process, the actual improvement in overall system performance may be very small. For example, the Andrew benchmark takes 280 seconds to complete. A pure copy-on-write scheme could eliminate 33% of the 2846 page copy operations (see Tables 7-3 and 7-4). If the fault cost were 0, then 2.5 milliseconds could be saved on each of the 939 page copy operations that would be eliminated, for a total savings of 2.3 out of the 280 seconds of execution time. This is less than a 1% improvement in the performance of the benchmark.

The other potential benefit from copy-on-write mechanisms is a reduction in memory use. By eliminating 30 to 40 percent of the page copy operations that would have been required under copy-on-fork, the amount of memory required to fork a process with copy-on-write will be 30 to 40 percent smaller. If a very large process forks, then this can potentially result in a substantial reduction in the demand placed on physical memory, which may result in an overall reduction in the number of page faults encountered by the system. However, for the programs that I measured, the amount of memory saved by copy-on-write in relation to the amount of physical memory available should be insignificant. This is especially true given the fact that most processes immediately exec after forking, so any extra memory will only be required for a brief instant. For example, in the Andrew benchmark, no process that forks has more than about 150 Kbytes of stack and heap, of which 100 Kbytes end up getting copied anyway. Since the machine that I ran the benchmark on has 16 Mbytes of memory, the 50 Kbytes that are unnecessarily copied occupies an insignificant amount space.

7.7. Conclusions

Copy-on-write has been gaining popularity in recent years as a mechanism to provide better fork performance. My measurements of copy-on-write indicate that it can indeed provide a tremendous performance improvement over copy-on-fork schemes if very few of the virtually-copied pages are modified. However, the measurements of normal use indicate that more than 58% of pages that are shared copy-on-write do get modified. As a result, less than 42% of the page copy operations that would have been required with a copy-on-fork scheme are eliminated. Thus, although copy-on-write has

tremendous potential, in practice it yields only a moderate performance gain over copy-on-fork.

There are two additional factors that may make it difficult to achieve even the 40% improvement suggested by the above measurements: page faults and cache flushing. An extra page fault will occur for each of the copy-on-write pages that ends up getting copied. Without a highly-tuned page fault handler, the additional page-fault overhead will more than compensate for the reduction in page copy operations. In addition, the architectural trend towards virtually-addressed caches has added the overhead of cache flushes to any copy-on-write implementation. The actual overhead will depend on the program behavior, the architecture, and the copy-on-write implementation, but my experience suggests that copy-on-write may actually result in worse performance than copy-on-fork for most applications on these machines. System designers need to pay very close attention to the fault cost and the cache flushing overhead if they wish to achieve the maximum benefit from copy-on-write.

The Sprite COW-COR scheme, which is a mixture of copy-on-write and copy-on-reference, provides a simple alternative implementation to pure copy-on-write schemes. It only requires 10 to 20 percent more page copy operations than a pure copy-on-write scheme, yet requires fewer cache flushes on machines with virtually-addressed caches. Estimates of the cache flushing overhead on a SPUR indicate that for the SPUR architecture COW-COR can actually provide slightly better fork performance than pure copy-on-write. However, on the Sun-4 architecture, which has a lower cache flushing cost than a SPUR, COW-COR can be slightly worse than pure copy-on-write. Thus, whether or not COW-COR is better than pure copy-on-write will depend on the

architecture.

CHAPTER 8

Conclusions

This dissertation has described the design, implementation and performance of several techniques for managing physical memory in a network operating system. In addition to measuring the mechanisms used daily in Sprite, I have also measured a variety of alternative mechanisms; these measurements provide the first quantitative comparisons between many of the popular memory-management techniques. My measurements have demonstrated that, by effectively utilizing physical memory, all workstations in a network, including those without disks, can attain high-performance data access while retaining the ease of sharing possible with timesharing systems. This high-performance can be attained while utilizing only a small portion of server CPU cycles and network bandwidth.

There are two keys to providing high performance in a network operating system. First, client caches must be allowed to become large without impacting virtual memory performance. The variable-size cache mechanism that I developed for Sprite lets the file cache vary in size while balancing the needs of the virtual memory system and the file system. The Sprite mechanism is better than any fixed-size cache mechanism and is a viable alternative to mapped files.

The Sprite mechanism has the advantage over mapped files that it allows the file system to be penalized so that it will be more difficult for the file system to take memory from the virtual memory system. The use of the penalty appears to be

effective in improving interactive response without degrading file system performance. The optimal value of the penalty is not yet clear; how much to penalize the file system will depend on the behavior of the users of the system.

The other key to attaining high performance is to use the correct writing policy on clients and servers. There is a tradeoff between reliability and performance; the most reliable policies give the worst performance, and the least reliable the best performance. As CPUs get faster and disks do not, the writing policy will become even more important; any policy that requires application programs to wait for the disk will cause serious performance degradation. The client and server writing policies that provides a good compromise between reliability and performance are to delay write-backs for 30 seconds. This gives performance comparable to policies with longer delays, while ensuring at most 60 seconds of data are lost in a system crash.

Attaining high performance need not require a relaxation on the consistency guarantees for files. Most distributed systems that cache data on client workstations do not provide the same level of consistency that was provided in timesharing systems. Thus, users cannot share data as easily as they once could. However, the cache consistency mechanism that I have developed for Sprite is simple yet lets file access have the same semantics as if all processes on all of the workstations were executing on a single timesharing system.

Another component to the performance of user programs in addition to file system performance is the speed of process creation. A very popular optimization on many systems is to eliminate page copy operations when a process is created through the use

of copy-on-write. Although copy-on-write can indeed provide a tremendous improvement in performance, my measurements of normal use indicate that copy on write will provide at best a 40% improvement in fork performance. Because of page fault overhead and the extra cache overhead on machines with virtually-addressed caches, even this 40% improvement will be very difficult to attain. The Sprite COW-COR scheme, which is a mixture of copy-on-write and copy-on-reference, provides a simple alternative implementation to pure copy-on-write schemes. It only requires 10 to 20 percent more page copies than a pure copy-on-write scheme, yet may require fewer cache flushes on machines with virtually-addressed caches.

The next several years should be very exciting because of tremendous increases in memory sizes, network speeds and CPU speeds. The work that I have presented in this dissertation will hopefully be useful for system designers who want to get the best performance out of the systems of the future. I believe that the key to attaining high performance in both present and future systems is to effectively utilize large physical memories. This will not only let users get their work done as efficiently as possible, but will also greatly improve system scalability.

CHAPTER 9

Bibliography

[Akh87] P. Akhtar, ‘‘A Replacement for Berkeley Memory Management’’, *Proceedings of the USENIX 1987 Summer Conference*, JUNE 1987, 69-79.

[BLM87] M. J. Bach, M. W. Luppi, A. S. Melamed and K. Yueh, ‘‘A Remote-File Cache for RFS’’, *Proceedings of the USENIX Summer 1987 Conference*, June 1987, 275-280.

[BCD72] A. Bensoussan, C. T. Clingen and R. C. Daley, ‘‘The MULTICS Virtual Memory: Concepts and Design’’, *Comm. of the ACM 15*, 5 (May 1972).

[BiN84] A. D. Birrell and B. J. Nelson, ‘‘Implementing Remote Procedure Calls’’, *ACM Transactions on Computer Systems 2*, 1 (Feb. 1984), 39-59.

[BBM72] D. G. Bobrow, J. D. Burchfiel, D. L. Murphy and R. S. Tomlinson, ‘‘TENEX, a Paged Time Sharing System for the PDP-10’’, *Comm. of the ACM 15*, 3 (Mar. 1972), 1135-143.

[BKT85] M. R. Brown, K. N. Kolling and E. A. Taft, ‘‘The Alpine File System’’, *Trans. Computer Systems 3*, 4 (Nov. 1985), 261-293.

[CaW86] L. F. Cabrera and J. Wylie, ‘‘QuickSilver Distributed File Services: An Architecture for Horizontal Growth’’, Research Report RJ 5578 (56697), San Jose, California, June 1986.

[ChR85] D. R. Cheriton and P. J. Roy, ‘‘Performance of the V Storage Server: A Preliminary Report’’, *Proc. of the 1985 ACM Computer Science Conference*, Mar. 1985, 302-308.

[DaD68] R. C. Daley and J. B. Dennis, ‘‘Virtual Memory, Processes and Sharing in MULTICS’’, *Comm. of the ACM 11*, 5 (May 1968), 306-312.

[Flo86] R. Floyd, ‘‘Short-Term File Reference Patterns in a UNIX Environment’’, Technical Report Tech. Rep. 177, The University of Rochester, Mar. 1986.

[GMS87] R. A. Gingell, J. P. Moran and W. A. Shannon, ‘‘Virtual Memory Architecture in SunOS’’, *Proceedings of the USENIX 1987 Summer Conference*, JUNE 1987, 81-94.

[Gus87] R. Gusella, ‘‘The Analysis of Diskless Workstation Traffic on the Ethernet’’, Technical Report UCB/Computer Science Dpt. 87/379, University of California, Berkeley, Dec. 1987.

[Hil86] M. D. Hill, et al., ‘‘SPUR: A VLSI Multiprocessor Workstation’’, *IEEE Computer 19*, 11 (Nov. 1986), 8-22.

[How88] J. Howard, et al., ‘‘Scale and Performance in a Distributed File System’’, *Trans. Computer Systems 6*, 1 (Feb. 1988), 51-81.

[Kel86] E. Kelly, *Sun-4 Architecture Manual*, Sun Microsystems Inc., Nov. 1986.

[Kel88] E. Kelly, Personal Communication, Oct. 1988.

[Ken86] C. A. Kent, *Cache Coherence in Distributed Systems*, Phd Thesis, Purdue University, 1986.

[LZC86] E. Lazowska, J. Zahorjan, D. Cheriton and W. Zwaenepoel, ‘‘File Access Performance of Diskless Workstations’’, *Trans. Computer Systems*, Aug. 1986.

[LLH85] P. Leach, P. Levine, J. Hamilton and B. Stumpf, ‘‘The File System of an Integrated Local Network’’, *Proc. of the 1985 ACM Computer Science Conference*, Mar. 1985, 309-324.

[Lea83] P. J. Leach, et al., ‘‘The Architecture of an Integrated Local Network’’, *IEEE Journal on Selected Areas in Communications SAC-1*, 5 (Nov. 1983), 842-857.

[Li86] K. Li, *Shared Virtual Memory on a Loosely Coupled Multiprocessor*, PhD Thesis, Yale University, 1986.

[MJL84] M. K. McKusick, W. N. Joy, S. J. Leffler and R. S. Fabry, ‘‘A Fast File System for UNIX’’, *Trans. Computer Systems 2*, 3 (Aug. 1984), 181-197..

[Mor86] J. H. Morris, et al., ‘‘Andrew: A Distributed Personal Computing Environment’’, *Comm. of the ACM 29*, 3 (Mar. 1986), 184-201.

[Mur72] D. L. Murphy, ‘‘Storage organization and management in TENEX’’, *Proceedings AFIPS Fall Joint Computer Conference 15*, 3 (1972), 23-32.

[Nel86] M. N. Nelson, ‘‘The Sprite Virtual Memory System’’, Technical Report UCB/Computer Science Dpt. 86/301, University of California, Berkeley, June 1986.

[OCD88] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson and B. B. Welch, ‘‘The Sprite Network Operating System’’, *IEEE Computer 21*, 2

(Feb. 1988), 23-36.

[Ous85] J. K. Ousterhout, et al., ‘‘A Trace-Driven Analysis of the 4.2 BSD UNIX File System’’, *Proceedings of the 10th Symp. on Operating System Prin.*, Dec. 1985, 15-24.

[PoW85] G. Popek and B. Walker, editors, *The LOCUS Distributed System Architecture*, MIT Press, 1985.

[RaR81] R. F. Rashid and G. G. Robertson, ‘‘Accent: A communication oriented network operating system kernel’’, *Proceedings of the 8th Symposium on Operating Systems Principles*, 1981, 164-175.

[RaF86] R. F. Rashid and R. Fitzgerald, ‘‘The Integration of Virtual Memory Management and Interprocess Communication in Accent’’, *Trans. Computer Systems 4*, 2 (May 1986), 147-177.

[Ras88] R. Rashid, Personal Communication, Mar. 1988.

[Ras87] R. Rashid, et al., ‘‘Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures’’, *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, Oct. 1987, 31-39.

[Red80] D. D. Redell, et al., ‘‘Pilot: An Operating System for a Personal Computer’’, *Communications of the ACM 23*, 2 (Feb. 1980), 81-92.

[Rif86] A. P. Rifkin, et al., ‘‘RFS Architectural Overview’’, *USENIX Association 1986 Summer Conference Proceedings*, 1986.

[RiT74] D. M. Ritchie and K. Thompson, ‘‘The UNIX Time-Sharing System’’, *Comm. of the ACM 17*, 7 (July 1974), 365-375..

[San85] R. Sandberg, et al., ‘‘Design and Implementation of the Sun Network Filesystem’’, *Proceedings of the USENIX 1985 Summer Conference*, JUNE 1985, 119-130.

[Sat85] M. Satyanarayanan, et al., ‘‘The ITC Distributed File System: Principles and Design’’, *Proceedings of the 10th Symp. on Operating System Prin.*, 1985, 35-50.

[SGN85] M. Schroeder, D. Gifford and R. Needham, ‘‘A Caching File System for a Programmer’s Workstation’’, *Proceedings of the 10th Symp. on Operating System Prin.*, Dec. 1985, 25-34.

[Smi85] A. J. Smith, ‘‘Disk Cache - Miss Ratio Analysis and Design Considerations’’, *Trans. Computer Systems 3*, 3 (Aug. 1985), 161-203.

[SSS85] *Sun-3 Architecture Manual*, Sun Microsystems Inc., July 1985.

[SCC86] E. W. Sznyter, P. Clancy and J. Crossland, ‘‘A New Virtual-Memory Implementation for Unix’’, *Proceedings of the USENIX 1986 Summer Conference*, JUNE 1986, 81-88.

[Tho87] J. G. Thompson, *Efficient Analysis of Caching Systems*, Phd Thesis, University of California at Berkeley, 1987.

[Wal83] B. Walker, et al., ‘‘The LOCUS Distributed Operating System’’, *Proceedings of the 9th Symp. on Operating System Prin. 17*, 5 (Nov. 1983), 49-70.

[Wel86] B. B. Welch, ‘‘The Sprite Remote Procedure Call System’’, Technical Report UCB/Computer Science Dpt. 86/302, University of California, Berkeley, June 1986.

[WeO86] B. B. Welch and J. K. Ousterhout, ‘‘Prefix Tables: A Simple Mechanism for Locating Files in a Distributed Filesystem’’, *Proc. of the 6th Int’l Conf. on Distributed Computing Systems*, May 1986, 184-189.

[Woo88] D. Wood, Personal Communication, Apr. 1988.

APPENDIX A

Detailed Results from Chapter 4

This appendix contains 5 tables of detailed results that were not given in Chapter 4. The five tables A-1 through A-5 contain the results of running the five benchmarks Andrew, Vm-make, Sort, Ditroff and Diff on Sun-3 workstations. The top row of each line contains the results and the bottom row contains the standard deviations from the 3 runs. The first two lines of each table are the results when the benchmark was run locally on the file server. The next two lines are when the benchmark was run on a client with no cache. The rest of the lines are for various client cache sizes. The first column of each table is the elapsed time in seconds, the second column the number of network bytes transferred during the benchmark, the third column the server utilization, the fourth column the number of disk reads and disk writes and the last column the disk utilization.

Results for the Andrew Benchmark					
Cache Size	Elapsed Time	Network Kbytes	Server Util	Disk I/O's	Disk Util
4 Mbytes Local, cold	265 2.53	-- --	-- --	1553 106.53	14% 0.58
4 Mbytes Local, warm	255 2.53	-- --	-- --	924 40.11	10% 0.00
No cache, cold	321 1.53	24361 75.63	17.66% 0.09	1387 6.35	9% 0.00
No cache, warm	307 6.08	24320 2.31	18.00% 0.11	863 359.80	6% 1.73
128 Kbytes	301 3.06	14173 184.64	14.76% 0.09	650 9.54	5% 0.00
256 Kbytes	290 1.00	11118 108.15	14.00% 0.03	657 10.58	5% 0.00
512 Kbytes	286 1.15	9617 305.21	13.58% 0.06	644 7.77	5% 0.58
1024 Kbytes	283 1.00	8380 455.17	13.26% 0.11	652 7.51	6% 0.00
2048 Kbytes	278 0.58	6303 81.95	12.66% 0.01	652 1.73	5% 0.58
3072 Kbytes	277 0.00	5496 37.29	12.43% 0.02	653 5.69	6% 0.00
4096 Kbytes, warm	275 0.00	4378 13.00	12.13% 0.01	647 1.73	5% 0.00
4096 Kbytes, cold	288 2.31	6425 381.73	12.49% 0.07	1208 85.81	8% 0.58

Table A-1. Results from the Andrew Benchmark.

Results for the VM Benchmark					
Cache Size	Elapsed Time	Network Kbytes	Server Util	Disk I/O's	Disk Util
4 Mbytes Local, cold	284.39 1.03	-- --	-- --	948 23.16	7% 0.58
4 Mbytes Local, warm	277.10 1.03	-- --	-- --	586 4.58	5% 0.00
No cache, cold	336.93 0.76	17849 9.54	13.88% 0.04	900 9.71	6% 0.00
No cache, warm	330.10 0.74	17717 0.58	14.03% 0.02	568 13.87	4% 0.00
128 Kbytes	318.68 0.25	13847 194.07	12.02% 0.06	456 1.73	3% 0.00
256 Kbytes	313.33 1.01	11474 157.61	11.31% 0.11	473 28.58	3% 0.58
512 Kbytes	303.94 0.42	6870 135.74	9.91% 0.03	442 6.24	3% 0.00
1024 Kbytes	298.09 0.82	4403 150.41	9.17% 0.04	437 2.65	3% 0.00
2048 Kbytes	296.16 1.49	3099 48.68	8.89% 0.06	430 0.58	3% 0.00
3072 Kbytes	295.49 0.81	2884 3.06	8.86% 0.02	430 5.03	3% 0.58
4096 Kbytes, warm	295.77 0.21	2885 2.08	8.87% 0.01	432 1.00	3% 0.58
4096 Kbytes, cold	304.75 0.18	3956 0.58	8.97% 0.01	764 2.65	5% 0.00

Table A-2. Results from the Vm-make Benchmark.

Results for the Sort Benchmark					
Cache Size	Elapsed Time	Network Kbytes	Server Util	Disk I/O's	Disk Util
4 Mbytes Local, cold	63.56 0.20	-- --	-- --	716 16.09	20% 0.58
4 Mbytes Local, warm	59.70 0.20	-- --	-- --	470 3.06	15% 0.00
No cache, cold	74.84 0.40	6154 9.81	15.03% 0.05	855 11.14	21% 0.00
No cache, warm	70.79 0.44	6152 2.31	15.16% 0.08	564 9.54	16% 1.53
128 Kbytes	68.82 0.42	6152 3.61	15.25% 0.07	527 16.46	16% 1.15
256 Kbytes	71.39 2.89	6170 35.16	14.83% 0.35	550 71.60	15% 1.15
512 Kbytes	71.92 0.87	6154 4.36	14.34% 0.08	462 13.58	12% 0.00
1024 Kbytes	70.84 0.09	6105 2.00	13.99% 0.07	360 3.46	8% 0.58
2048 Kbytes	63.59 0.10	3392 39.89	9.39% 0.18	307 3.06	7% 0.58
3072 Kbytes	59.37 0.44	2058 256.77	7.37% 0.40	305 3.21	8% 0.00
4096 Kbytes, warm	58.75 0.04	1730 29.37	7.04% 0.02	306 2.00	8% 0.00
4096 Kbytes, cold	64.74 0.16	3078 46.03	9.49% 0.27	581 0.58	13% 0.00

Table A-3. Results from the Sort Benchmark.

Results for the Ditroff Benchmark					
Cache Size	Elapsed Time	Network Kbytes	Server Util	Disk I/O's	Disk Util
4 Mbytes Local, cold	127.83 0.38	-- --	-- --	289 15.01	5% 0.58
4 Mbytes Local, warm	125.03 0.38	-- --	-- --	133 0.58	2% 0.00
No cache, cold	133.17 0.39	2050 10.39	4.93% 0.04	267 2.00	5% 0.00
No cache, warm	132.43 1.94	2362 269.05	5.12% 0.30	206 129.06	3% 2.31
128 Kbytes	126.70 0.50	836 266.77	1.81% 0.20	116 1.00	2% 0.00
256 Kbytes	126.41 0.05	682 2.08	1.68% 0.02	117 2.65	2% 0.00
512 Kbytes	126.29 0.02	578 4.73	1.61% 0.01	118 2.89	2% 0.00
1024 Kbytes	125.85 0.01	325 1.15	1.41% 0.02	119 2.65	2% 0.00
2048 Kbytes	125.87 0.02	325 1.15	1.42% 0.02	118 3.06	2% 0.00
3072 Kbytes	125.87 0.02	325 1.15	1.42% 0.02	118 3.06	2% 0.00
4096 Kbytes, cold	128.01 0.92	839 270.49	2.05% 0.28	282 46.77	5% 0.00

Table A-4. Results from the Ditroff Benchmark.

Results for the Diff Benchmark					
Cache Size	Elapsed Time	Network Kbytes	Server Util	Disk I/O's	Disk Util
4 Mbytes Local, cold	21.09 0.12	-- --	-- --	553 0.58	76% 0.58
4 Mbytes Local, warm	4.60 0.12	-- --	-- --	32 1.73	11% 0.58
No cache, cold	25.15 0.13	2289 5.20	14.33% 0.08	548 3.61	63% 0.58
No cache, warm	8.47 0.10	2286 0.00	27.18% 0.10	24 1.15	4% 0.58
128 Kbytes	8.66 0.05	2284 4.04	26.72% 0.36	24 1.15	4% 0.58
256 Kbytes	8.73 0.07	2285 1.15	26.56% 0.51	24 1.15	4% 0.58
512 Kbytes	8.69 0.09	2285 1.15	26.68% 0.16	24 1.15	4% 0.58
1024 Kbytes	8.84 0.16	2308 41.00	26.49% 0.12	25 1.73	4% 0.58
2048 Kbytes	8.72 0.08	2285 1.15	26.56% 0.22	24 1.53	5% 0.58
2128 Kbytes	4.54 0.01	6 0.58	2.18% 0.74	24 1.15	8% 0.58
3072 Kbytes	4.53 0.01	5 1.15	2.18% 0.75	25 1.15	9% 0.58
4096 Kbytes, warm	4.53 0.00	5 1.15	1.53% 0.13	26 2.08	10% 1.15
4096 Kbytes, cold	25.05 0.13	2284 0.00	14.21% 0.10	548 0.58	64% 0.00

Table A-5. Results from the Diff Benchmark.

APPENDIX B

Standard Deviations from Chapter 5

This appendix contains tables of the standard deviations for the results given in the tables in Chapter 5. Each data point in the tables in Chapter 5 was computed by taking the average of three runs of the given benchmark.

Network Kbytes vs. Client Policy Standard Deviations									
Client Policy	Andrew			Vm			Sort		
	Read	Write	Total	Read	Write	Total	Read	Write	Total
WT	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.6
WBOC	0.6	0.0	0.0	0.0	0.0	0.0	0.6	0.0	0.0
ASAP	0.0	0.0	0.6	0.0	0.6	0.6	0.6	0.0	0.0
WBOC-ASAP	0.6	0.0	0.6	0.0	0.0	0.6	0.0	0.0	0.6
delay-30	0.0	2.5	2.0	0.0	2.6	2.1	1.2	61.4	62.6
full-delay	0.6	1.2	1.2	0.0	2.6	2.3	0.0	0.6	0.0
WT-TMP	0.6	1.7	1.7	0.0	1.2	1.7	0.0	1.2	1.2
WBOC-TMP	0.6	0.6	0.6	0.6	0.0	0.0	0.0	0.6	0.0
ASAP-TMP	0.0	0.0	0.0	0.6	0.0	0.6	0.0	0.0	0.0

Table B-1. Network Kbytes vs. Client Policy. This table shows the standard deviations for the results in Table 5-4.

Disk Traffic Standard Deviations: 30-Second Delay on Server												
Client Policy	Andrew				Vm-make				Sort			
	Disk Util	Disk Writes			Disk Util	Disk Writes			Disk Util	Disk Writes		
		Data	Ind/ Desc	Total		Data	Ind/ Desc	Total		Data	Ind/ Desc	Total
WT	0.0	1.0	2.9	2.1	0.0	2.1	5.1	3.5	1.2	19.9	4.0	23.8
WBOC	0.0	0.0	2.6	2.6	0.0	0.6	4.2	4.7	1.0	13.3	1.0	12.4
ASAP	0.0	2.6	5.6	6.1	0.0	0.0	2.5	2.5	0.6	13.9	1.2	14.5
WBOC-ASAP	0.0	0.0	6.6	6.6	0.0	0.6	3.6	3.2	0.6	14.4	1.2	15.6
delay-30	0.0	0.6	9.3	9.5	0.0	0.6	3.6	4.0	0.0	0.0	0.6	0.6
full-delay	0.0	0.0	23.3	23.3	0.0	0.0	1.7	1.7	0.0	0.0	0.6	0.6
WT-TMP	0.0	1.0	7.0	7.8	0.0	0.6	6.0	5.5	0.6	0.0	0.6	0.6
WBOC-TMP	0.6	0.6	2.6	2.9	0.0	0.0	7.4	7.4	0.0	0.0	0.6	0.6
ASAP-TMP	0.6	0.0	5.7	5.7	0.0	0.6	2.5	3.1	0.0	0.0	0.6	0.6

Table B-2. The table shows the standard deviations for the results in Table 5-5.

Disk Traffic Standard Deviations: Write-Through on Server												
Client Policy	Andrew				Vm-make				Sort			
	Disk Util	Disk Writes			Disk Util	Disk Writes			Disk Util	Disk Writes		
		Data	Ind/ Desc	Total		Data	Ind/ Desc	Total		Data	Ind/ Desc	Total
WT	1.0	1.2	8.5	7.4	0.0	2.3	13.7	15.9	0.6	1.2	2.6	3.8
WBOC	0.0	0.0	3.8	3.8	0.0	0.0	4.4	4.4	0.6	0.0	2.0	2.0
ASAP	0.6	2.6	4.0	6.4	0.0	0.6	3.1	3.6	0.6	0.0	0.6	0.6
WBOC-ASAP	1.0	1.0	3.2	2.3	0.0	0.6	5.0	5.5	0.6	0.0	0.0	0.0
delay-30	1.0	3.5	8.9	12.3	0.0	1.5	2.5	2.3	0.6	8.2	32.9	41.0
full-delay	0.6	0.0	14.8	14.8	0.0	0.0	7.8	7.8	0.0	0.0	1.5	1.5
WT-TMP	0.0	1.2	1.7	2.5	0.0	1.2	7.5	6.6	0.0	0.6	2.5	3.1
WBOC-TMP	0.6	0.0	1.2	1.2	0.0	0.0	4.0	4.0	0.0	0.0	2.6	2.6
ASAP-TMP	0.6	3.1	7.2	10.3	0.0	0.0	1.7	1.7	0.0	0.0	0.0	0.0

Table B-3. The table shows the standard deviations for the results in Table 5-6.

Disk Traffic Standard Deviations: ASAP on Server												
Client Policy	Andrew				Vm-make				Sort			
	Disk Util	Disk Writes			Disk Util	Disk Writes			Disk Util	Disk Writes		
		Data	Ind/ Desc	Total		Data	Ind/ Desc	Total		Data	Ind/ Desc	Total
WT	0.6	7.0	8.9	15.6	0.0	0.6	2.9	3.2	0.0	1.0	2.1	2.5
WBOC	0.0	1.0	17.6	17.4	0.0	2.5	6.5	8.2	0.6	0.6	6.0	5.6
ASAP	0.0	0.6	10.1	10.4	0.0	1.2	7.4	8.2	0.6	0.6	10.4	9.8
WBOC-ASAP	0.6	0.6	5.0	5.5	0.0	1.5	7.9	9.5	0.6	0.6	7.0	7.5
delay-30	0.0	5.5	13.1	18.2	0.0	1.0	6.8	7.1	0.6	21.7	11.4	32.7
full-delay	0.6	0.0	11.1	11.1	0.0	2.5	0.6	2.0	0.6	0.6	2.5	2.0
WT-TMP	0.6	1.5	12.5	13.5	0.0	2.3	9.1	7.4	0.0	1.5	1.5	3.0
WBOC-TMP	0.0	0.6	1.2	1.0	0.0	1.0	3.1	4.0	0.6	0.6	9.0	8.5
ASAP-TMP	0.6	3.1	20.1	18.6	0.6	0.6	7.6	7.9	0.0	0.6	1.7	2.1

Table B-4. The table shows the standard deviations for the results in Table 5-7.

Disk Traffic Standard Deviations: Last Dirty Block												
Client Policy	Andrew				Vm-make				Sort			
	Disk Util	Disk Writes			Disk Util	Disk Writes			Disk Util	Disk Writes		
		Data	Ind/ Desc	Total		Data	Ind/ Desc	Total		Data	Ind/ Desc	Total
WT	0.0	1.0	6.0	6.7	0.0	0.6	5.5	5.6	0.6	0.6	2.6	2.1
WBOC	0.0	0.0	3.8	3.8	0.0	0.0	3.5	3.5	0.6	0.6	2.3	2.9
ASAP	0.0	13.0	12.3	23.8	0.0	0.0	4.5	4.5	1.2	1.2	4.4	4.9
WBOC-ASAP	0.0	0.6	7.2	7.0	0.0	0.0	2.6	2.6	0.6	0.0	6.7	6.7
delay-30	0.0	0.6	2.3	2.9	0.0	2.1	0.6	1.5	2.1	43.1	4.0	46.9
full-delay	0.0	0.6	2.9	3.2	0.0	0.0	2.5	2.5	0.6	0.0	1.7	1.7
WT-TMP	0.0	1.2	4.5	3.6	0.0	1.2	6.4	7.0	0.0	0.0	0.6	0.6
WBOC-TMP	0.0	0.0	6.2	6.2	0.0	0.0	4.0	4.0	0.6	0.6	1.0	0.6
ASAP-TMP	0.0	0.0	7.2	7.2	0.0	0.0	8.0	8.0	0.0	0.0	4.4	4.4

Table B-5. The table shows the standard deviations for the results in Table 5-8.

Client Elapsed Time and Server Utilization: 30-Second Delay on Server						
Client Policy	Andrew		Vm-make		Sort	
	Elapsed Time	Server Util	Elapsed Time	Server Util	Elapsed Time	Server Util
WT	0.0	0.1	0.6	0.1	0.6	0.1
WBOC	0.6	0.0	0.4	0.0	0.1	0.1
ASAP	1.2	0.0	0.9	0.0	1.1	0.1
WBOC-ASAP	0.6	0.0	0.9	0.0	0.9	0.0
delay-30	0.6	0.0	0.2	0.0	0.1	0.3
full-delay	1.0	0.0	0.1	0.0	0.0	0.1
WT-TMP	0.6	0.0	0.5	0.0	0.0	0.1
WBOC-TMP	0.0	0.0	0.4	0.0	0.0	0.1
ASAP-TMP	2.9	0.1	0.3	0.0	0.0	0.1

Table B-6. This table contains the standard deviations for the results given in Table 5-9.

Client Elapsed Time and Server Utilization: Write-Through on Server						
Client Policy	Andrew		Vm-make		Sort	
	Elapsed Time	Server Util	Elapsed Time	Server Util	Elapsed Time	Server Util
WT	1.5	0.1	2.7	0.1	0.0	0.0
WBOC	2.5	0.1	1.7	0.0	0.3	0.0
ASAP	1.7	0.1	0.5	0.0	0.1	0.1
WBOC-ASAP	4.6	0.2	0.8	0.0	0.2	0.0
delay-30	0.6	0.1	0.5	0.0	2.8	0.1
full-delay	0.6	0.0	1.0	0.0	0.1	0.0
WT-TMP	2.1	0.1	1.6	0.0	0.3	0.1
WBOC-TMP	2.9	0.1	0.7	0.0	0.1	0.0
ASAP-TMP	0.6	0.0	0.8	0.0	0.1	0.1

Table B-7. This table contains the standard deviations for the results given in Table 5-10.

Client Elapsed Time and Server Utilization: ASAP on Server						
Client Policy	Andrew		Vm-make		Sort	
	Elapsed Time	Server Util	Elapsed Time	Server Util	Elapsed Time	Server Util
WT	0.0	0.0	0.4	0.0	0.1	0.1
WBOC	0.6	0.0	0.9	0.0	0.2	0.2
ASAP	0.0	0.0	0.5	0.0	0.0	0.0
WBOC-ASAP	0.6	0.0	1.1	0.1	0.0	0.1
delay-30	0.0	0.0	1.9	0.1	0.2	0.5
full-delay	0.0	0.0	1.0	0.0	0.1	0.1
WT-TMP	0.6	0.0	1.8	0.0	0.1	0.1
WBOC-TMP	0.6	0.0	0.7	0.0	0.0	0.2
ASAP-TMP	1.0	0.0	0.5	0.0	0.0	0.1

Table B-8. This table contains the standard deviations for the results given in Table 5-11.

Client Elapsed Time and Server Utilization: LDB Policy						
Client Policy	Andrew		Vm-make		Sort	
	Elapsed Time	Server Util	Elapsed Time	Server Util	Elapsed Time	Server Util
WT	1.2	0.0	1.9	0.1	0.7	0.1
WBOC	0.6	0.0	1.1	0.0	0.4	0.1
ASAP	0.0	0.0	1.4	0.0	0.2	0.2
WBOC-ASAP	1.0	0.0	1.4	0.1	0.3	0.1
delay-30	0.6	0.0	0.5	0.0	0.8	0.6
full-delay	0.6	0.0	1.1	0.0	0.0	0.1
WT-TMP	0.6	0.0	0.4	0.0	0.2	0.0
WBOC-TMP	0.6	0.0	0.6	0.0	1.2	0.2
ASAP-TMP	1.0	0.0	1.1	0.0	0.0	0.1

Table B-9. This table contains the standard deviations for the results given in Table 5-12.

APPENDIX C

Detailed Results from Chapter 6

This appendix contains tables of results from Chapter 6. The tables were not given in Chapter 6 because they contained unnecessary details. The tables C.1 through C.10 give the performance of the variable-sized cache benchmark for the 5 physical memory sizes. The top row in each table is the result when the client did not use any cache. The next few rows before the row labeled *norm* are when the file system cache was fixed at various sizes. The row labeled *norm* was with a variable-size cache with no file system penalty. The remaining rows are when the file system was penalized from 60 to 960 seconds. Tables C.1, C.3, C.5, C.7, and C.9 contain the results from the benchmarks; each row was generated by taking the average of the results from three runs of the benchmark in the given configuration. Tables C.2, C.4, C.6, C.8, and C.10 contain the standard deviations of the three runs of the benchmark.

In tables C.1, C.3, C.5, C.7, and C.9 each row is subdivided into two rows. The upper row is the result of the benchmark and the lower row is the result relative to the row labeled *norm*.

The columns of each table are as follows. The first column is the amount of fixed cache or the amount of penalty. The second column is the percent of the server's CPU that was utilized while running the benchmark. The third column is the number of seconds that the benchmark took to execute. Column 4 is the total number of page faults and Column 5 is the number of faults that came from swap space. The sixth

column is the number of pages that were written out to swap space. Columns 7 through 9 are the number of file system Kbytes that were transferred by the network. Finally, Columns 10 through 12 are the total number of Kbytes, including both VM and FS bytes, that were transferred by the network.

10 MegaBytes of Memory on the Client											
	Server	Elap	Faults		Page	FS Net I/O			Total Net I/O		
	Util	Time	Total	Swap	Outs	Read	Write	Total	Read	Write	Total
nocc	24.91	982	4569	2830	3098	33948	9810	43759	74647	39506	114154
	1.07	1.04	1.38	1.32	0.83	1.49	1.03	1.36	1.42	0.90	1.18
0.5M	23.87	951	4701	3332	3192	30185	9506	39692	71776	39804	111580
	1.02	1.01	1.42	1.56	0.85	1.33	1.00	1.23	1.37	0.90	1.16
1M	36.73	1899	48869	44919	6272	28787	9485	38273	438587	76257	514845
	1.57	2.02	14.77	20.99	1.67	1.26	1.00	1.19	8.35	1.73	5.33
norm	23.37	942	3309	2140	3748	22761	9485	32247	52533	43987	96521
	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
60	23.19	928	4009	2703	3183	24828	9485	34314	60434	39398	99832
	0.99	0.99	1.21	1.26	0.85	1.09	1.00	1.06	1.15	0.90	1.03
120	24.04	923	4424	3064	3234	27075	9485	36561	66234	39990	106225
	1.03	0.98	1.34	1.43	0.86	1.19	1.00	1.13	1.26	0.91	1.10
240	23.34	923	3806	2628	3042	29600	9485	39086	63686	38260	101946
	1.00	0.98	1.15	1.23	0.81	1.30	1.00	1.21	1.21	0.87	1.06
480	23.59	909	3556	2399	3090	29666	9486	39152	61681	38629	100311
	1.01	0.97	1.07	1.12	0.82	1.30	1.00	1.21	1.17	0.88	1.04
960	23.87	919	3837	2654	3106	29727	9485	39213	64080	38829	102910
	1.02	0.98	1.16	1.24	0.83	1.31	1.00	1.22	1.22	0.88	1.07

Table C-1. 10 Mbytes of memory on Client.

Standard Deviations: 10 MegaBytes of Memory on the Client											
	Server	Elap	Faults		Page	FS Net I/O			Total Net I/O		
	Util	Time	Total	Swap	Outs	Read	Write	Total	Read	Write	Total
nocc	0.4	34	744	591	247	25	0.0	25	6182	2258	8425
0.5M	0.4	8.7	439	363	79	16	2.3	15	3674	763	4378
1M	0.3	66	2731	2811	27	215	0.6	215	22609	948	23556
norm	0.4	20	212	213	149	721	0.6	721	2166	1288	3247
60	0.7	34	958	769	202	172	0.6	172	7821	1900	9431
120	0.8	39	1026	863	304	929	0.0	929	7982	2800	10536
240	0.2	43	727	628	194	256	0.0	256	5836	1797	7614
480	0.2	18	332	287	107	310	0.6	311	3033	986	4015
960	1.0	6.4	934	778	164	205	0.0	205	7637	1599	9179

Table C-2. Standard deviations 10 Mbytes of memory on Client.

11 MegaBytes of Memory on the Client											
	Server	Elap	Faults		Page	FS Net I/O			Total Net I/O		
	Util	Time	Total	Swap	Outs	Read	Write	Total	Read	Write	Total
nocc	23.57	846	2105	836	2481	33941	9810	43751	54005	33636	87642
	1.07	1.01	0.86	0.57	0.85	1.73	1.03	1.50	1.28	0.91	1.11
0.5M	21.82	793	1739	866	2230	30242	9508	39750	47041	30935	77976
	0.99	0.94	0.71	0.59	0.76	1.54	1.00	1.37	1.12	0.84	0.99
1M	21.57	850	2304	1323	2431	29380	9485	38866	50857	32720	83578
	0.98	1.01	0.94	0.90	0.83	1.50	1.00	1.34	1.21	0.89	1.06
2M	35.71	1896	47323	43525	6051	28419	9485	37905	425297	73998	499296
	1.63	2.26	19.22	29.55	2.06	1.45	1.00	1.30	10.11	2.01	6.33
norm	21.96	840	2462	1473	2934	19609	9485	29094	42080	36781	78862
	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
60	21.13	821	2027	1097	2355	23287	9482	32769	42206	31828	74034
	0.96	0.98	0.82	0.74	0.80	1.19	1.00	1.13	1.00	0.87	0.94
120	20.85	798	1708	874	2065	24476	9482	33958	40754	29356	70110
	0.95	0.95	0.69	0.59	0.70	1.25	1.00	1.17	0.97	0.80	0.89
240	21.54	797	1766	915	2265	28855	9485	38341	45819	31249	77068
	0.98	0.95	0.72	0.62	0.77	1.47	1.00	1.32	1.09	0.85	0.98
480	22.11	777	1714	867	2303	29195	9485	38681	45729	31476	77205
	1.01	0.93	0.70	0.59	0.79	1.49	1.00	1.33	1.09	0.86	0.98
960	21.38	780	1760	910	2053	29500	9484	38984	46402	29406	75808
	0.97	0.93	0.72	0.62	0.70	1.50	1.00	1.34	1.10	0.80	0.96

Table C-3. 11 Mbytes of memory on Client.

Standard Deviations: 11 MegaBytes of Memory on the Client											
	Server	Elap	Faults		Page	FS Net I/O			Total Net I/O		
	Util	Time	Total	Swap	Outs	Read	Write	Total	Read	Write	Total
nocc	0.5	23	34	20	166	27	0.0	27	337	1468	1537
0.5M	0.1	14	58	39	25	26	2.3	27	530	232	751
1M	0.4	8.5	105	91	65	8.0	0.0	8.0	894	574	1458
2M	2.2	285	14042	12851	964	172	0.0	172	116909	11367	128158
norm	0.1	14	35	30	129	325	0.0	324	223	1093	1268
60	0.6	27	159	151	168	756	2.3	757	2097	1416	2495
120	0.2	5.1	73	56	130	206	2.3	208	807	1085	1510
240	0.8	8.9	84	73	290	1009	0.0	1009	1435	2515	3950
480	0.5	16	46	46	187	228	0.0	228	635	1610	2206
960	0.2	9.6	35	30	93	147	2.3	150	364	771	534

Table C-4. Standard deviations with 11 Mbytes of memory on Client.

12 MegaBytes of Memory on the Client											
	Server	Elap	Faults		Page	FS Net I/O			Total Net I/O		
	Util	Time	Total	Swap	Outs	Read	Write	Total	Read	Write	Total
nocc	23.02	800	1931	670	1766	33936	9810	43747	52424	27498	79922
	1.10	1.02	0.90	0.55	0.75	2.22	1.03	1.76	1.50	0.87	1.20
0.5M	21.18	768	1485	623	1620	30241	9508	39750	44819	25692	70512
	1.02	0.98	0.69	0.51	0.69	1.98	1.00	1.60	1.29	0.81	1.06
1M	21.47	783	1592	767	1872	29336	9485	38822	44810	27819	72630
	1.03	1.00	0.74	0.63	0.80	1.92	1.00	1.57	1.29	0.88	1.09
2M	22.94	878	2975	1844	2640	28840	9485	38326	55914	34651	90566
	1.10	1.12	1.39	1.52	1.12	1.88	1.00	1.55	1.60	1.10	1.36
3M	38.41	2217	63571	59273	6674	22599	9486	32085	554644	82920	637564
	1.84	2.82	29.68	48.82	2.84	1.48	1.00	1.29	15.91	2.62	9.59
norm	20.86	787	2142	1214	2351	15306	9482	24789	34864	31592	66456
	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
60	20.11	763	1813	901	1842	16826	9484	26311	33623	27255	60879
	0.96	0.97	0.85	0.74	0.78	1.10	1.00	1.06	0.96	0.86	0.92
120	20.10	751	1523	704	1619	23177	9484	32661	37788	25521	63310
	0.96	0.95	0.71	0.58	0.69	1.51	1.00	1.32	1.08	0.81	0.95
240	20.04	754	1490	676	1622	25194	9485	34679	39604	25577	65181
	0.96	0.96	0.70	0.56	0.69	1.65	1.00	1.40	1.14	0.81	0.98
480	20.74	756	1463	627	1849	26156	9481	35638	40430	27506	67936
	0.99	0.96	0.68	0.52	0.79	1.71	1.00	1.44	1.16	0.87	1.02
960	21.25	780	1658	849	2024	28242	9482	37724	44261	29089	73351
	1.02	0.99	0.77	0.70	0.86	1.85	1.00	1.52	1.27	0.92	1.10

Table C-5. 12 Mbytes of memory on Client.

Standard Deviations: 12 MegaBytes of Memory on the Client											
	Server	Elap	Faults		Page	FS Net I/O			Total Net I/O		
	Util	Time	Total	Swap	Outs	Read	Write	Total	Read	Write	Total
nocc	0.2	9.9	26	24	42	4.6	0.0	4.6	207	354	222
0.5M	0.5	13	26	45	126	21	2.3	24	190	1066	878
1M	0.4	3.6	43	31	25	28	0.0	28	339	213	421
2M	0.1	6.4	48	43	53	34	0.0	34	401	480	778
3M	0.8	159	7489	6900	521	83	0.6	83	62404	6107	68449
norm	0.6	27	140	126	55	620	2.3	623	1151	467	1180
60	0.2	13	17	14	42	567	2.3	569	719	323	411
120	0.1	4.0	9.3	2.0	57	247	2.3	249	198	497	503
240	0.3	9.8	30	33	88	480	0.6	480	492	746	1230
480	0.5	23	12	40	158	1302	0.0	1302	1295	1324	289
960	0.2	4.2	41	36	73	308	2.3	310	642	618	1216

Table C-6. Standard deviations with 12 Mbytes of memory on Client.

14 MegaBytes of Memory on the Client											
	Server	Elap	Faults		Page	FS Net I/O			Total Net I/O		
	Util	Time	Total	Swap	Outs	Read	Write	Total	Read	Write	Total
nocc	20.94	782	1684	440	1189	33942	9810	43752	50298	22560	72858
	1.10	1.13	1.05	0.62	0.72	4.52	1.04	2.58	2.27	0.89	1.53
0.5M	19.80	714	1090	317	1160	30279	9508	39787	41509	21706	63216
	1.04	1.03	0.68	0.45	0.70	4.03	1.00	2.34	1.87	0.86	1.33
1M	19.28	740	1309	475	1084	29368	9485	38854	42379	21087	63467
	1.02	1.07	0.82	0.67	0.66	3.91	1.00	2.29	1.91	0.83	1.34
2M	20.30	747	1411	586	1523	28908	9485	38394	42805	24796	67602
	1.07	1.08	0.88	0.83	0.92	3.85	1.00	2.26	1.93	0.98	1.42
3M	20.86	763	1690	870	2008	23471	9485	32957	39536	28862	68399
	1.10	1.10	1.05	1.23	1.22	3.13	1.00	1.94	1.78	1.14	1.44
4M	21.88	846	2810	1736	2674	19751	9485	29236	45093	34735	79829
	1.15	1.22	1.75	2.45	1.62	2.63	1.00	1.72	2.03	1.37	1.68
5M	36.79	1953	53122	49494	6211	15252	9485	24738	459903	76288	536191
	1.94	2.82	33.08	69.81	3.76	2.03	1.00	1.46	20.74	3.01	11.29
norm	18.99	692	1606	709	1653	7508	9469	16978	22171	25317	47489
	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
60	18.58	671	1355	555	1222	12478	9484	21963	25196	21761	46957
	0.98	0.97	0.84	0.78	0.74	1.66	1.00	1.29	1.14	0.86	0.99
120	18.26	678	1280	503	1194	14112	9484	23596	26265	21561	47826
	0.96	0.98	0.80	0.71	0.72	1.88	1.00	1.39	1.18	0.85	1.01
240	18.47	689	1288	509	1128	19127	9483	28610	31542	21149	52691
	0.97	1.00	0.80	0.72	0.68	2.55	1.00	1.69	1.42	0.84	1.11
480	18.47	691	1268	479	1095	19102	9485	28588	31342	20874	52216
	0.97	1.00	0.79	0.68	0.66	2.54	1.00	1.68	1.41	0.82	1.10
960	18.19	714	1291	491	1266	18912	9482	28394	31364	22315	53679
	0.96	1.03	0.80	0.69	0.77	2.52	1.00	1.67	1.41	0.88	1.13

Table C-7. 14 Mbytes of memory on Client.

Standard Deviations: 14 MegaBytes of Memory on the Client											
	Server	Elap	Faults		Page	FS Net I/O			Total Net I/O		
	Util	Time	Total	Swap	Outs	Read	Write	Total	Read	Write	Total
nocc	0.0	1.2	32	24	53	0.0	0.0	0.0	251	456	204
0.5M	0.4	2.3	16	16	101	16	2.3	17	142	859	999
1M	0.3	15	9.5	20	28	0.0	0.0	0.0	78	224	154
2M	0.1	5.5	31	27	130	6.1	0.0	6.1	280	1130	1388
3M	0.4	11	70	68	136	26	0.0	26	617	1185	1778
4M	0.4	13	172	136	50	64	0.6	64	1435	490	1924
5M	1.6	274	12915	11907	861	209	0.6	209	107462	10312	117774
norm	0.3	10	46	33	125	457	6.9	461	783	1036	664
60	0.2	9.6	47	28	22	714	2.3	714	942	189	1008
120	0.1	14	75	78	4.4	594	2.3	595	847	68	913
240	0.5	5.0	37	21	110	1865	2.1	1866	1667	992	2657
480	0.5	1.5	38	29	78	541	0.0	541	520	662	908
960	0.5	23	14	17	175	590	2.3	588	711	1485	1566

Table C-8. Standard deviations with 14 Mbytes of memory on Client.

16 MegaBytes of Memory on the Client											
	Server	Elap	Faults		Page	FS Net I/O			Total Net I/O		
	Util	Time	Total	Swap	Outs	Read	Write	Total	Read	Write	Total
nocc	19.82 1.09	721 1.09	1253 0.80	158 0.24	431 0.46	33938 4.75	9810 1.04	43748 2.63	46596 2.19	16029 0.84	62625 1.55
0.5M	17.40 0.96	667 1.01	699 0.45	33 0.05	88 0.10	30308 4.24	9506 1.00	39814 2.40	38135 1.79	12525 0.66	50660 1.25
1M	18.49 1.02	690 1.04	904 0.58	195 0.29	649 0.70	29423 4.11	9485 1.00	38909 2.34	38991 1.83	17278 0.90	56270 1.39
2M	20.06 1.10	691 1.04	1021 0.65	289 0.43	1011 1.09	28926 4.04	9484 1.00	38411 2.31	39497 1.85	20368 1.07	59865 1.48
3M	19.57 1.08	729 1.10	1277 0.82	459 0.68	1247 1.34	23469 3.28	9485 1.00	32955 1.98	35993 1.69	22289 1.17	58283 1.44
4M	20.22 1.11	736 1.11	1445 0.93	634 0.94	1662 1.79	19818 2.77	9484 1.00	29302 1.76	33647 1.58	25736 1.35	59384 1.47
5M	21.33 1.17	753 1.14	1707 1.09	868 1.29	2239 2.41	15924 2.23	9485 1.00	25410 1.53	31857 1.49	30589 1.60	62446 1.54
6M	22.27 1.22	870 1.31	2848 1.83	1724 2.56	2939 3.16	13099 1.83	9485 1.00	22585 1.36	38521 1.81	36759 1.92	75280 1.86
7M	38.73 2.13	2491 3.76	76889 49.29	72290 107.42	6968 7.50	8591 1.20	9485 1.00	18077 1.09	651019 30.53	88140 4.61	739160 18.28
norm	18.20 1.00	663 1.00	1560 1.00	673 1.00	929 1.00	7152 1.00	9460 1.00	16613 1.00	21324 1.00	19110 1.00	40435 1.00
60	17.34 0.95	651 0.98	1151 0.74	432 0.64	697 0.75	9155 1.28	9485 1.00	18641 1.12	19965 0.94	17157 0.90	37122 0.92
120	18.25 1.00	653 0.99	1190 0.76	455 0.68	913 0.98	11781 1.65	9485 1.00	21267 1.28	23056 1.08	19087 1.00	42144 1.04
240	16.95 0.93	660 1.00	993 0.64	307 0.46	342 0.37	14352 2.01	9485 1.00	23838 1.43	24013 1.13	14269 0.75	38282 0.95
480	16.27 0.89	651 0.98	979 0.63	291 0.43	405 0.44	14253 1.99	9485 1.00	23739 1.43	23799 1.12	14798 0.77	38597 0.95
960	16.50 0.91	652 0.98	987 0.63	298 0.44	607 0.65	14283 2.00	9485 1.00	23769 1.43	23923 1.12	16500 0.86	40424 1.00

Table C-9. 16 Mbytes of memory on Client.

Standard Deviations: 16 MegaBytes of Memory on the Client											
	Server	Elap	Faults		Page	FS Net I/O			Total Net I/O		
	Util	Time	Total	Swap	Outs	Read	Write	Total	Read	Write	Total
nocc	0.0	0.6	6.9	0.6	160	4.6	0.0	4.6	42	1348	1306
0.5M	0.7	4.4	24	12	55	32	2.3	34	215	471	276
1M	1.1	6.4	3.6	35	430	4.6	0.0	4.6	78	3649	3724
2M	0.4	1.5	25	27	41	61	1.7	63	263	335	178
3M	0.3	21	12	9.1	210	24	0.6	24	130	1785	1838
4M	0.2	18	35	38	58	51	2.3	54	313	478	183
5M	0.4	14	64	16	219	10	0.0	10	509	1783	1428
6M	0.5	13	441	341	108	69	0.0	69	3596	976	3727
7M	1.6	410	18882	17673	1061	380	0.0	380	157007	13399	170366
norm	0.9	5.2	21	5.5	299	18	2.3	16	125	2538	2472
60 pen	0.4	1.5	8.1	5.5	214	215	0.0	215	181	1791	1970
120 pen	0.9	9.3	51	40	287	456	0.0	456	663	2465	2874
240 pen	0.5	3.8	18	23	141	140	0.0	140	152	1204	1116
480 pen	0.5	3.1	21	20	123	415	0.0	415	506	1020	591
960 pen	0.6	4.0	11	20	306	197	0.0	197	203	2605	2785

Table C-10. Standard deviations with 16 Mbytes of memory on Client.

Physical Memory Management in a Network Operating System

Michael Newell Nelson

Abstract

This dissertation develops and measures methods of using large main memories to provide high performance in a network operating system. The dissertation covers three areas: file caching, virtual memory, and the interaction between the two. The work in all three areas was done as part of Sprite, a new network operating system that is being built here at Berkeley.

The first part of the dissertation presents results obtained through the development of the Sprite file system, which uses large main-memory file caches to achieve high performance. Sprite 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. Benchmark programs indicate that client caches allow diskless Sprite workstations to perform within 0-8% of workstations with disks. In addition, client caching reduces server loading by 50% and network traffic by 75%.

In addition to demonstrating the performance advantages of client caching, this dissertation also shows the advantage of writing policies that delay the writing of blocks from client caches to server caches and from server caches to disk. A measurement of 9 different writing policies on the client and 4 on the server shows that delayed-write policies provide the best performance in terms of network bytes transferred, disk

utilization, server utilization and elapsed time. More restrictive policies such as write-through can cause benchmarks to execute from 25% to 100% more slowly than if delayed-write policies are used.

The second part of this dissertation looks at the interaction between the virtual memory system and the file system. It describes a mechanism that has been implemented as part of Sprite that allows the file system cache to vary in size in response to the needs of the virtual memory system and the file system. This is done by having the file system of each machine negotiate with the virtual memory system over physical memory use. This variable-size cache mechanism provides better performance than a fixed-size file system cache of any size over a mix of file-intensive and virtual-memory-intensive programs.

The last part of this dissertation focuses on copy-on-write mechanisms for efficient process creation. It describes a simple copy-on-write mechanism that has been implemented as part of Sprite which is a combination of copy-on-write (COW) and copy-on-reference (COR). The COW-COR mechanism can potentially improve fork performance over copy-on-fork schemes from 10 to 100 times if many page copies are avoided. However, in normal use more than 70% of the pages have to be copied anyway. The overhead of handling the page faults required to copy the pages results in worse overall performance than copy-on-fork; with a more optimized implementation forks would be about 20% faster with COW-COR than with copy-on-fork. A pure COW scheme would eliminate 10 to 20 percent of the page copies required under COW-COR and would provide up to a 20% improvement in fork performance over COW-COR. However, because of extra cache-flushing overhead on machines with

virtually-addressed caches, pure COW may have worse overall performance than COW-COR on these types of machines.

Acknowledgements

There are many people that I would like to thank for making this dissertation possible. First, I would like to thank my committee, John Ousterhout, Domenico Ferrari, and Stuart Dreyfus for many useful comments that improved the presentation of my ideas. In particular, I would like to thank John Ousterhout who has helped me to become a much better writer and helped to steer me in the right direction. I have only to look back at the first drafts of my Masters report to see how far I have come because of John's help.

I especially want to thank the other members of the Sprite project, Brent Welch, Fred Douglis and Andrew Cherenson. Without their efforts Sprite would not exist. In particular, I want to thank Brent Welch. I had great fun working with Brent as we built the Sprite file system and I have learned a great deal from the technical discussions that we have had over the past several years.

The other members of the SPUR project have also been a great help. I feel very lucky to have been a part of the SPUR project. It not only provided me with the oppurtunity to learn about all facets of systems, from hardware to software, but it also allowed me to work with a great bunch of people.

Finally, I want to thank my girlfriend Betty. She has always been around to provide me with encouragement and was very patient during those numerous times when I had to stay late at school to work on my dissertation.

Table of Contents

List of Figures

List of Tables