Cache Coherence in Distributed Systems

Christopher A. Kent

December, 1987

digital **Western Research Laboratory** 100 Hamilton Avenue Palo Alto, California 94301 USA

This report is a slightly revised version of a thesis submitted in 1986 to the Department of Computer Sciences and Faculty of Purdue University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Partial support for this research was provided by NSF Grant DCR-8219178, and grants from Sun Microsystems and Digital Equipment Corporation.

This report was set in Times and Helvetica by the author using Scribe. The figures were produced using Adobe Illustrator and software of the author's design.

Times is a trademark and Helvetica is a registered trademark of Allied Corporation.
Illustrator is a trademark of Adobe Systems, Inc.
Scribe and UNILOGIC are registered trademarks of Unilogic, Ltd.
UNIX is a trademark of Bell Laboratories.
VAX is a trademark of Digital Equipment Corporation.

Copyright © 1986, 1987 by Christopher A. Kent. All Rights Reserved.

To the memory of my father and the persistence of my mother.

Abstract

Caching has long been recognized as a powerful performance enhancement technique in many areas of computer design. Most modern computer systems include a hardware cache between the processor and main memory, and many operating systems include a software cache between the file system routines and the disk hardware.

In a distributed file system, where the file systems of several client machines are separated from the server backing store by a communications network, it is desirable to have a cache of recently used file blocks at the client, to avoid some of the communications overhead. In this configuration, special care must be taken to maintain consistency between the client caches, as some disk blocks may be in use by more than one client. For this reason, most current distributed file systems do not provide a cache at the client machine. Those systems that do place restrictions on the types of file blocks that may be shared, or require extra communication to confirm that a cached block is still valid each time the block is to be used.

The Caching Ring is a combination of an intelligent network interface and an efficient network protocol that allows caching of all types of file blocks at the client machines. Blocks held in a client cache are guaranteed to be valid copies. We measure the style of use and performance improvement of caching in an existing file system, and develop the protocol and interface architecture of the Caching Ring. Using simulation, we study the performance of the Caching Ring and compare it to similar schemes using conventional network hardware.

1. Introduction

The principle of *locality of reference* [41, 42] is the observation that computer programs exhibit both spatial and temporal locality in referencing objects (such as memory words or disk blocks). Temporal locality means that objects to be referenced in the near future are likely to have been in use recently. Spatial locality means there is a high probability that objects needed in the near future can be located near the objects currently in use. Less expensive access to recently used objects increases program performance.

A *cache* is a device that exploits both spatial and temporal locality. It automatically maintains a copy of recently referenced objects in a higher-performance storage medium than that in which the objects themselves are stored. The program operates on copies that reside in the cache instead of operating directly on the objects, with a resultant increase in performance. The cache is responsible for propagating changes to the copies back to the stored objects. Figure 1-1 shows the difference between systems with and without a cache. The function $f(a)$ describes the cost of accessing an object in the storage module. The function $f'(a)$ describes the cost of accessing an object in the storage system that combines the storage module and the cache. Exploiting locality of reference allows the values of $f'(a)$ to be less than $f(a)$, for most a.

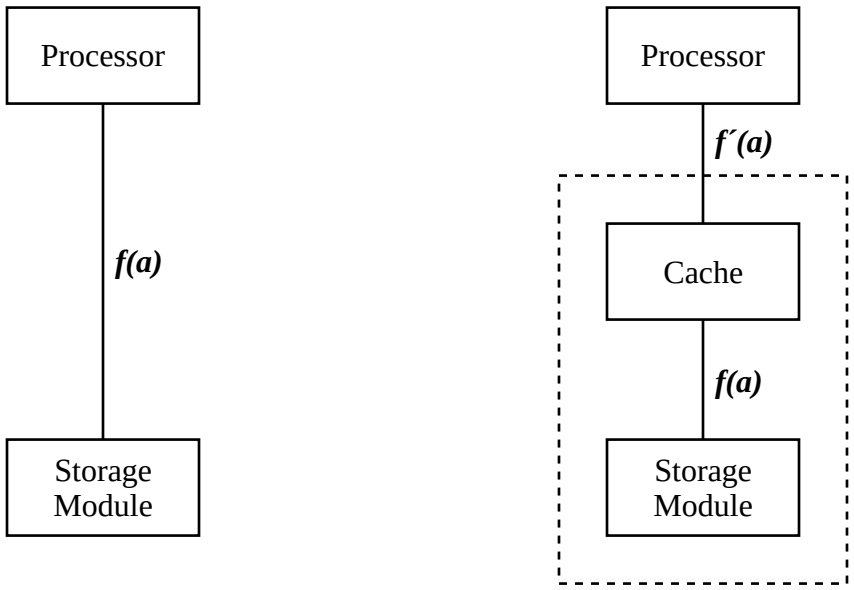


Figure 1-1: Systems without and with a cache

A cache system is *coherent* if, whenever an object is read, the returned value is the one most recently written. A system with only one cache is coherent because there is only one path to and from the objects — through the single cache. In a system with N processing elements, $N>1$, sharing a common pool of objects, there are N paths to and from the objects. If each path contains a cache holding copies of some of the objects, copies of the same objects can exist in more than one cache. A mechanism to propagate updates from one cache to another is necessary to insure coherence.

Several cache coherence mechanisms exist for systems of processors and caches that share a common block of main memory. The machines operate in an environment where systems are tightly coupled, highly synchronous, with reliable communication paths that are as fast as those in the memory subsystem.

It is increasingly common to connect processors in more loosely coupled systems. The only communication path between processors and the resources they share is a communications network [55] that has transmission speeds several orders of magnitude slower than main memory. We describe a mechanism for cache coherence in these systems.

1.1. Background

The idea that a computer should use a memory hierarchy dates back to at least the early portion of the 20th century; it is suggested in the pioneering paper of von Neumann *et al.* [59]. The motivation for a storage hierarchy in a processor is economic. The performance and cost of various storage technologies varies widely. Usually, the fastest and most expensive technology is used for the registers in the processor. Ideally, one would like to execute programs as if all data existed in the processor registers. When more data are required, larger, lower-cost storage technologies are used for data and instruction storage, proceeding from fast semiconductor memory, to slower semiconductor memory, to magnetic disk storage, and finally to magnetic tape or other archival storage media.

	Registers	Cache Memories	Main Memories	Secondary ‘‘Core’’	Backing Stores	Archival Stores
Access time (ns)	10	10	100	10^3	10^7	10^8
Transfer time (ns)	10	10	100	10^3	10^4	10^5
Addressable units	2^0-2^{10}	2^5-2^{14}	2^{16}-2^{24}	2^{20}-2^{26}	2^{25}-2^{30}	2^{25}-2^{40}
Technology	Semiconductor	Semiconductor	Semiconductor	Semiconductor	Magnetic	Magnetic Optical

Table 1-1: Characteristics of various memory technologies

A memory level becomes a performance bottleneck in the system when the device accessing the memory can generate access requests faster than the memory can service them. By adding a small memory that fits the speed of the device, and using the small memory properly, one can achieve a significant increase in performance. Copies of objects in the slower memory can be temporarily placed in the faster memory. The accessor then operates only on the faster memory,

eliminating the bottleneck while retaining the advantages of the larger, slower memory. The objects to be placed in the faster memory are selected to minimize the time the accessor spends waiting for objects to be copied back and forth. The overhead of the copying operations is offset by the performance advantage gained through repeated references to copies in the faster memory.

The process of selecting which objects to move between levels in the memory hierarchy was first automated in the ATLAS demand paging supervisor [22, 31]. The ATLAS machine had two levels in its hierarchy – core memory and a drum. The demand paging supervisor moved memory between the core memory and the drum in fixed-sized groups called *pages*. An automatic system was built that allowed users to view the combination of the two storage systems as a single level (*i.e.*, the operation of the mechanism was *transparent*). This ‘‘one-level storage system’’ incorporated an automatic learning program that monitored the behavior of the main program and attempted to select the correct pages to move to and from the drum.

The ATLAS one-level store was the first example of *virtual memory* – a mechanism that expands the space available for programs and data beyond the limits of physical main memory. In fact, this mechanism is simply an environment where programs and data are stored in their entirety outside of main memory, and main memory is a cache for the processor.

The IBM 360/85 [12] incorporated the first application of this idea to high speed devices. The term cache was introduced in [34] to describe the high speed associative buffer in the memory subsystem of the 360/85. This buffer was used to hold copies of recently referenced words from main memory.

So far, we have discussed caching only in the context of managing the memory space of a processor. Many other forms of caching exist. Caches of recently referenced disk blocks held in main memory increase overall disk system performance [58, 51]. Digital typesetters cache font information to reduce the amount of data transmitted over the communications channel from the host computer [23]. Program execution times can be enhanced by precomputing and caching values of expensive functions (*e.g.*, trigonometric functions) and using table lookup rather than run-time computation. Applicative language systems cache the result values of expressions to avoid needless recomputation [30].

1.2. Caching in computer systems

1.2.1. Single cache systems

We now examine the most common type of cache in computer systems – that found in a uniprocessor system between the central processor unit (CPU) and main memory. For example, with a CPU cycle time of 60ns and memory access time of 150ns, there is a large disparity between the relative speeds of the CPU’s need to access memory and the ability of the memory system to satisfy requests. In a configuration where the CPU directly accesses main memory, the CPU will waste two to three cycles per memory reference, waiting for memory to respond. (See Figure 1-2.) Wiecek measured CPU instruction set usage in a time-sharing environment on a VAX-11 processor. This study showed that 50 – 60% of the executed instructions read memory, and 30 – 40% wrote memory [60]. For the VAX-11, the average number of memory references

per instruction is 1.18. McDaniel found similar results in his study of instruction set usage on a personal workstation in [35]. Thus, in our example, lack of a cache would cause the CPU to wait an average of 2.875 cycles for each memory reference, allowing an average 35% processor utilization.

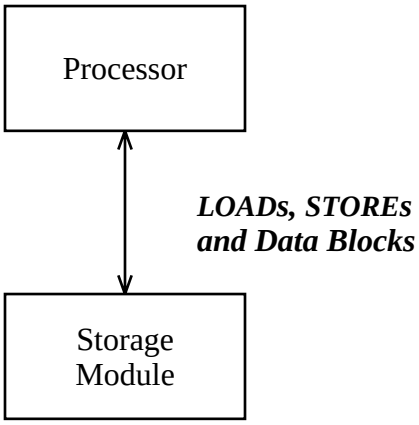


Figure 1-2: Uniprocessor without cache

A cache is introduced as a small amount of high-speed memory between the CPU and memory. (See Figure 1-3.) The cache memory has an access time comparable to the cycle time of the CPU. The cache hardware contains control logic to manage a directory of the locations stored in the cache, as well as the additional memory for cached values. When the CPU performs a LOAD from memory, the cache first searches its directory for a copy of the desired location. If found, the cache returns a copy of the contents. If the location is not cached, the CPU waits while the cache fetches the location from slower main memory, copies the value into the cache, and returns it to the CPU. In practice, the cache manipulates *blocks* of memory consisting of several contiguous words, to reduce directory overhead. When the contents of a particular memory word must be copied into the cache, the entire surrounding block is copied into memory.

The amount of storage in a cache is finite. If no free space remains in the cache to hold the contents of the block just fetched, one of the currently held locations must be selected for removal. Most systems remove the least recently used object under the assumption that the entries in the cache not used for the longest period are the least likely to be re-used in the near future.

When the CPU executes a STORE to memory, the cache checks its directory for a copy of the referenced location. If found, the cache updates the copy, and does not write to main memory. The cache writes the update to main memory when the block containing the location is selected for removal from the cache. This minimizes the average access delays on a STORE. A cache that uses this method of propagating updates is called a *write-back* cache.

A *write-through* cache copies the update to main memory at the same time that the copy in the cache is modified. Updating main memory immediately generates more traffic to the memory, since every STORE instruction generates a main memory reference. While the main memory executes the STORE, the CPU is blocked from making any other memory references.

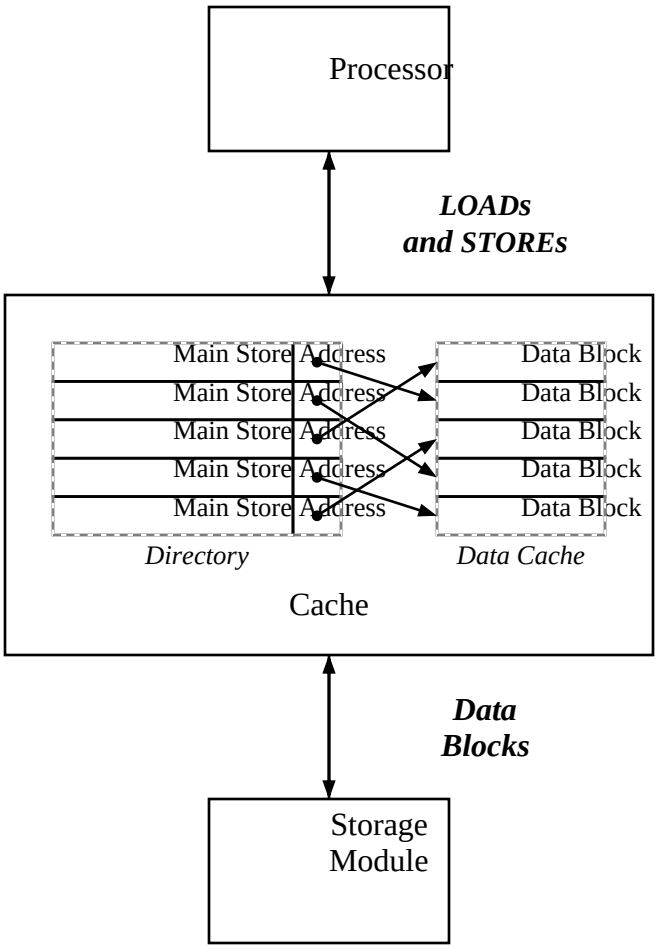


Figure 1-3: Uniprocessor with cache/directory

In a disk block cache, the file system portion of the operating system maintains copies of recently accessed disk blocks. All disk operations search the disk cache for the desired block before accessing the disk, yielding greatly improved disk subsystem performance.

1.2.2. Multiple cache systems

In a tightly coupled multiprocessor, *N* CPUs share the same main storage. If each CPU has a private cache, there are now multiple access paths to main memory, and care must be taken to preserve coherence.

Let us examine a specific example consisting of two tasks, T_1 and T_2, running on processors P_1 and P_2 with caches C_1 and C_2 (see Figure 1-4). Let *a* be the address of a main memory location that is referenced and modified by both tasks. A modification of *a* by T_1 is completed in C_1 but not returned to main memory. Thus, a subsequent LOAD by T_2 will return an obsolete value of *a*.

Even a write-through cache does not insure coherence. After both T_1 and T_2 have referenced *a*, subsequent references will be satisfied by the cache, so a new value written to main memory by one processor will not be seen by the other.

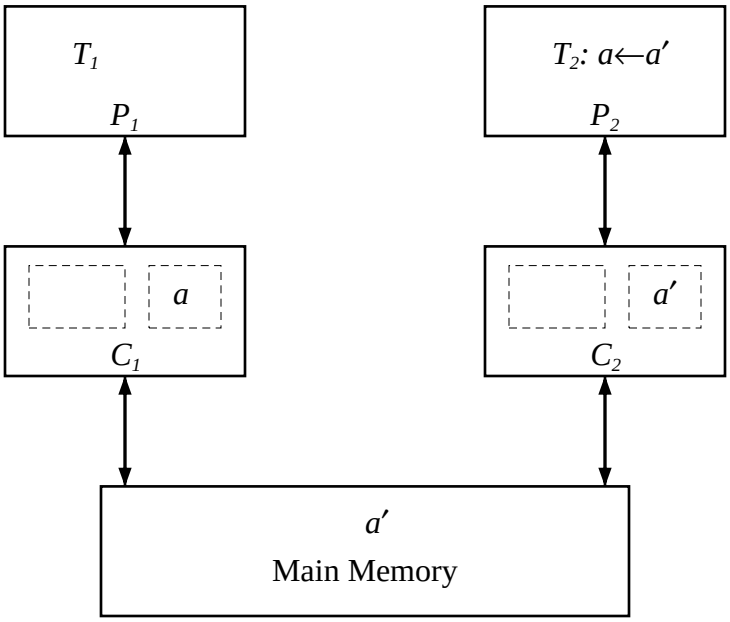


Figure 1-4: Two competing caches after T_2 modifies a

Sharing a single cache between the N processors eliminates coherence problems. But such a cache is likely to be a performance bottleneck. The demands on it would be N times that of a single cache, because it would handle all the data requests for each of the N processors. Also, with all the processors sharing a single cache, much of the history of reference for each processor will be lost, and with it, much of the performance advantage.

A mechanism is necessary to couple the caches and actively manage their contents. Several such mechanisms have been devised, relying on the removal of memory blocks from caches whenever there is a risk that their contents may have been modified elsewhere in the system. Inappropriate (too frequent) removal will result in greatly decreased performance, because more time will be spent waiting for blocks to be loaded into the cache.

1.2.2.1. Tang's solution

Tang presented the first practical design for a multicache, multiprocessor system [56]. The cache structure for each processor is the same as in a single cache system, with some additional features to facilitate communication among caches.

Tang makes a distinction between cache entries that are *private* and *shared*. An entry is private if it has been modified with respect to main memory, or is about to be modified by the corresponding processor. A private entry can exist in only one cache so that, at any instant, there is only one version of the data in the system.

A shared entry has not been modified by any processor. It is allowed to exist simultaneously in several caches in the system, to allow 'read only' data to be accessed more efficiently.

Communication among the caches is controlled by a *storage controller* that maintains a central directory of the contents of all the caches. All communication between the caches and main memory passes through this storage controller. When the cache fetches a memory location,

the cache controller alters the cache directory to show that a copy of the fetched memory location is present in the cache. The storage controller also alters the central directory to show that a copy of the memory location is in the cache.

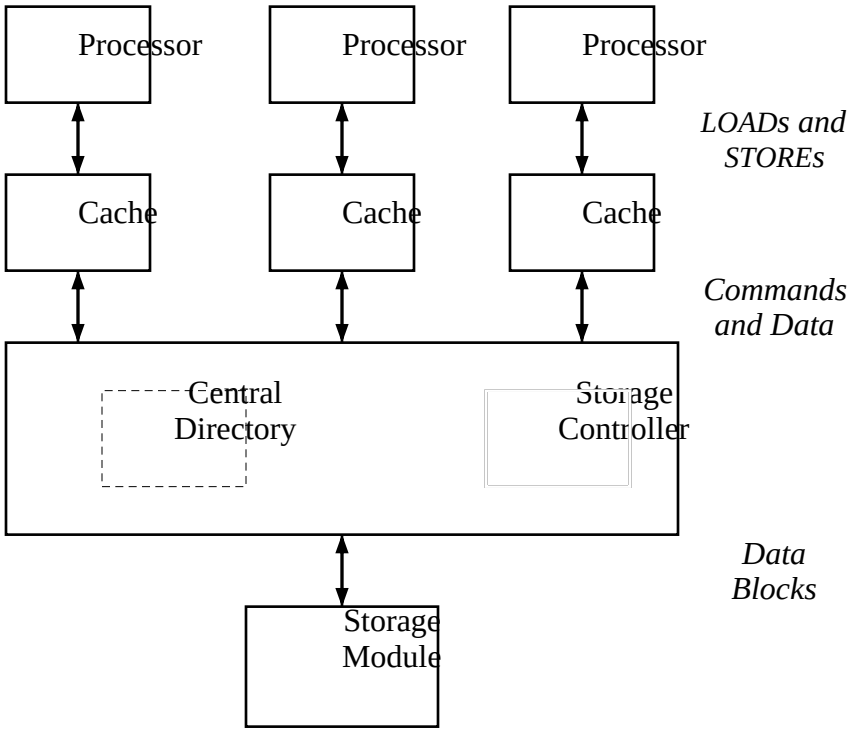


Figure 1-5: Tang's multiprocessor

The normal LOAD and STORE instructions between the processor and the caches are augmented with new commands sent from the caches to the storage controller and from the controller to the caches. Using these commands, the storage controller ensures that the cache system remains coherent. The controller converts shared blocks to private blocks when a processor is about to write a location, then converts private blocks to shared blocks when another processor attempts to read a location previously marked as private.

1.2.2.2. The *Presence Bit* solution

The Presence Bit solution for multicache coherence [44] is similar to Tang's solution. Instead of duplicating each cache's directory in a central directory, main memory has N+1 extra bits per block. N of these bits correspond to the caches in the system, and are set if and only if the corresponding cache has a copy of the block. The remaining bit is reset if and only if the contents of the main memory block are identical to all cached copies. Each cache has, associated with each block, a bit that is set to show that this cache has the only copy of this block.

The commands that are executed between the caches and main memory are essentially identical to those between Tang's storage controller and caches. The advantage of the Presence Bit solution is lower overhead per memory block.

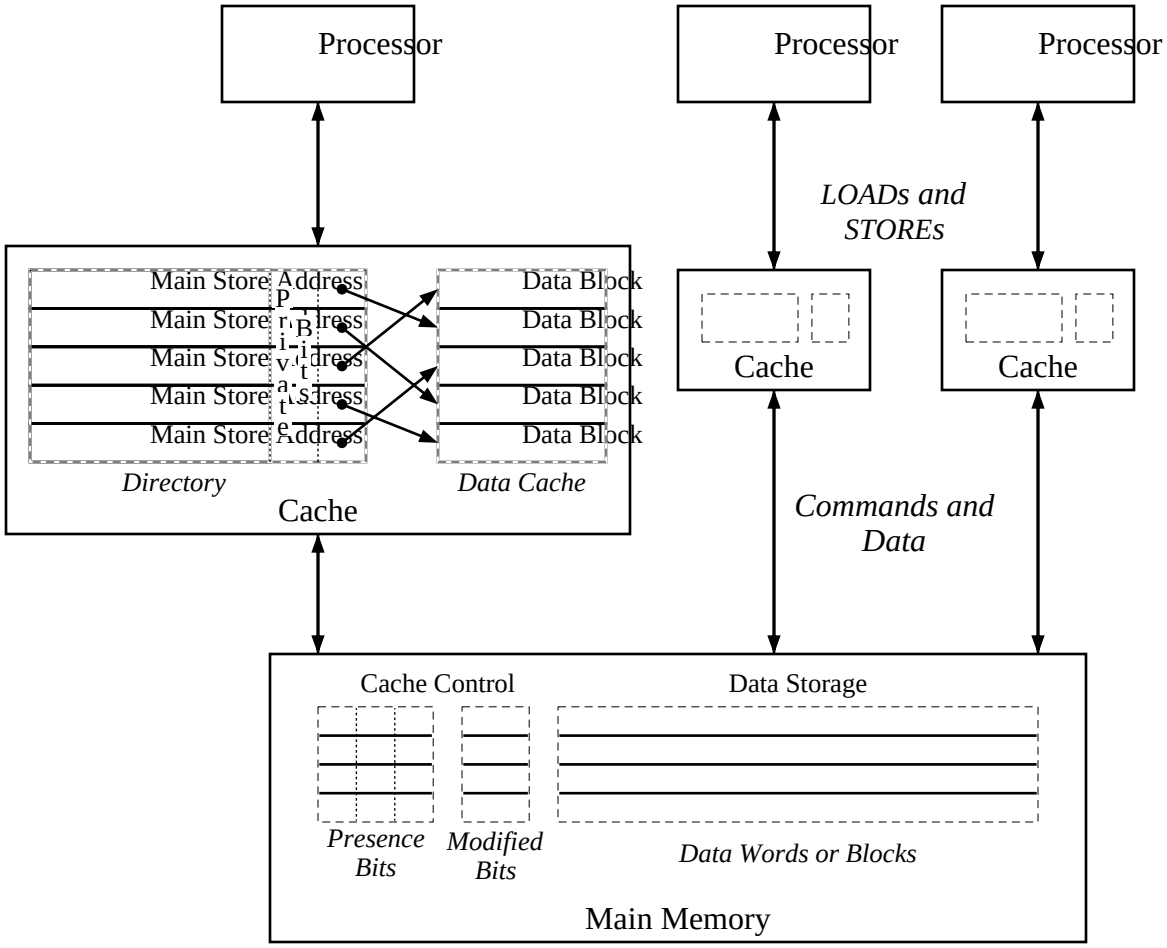


Figure 1-6: Presence Bit solution

1.2.2.3. The *Snoopy* or *Two-Way* cache

A *snoopy* cache is one that watches all transactions between processors and main memory and can manipulate the contents of the cache based on these transactions.

Three kinds of snoopy cache mechanisms have been proposed. A *write-through* strategy [2] writes all cache updates through to the main memory. Caches of the other processors monitor these updates, and remove held copies of memory blocks that have been updated.

A second strategy is called *write-first* [24]. On the first STORE to a cached block, the update is written through to main memory. The write forces other caches to remove any matching copies, thus guaranteeing that the writing processor holds the only cached copy. Subsequent STOREs can be performed in the cache. A processor LOAD will be serviced either by the memory or by a cache, whichever has the most up-to-date version of the block.

The third strategy is called *ownership*. This strategy is used in the SYNAPSE multiprocessor [54]. A processor must ‘‘own’’ a block of memory before it is allowed to update it. Every main memory block or cache block has associated with it a single bit, showing whether the device holding that block is the block’s owner. Originally, all blocks are owned by the shared

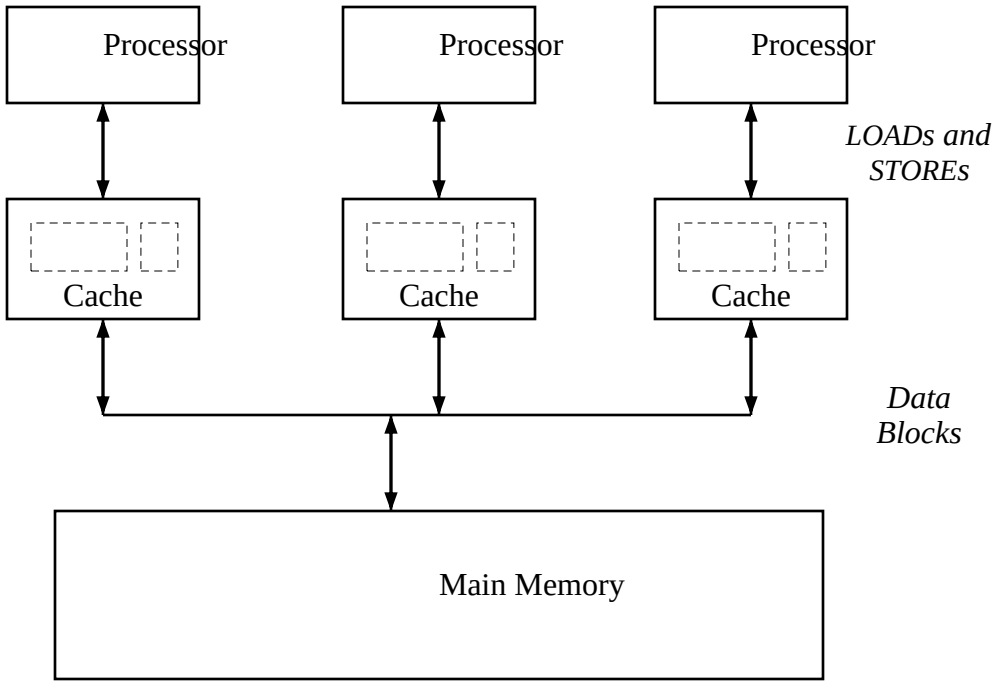


Figure 1-7: Snoopy Cache organization

main memory. When a cache needs to fetch a block, it issues a *public* read, to which the owner of the block responds by returning a current copy. Ownership of the block does not change.

When a processor *P* desires to modify a block, ownership of the block is transferred from the current owner (either main memory or another cache) to *P*'s cache. This further reduces the number of STOREs to main memory. All other caches having a copy of this block notice the change in ownership and remove their copy. The next reference causes the new contents of the block to be transferred from the new owner. Ownership of a block is returned to main memory when a cache removes the block in order to make room for a newly accessed block.

A snoopy cache has the smallest bit overhead of the discussed solutions, but the communication path must be fast and readily accessible by all potential owners of memory blocks. Operations between owners are tightly synchronized. The other solutions allow the caches and memory to be more loosely coupled, but rely on a central controller for key data and arbitration of commands.

1.3. Distributed Cache Systems

With the continuing decline in the cost of computing, we have witnessed a dramatic increase in the number of independent computer systems. These machines do not compute in isolation, but rather are often arranged into a *distributed system* consisting of single-user machines (*workstations*) connected by a fast local-area network (LAN). The workstations need to share resources, often for economic reasons. In particular, it is desirable to provide the sharing of disk files. Current network technology does not provide sufficiently high transfer rates to allow a processor's main memory to be shared across the network.

Management of shared resources is typically provided by a trusted central authority. The workstations, being controlled by their users, cannot be guaranteed to be always available or be fully trusted. The solution is to use *server* machines to administer the shared resources. A *file server* is such a machine that makes available a large quantity of disk storage to the *client* workstations. The clients have little, if any, local disk storage, relying on the server for all long-term storage.

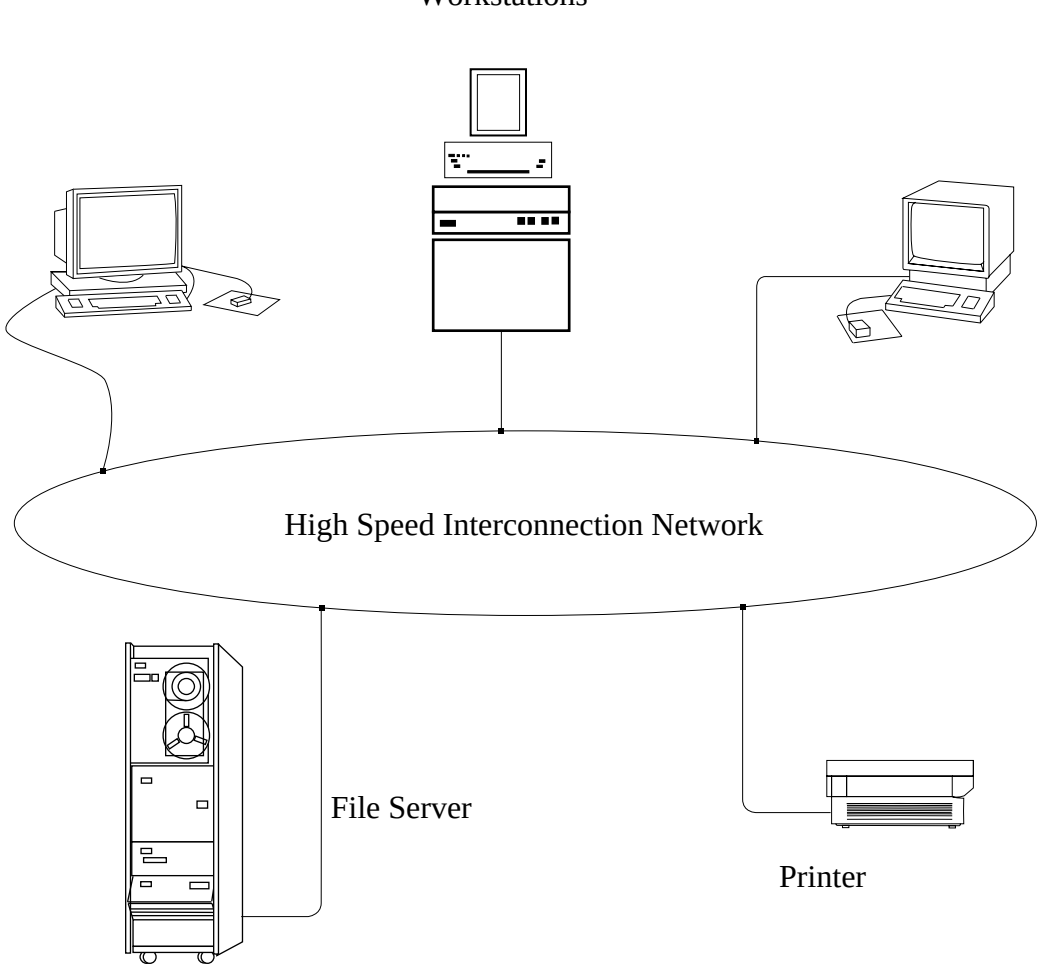


Figure 1-8: Typical distributed system

The disparity in speeds between processor and remote disk make an effective caching scheme desirable. However, no efficient, fully transparent solutions exist for coherence in a distributed system. Distributed data base systems [13] use locking protocols to provide coherent sharing of objects between clients on a network. These mechanisms are incorporated into the systems at a very high level, built on a non-transparent network access mechanism, and are not concerned with performance improvements. We prefer a solution that is integral to the network file system, and provides the extra performance of an embedded cache.

Several distributed file systems that include some form of caching exist. The next sections present a survey of their characteristic features.

1.3.1. Sun Microsystems' Network Disk

Sun Microsystems' Network Disk [48] is an example of the simplest form of sharing a disk across the network. The client workstation contains software that simulates a locally attached disk by building and transmitting command packets to the disk server. The server responds by transferring complete disk blocks. The client has a disk block caching system, keeping the most recently used blocks in main memory. The server's disk is partitioned into as many logical disks as there are clients. No provision is made for communication among clients' caches; clients can only share read-only data.

1.3.2. CFS

The Cedar experimental programming environment [8] developed at the Xerox Palo Alto Research Center supports a distributed file system called *CFS* [9]. Each of the Cedar workstations has a local disk, and this disk can be used for local private files or shared files copied from a remote file server.

A file to be shared is first created as a file on the local disk. To make the file available for sharing, the client transfers it to the remote file server. A client on another workstation can then share the file by copying it to his local disk. The portion of the disk not occupied by local files is used as a cache for remote files. Files are transferred between client and server as a whole.

Coherence of the cache of files on local disk is guaranteed because shared files may not be modified. To update the contents of a shared file, a new version which reflects the updated information is created on the server. This version has the same name as the original file upon which it is based; only the version numbers differ. Thus, all cached copies of a particular version of a file contain the same data. It is possible, however, to have a cached copy of a file that does not reflect the latest version of the file.

1.3.3. The ITC Distributed File System

The Information Technology Center of Carnegie-Mellon University is building a campus-wide distributed system. *Vice*, the shared component of the distributed system, implements a distributed file system that allows sharing of files [28]. Each client workstation has a local disk, which is used for private files or shared files from a Vice file server. Shared files are copied as a whole to the local disk upon open, and the client operating system uses this local copy as a cache to satisfy disk requests. In this regard, the ITC caching mechanism is similar to that of CFS.

Cache validation is currently performed by the client querying the server before each use of the cached copy. A future implementation will allow the server to invalidate the client's cached copy. Changes in the cached copy are stored back to the server when the file is closed.

1.3.4. Sun Microsystems Network File System

Sun Microsystems' second generation distributed file system [49] allows full sharing of remote files. Client workstations forward disk block requests to a file server. There, the appropriate disk is read, and the data is returned to the client. The client can cache the returned data and operate from the cache.

Changed blocks are written back to the server on file close. At that time, all blocks associated with the file are flushed from the cache. Each entry in the cache has an associated timeout; when the timeout expires, the entry is automatically removed from the cache. Cached files also have an associated timestamp. With each read from a remote file, the server returns the timestamp information for that file. The client compares the current timestamp information with the information previously held. If the two differ, all blocks associated with the file are flushed from the cache and fetched again.

Coherence between client caches is achieved by ensuring that each client is coherent with the server's cache. However, because the changes made by a client are not seen until the client closes the file, there can be periods of time when two clients caching the same file have different values for the same cached block.

1.3.5. Apollo DOMAIN

The Apollo DOMAIN operating system embodies a distributed file system that allows location transparent access of objects [16]. Each workstation acts as a client, and may act as a server if it has local disk storage. Main memory is used as a cache for local and remote objects in the file system.

The distributed file system does nothing to guarantee cache coherence between nodes. Client programs are required to use locking primitives provided by the operating system to maintain consistency of access. The designers decided that providing an automatic coherence mechanism in the cache system was counter to their efficiency goals.

1.4. Memory systems *vs.* Distributed systems

Let us return to the memory subsystem solutions and examine the fundamental assumptions that render them inappropriate for a distributed system environment. All the solutions require reliable communications between the various components of the memory hierarchy. In addition, the snoopy cache requires not only direct communications, but reliable receipt of broadcast messages. Reliable communication is achieved by building synchronous systems that allocate some portion of the cycle time to doing nothing but receiving messages.

Because electrical disturbances may occur on local area networks, it is not possible to achieve reliable communications without considerable overhead. Reliable stream-oriented protocols like TCP [57] are required for point-to-point connections. A broadcast network such as the Ethernet [61], on which hosts have the ability to receive all transmissions on the medium (*i.e.*, hosts can be *promiscuous*), would seem ideal for a snoopy cache implementation. However, the Ethernet provides only ''best effort'' delivery. To provide reliable broadcast communications, a specialized protocol must be employed [10, 43], with much overhead. Even if cheap reliable broadcast were available, processing every message on the network imposes a high load on systems.

Another problem is granularity of reference and locking. In a memory system, requests for a particular block are serialized by hardware. The hardware allows only a single processor access to a given main memory block at any time. While one processor is accessing the block, other processors must stall, waiting their turn. However, the time involved is small, typically one or two CPU cycle times, depending on the instruction that generated that access.

In a distributed system, in the time that it takes processor P_A to send a message indicating a desire for a private read or an update, processor P_B may be updating its shared copy of that same block (which should actually now be private to P_A and have been removed from P_B's cache). Because a distributed system is asynchronous, access to shared blocks must be serialized by explicit locking mechanisms. These mechanisms involve sending messages between clients and servers and encounter large communication delays. Because the communications delays are large, the size of the blocks that are locked are large, to maximize the ratio of available data to locking overhead. Unlike a memory system, locks are held for a long time (relative to processor cycle time), and a processor may have to stall for a long time waiting for a shared block.

1.5. Our Solution: The Caching Ring

We propose a network environment that provides transparent caching of file blocks in a distributed system. The user is not required to do any explicit locking, as in traditional database concurrency control algorithms, nor is there any restriction on the types of files that can be shared.

The design is inspired by both the snoopy memory cache and the Presence Bit multicache coherence solution. Caches that hold copies of a shared file object monitor all communications involving that object. The file server maintains a list of which caches have copies of every object that is being shared in the system, and issues commands to maintain coherence among the caches.

Our environment retains many of the benefits of low-cost local area networks. It uses a low-cost communications medium and is easily expandable. However, it allows us to create a more efficient mechanism for reliable broadcast or multicast than is available using ‘‘conventional’’ methods previously mentioned. Operation of the caches relies on an intelligent network interface that is an integral part of the caching system.

The network topology is a ring, using a token-passing access control strategy [19, 18, 45]. This provides a synchronous, reliable broadcast medium not normally found in networks such as the Ethernet.

1.5.1. Broadcasts, Multicasts, and Promiscuity

Because it is undesirable to burden hosts with messages that do not concern them, a multicast addressing mechanism is provided. Most multicast systems involve the dynamic assignment of arbitrary *multicast identifiers* to groups of destination machines (*stations*) by some form of centralized management. Each station must use a costly lookup mechanism to track the current set of identifiers involving the station. On every packet receipt, the station must invoke this lookup mechanism to determine if the packet should be copied from the network to the host.

The addressing mechanism in our network allows us to avoid the overhead of multicast identifier lookup, and avoid the requirement of central management. Addressing is based on an *N*-bit field of recipients in the header of the packets. Each station is statically assigned a particular bit in the bit field; if that bit is set, the station accepts the packet and acts on it. Each recipient resets its address bit before forwarding the packet. This provides positive acknowledgement of reception. Retransmission to those hosts that missed a packet requires minimal computation. Thus, it

is possible to efficiently manage 2^n multicast groups with reliable one ring cycle delay delivery, as opposed to n point-to-point messages for a multicast group of size n, which is typical for reliable multicast protocols on the Ethernet.

Missed packets are a rare problem, because the token management scheme controls when packets can arrive, and the interface hardware and software is designed to be always ready to accept the next possible packet, given the design parameters of the ring. The primary reasons for missed packets are that stations crash or are powered down, or packets that are damaged due to ring disturbances.

1.5.2. Ring Organization

Traffic on the ring consists of two types of packets: command and data. Each station introduces a fixed delay of several bit times to operate on the contents of a packet as it passes by, possibly recording results in the packet as it leaves. Command packets and their associated operations are formulated to keep delays at each station to a minimum constant time. If, for example, the appropriate response is to fetch a block of data from a backing store, the command packet is released immediately, and the block is then fetched and forwarded in a separate data packet.

The interface contains the names of the objects cached locally, while the objects themselves are stored in memory shared between the interface and the host. Thus, status queries and commands are quickly executed.

1.6. Previous cache performance studies

Many of the memory cache designs previously mentioned are paper designs and were never built. Of the ones that were built, only a few have been evaluated and reported on.

Bell *et al.* investigated the various cache organizations using simulation during the design process of a minicomputer [5]. Their results were the first comprehensive comparison of speed *vs.* cache size, write-through *vs.* write-back, and cache line size, and provided the basis for much of the ‘‘common knowledge’’ about caches that exists today.

Smith has performed a more modern and more comprehensive survey of cache organizations in [50]. This exhaustive simulation study compares the performance of various cache organizations on program traces from both the IBM System/360 and PDP-11 processor families.

A number of current multiprocessors use a variation of the snoopy cache coherence mechanism in their memory system. The primary differences are how and when writes are propogated to main memory, whether misses can be satisfied from another cache or only from memory, and how many caches can write a shared, cached block. Archibald and Baer have simulated and compared the design and performance of six current variations of the snoopy cache for use in multiprocessors [4]. In [33], Li and Hudak have studied a mechanism for a virtual memory that is shared between the processors in a loosely coupled multiprocessor, where the processors share physical memory that is distributed across a network and part of a global address space.

1.7. Previous file system performance studies

There has been very little experimental data published on file system usage or performance. This may be due to the difficulty of obtaining trace data, and the large amounts of trace data that is likely to result. The published studies tend to deal with older operating systems, and may not be applicable in planning future systems.

Smith studied the file access behavior of IBM mainframes to predict the effects of automatic file migration [52]. He considered only those files used by a particular interactive editor, which were mostly program files. The data were gathered as a series of daily scans of the disk, so they do not include files whose lifetimes were less than a day, nor do they include information about the reference patterns of the data within the files. Stritter performed a similar study covering all files on a large IBM system, scanning the files once a day to determine whether or not a given file had been accessed [53]. Satyanarayanan analyzed file sizes and lifetimes on a PDP-10 system [46], but the study was made statically by scanning the contents of disk storage at a fixed point in time. More recently, Smith used trace data from IBM mainframes to predict the performance of disk caches [51].

Four recent studies contain UNIX measurements that partially overlap ours: Lazowska *et al.* analyzed block size tradeoffs in remote file systems, and reported on the disk I/O required per user [32], McKusick *et al.* reported on the effectiveness of current UNIX disk caches [36], and Ousterhout *et al.* analyzed cache organizations and reference patterns in UNIX systems [38]. Floyd has reported extensively on short-term user file reference patterns in a university research environment [21]. We compare our results and theirs in Section 3.6.

1.8. Plan of the thesis

After defining the terminology used in the rest of the work, we examine the use and performance of a file system cache in a uniprocessor, first with local disks and then with remote disks. We then proceed to investigate applications of the caching ring to a multiple CPU, multiple cache system under similar loads.

Finally, we explore other areas in distributed systems where these solutions may be of use, as well as methods of adapting the ideal environment of the caching ring to conventional networking hardware.

2. Definitions and Terminology

As mentioned in Chapter 1, we are concerned with caching in distributed systems, and in particular, in file systems. In this chapter, we define the fundamental components of distributed systems, file systems, and cache systems, as well as a notation for discussing cache operations. Since much of our work is centered around the UNIX file system, we briefly introduce some details of that implementation.

2.1. Fundamental components of a distributed system

Physically, a distributed system consists of a set of *processors*, with a collection of local *storage mechanisms* associated with each processor. A processor is able to execute programs that access and manipulate the local storage, where the term *process* denotes the locus of control of an executing program [17]. In addition, an *interconnection network* connects the processors and allows them to communicate and share data via exchange of *messages* [55]. These messages are encapsulated inside *packets* when transmitted on the network.

2.1.1. Objects

We conceptually view the underlying distributed system in terms of an *object model* [29] in which the system is said to consist of a collection of *objects*. An object is either a physical resource (*e.g.*, a disk or processor), or an abstract resource (*e.g.*, a file or process). Objects are further characterized as being either *passive* or *active*, where passive objects correspond to stored data, and active objects correspond to processes that act on passive resources. For the purposes of this thesis, we use the term *object* to denote only *passive objects*.

The objects in a distributed system are partitioned into *types*. Associated with each object type is a *manager* that implements the object type and presents *clients* throughout the distributed system with an interface to the objects. The interface is defined by the set of *operations* that can be applied to the object.

An object is identified with a *name*, where a name is a *string* composed of a set of *symbols* chosen from a finite *alphabet*. For this thesis, all objects are identified by *simple* names, as defined by Comer and Peterson in [39]. Each object manager provides a *name resolution mechanism* that translates the name specified by the client into a name that the manager is able to *resolve* and use to access the appropriate object. Because there is a different object manager for each object type, two objects of different types can share the same name and still be properly identifiable by the system. The collection of all names accepted by the name resolution mechanism of a particular object manager constitutes the *namespace* of that object type.

An object manager can treat a name as a simple or *compound* name. A compound name is composed of one or more simple names separated by special *delimiter* characters. For example, an object manager implementing words of shared memory might directly map the name provided by the client into a hardware memory address. This is known as a *flat* namespace. On the other hand, an object manager implementing a *hierarchical* namespace in which objects are grouped together into *directories* of objects, provides a mechanism for adding structure to a collection of objects. Each directory is a mapping of simple names to other objects. During the evaluation of a name, a hierarchical evaluation scheme maintains a *current evaluation directory* out of the set of directories managed by the naming system. Each step of hierarchical name evaluation includes the following three steps:

1. Isolate the next simple name from the name being evaluated.
2. Determine the object associated with the simple name in the current evaluation directory.
3. (If there are more name components to evaluate) Set the current evaluation directory to the directory identified in Step 2, and return to Step 1.

2.1.2. Clients, managers, and servers

Clients and managers that invoke and implement operations are physically implemented in terms of a set of cooperating processes. Thus, they can be described by the model of distributed processing and concurrent programming of *remote procedure calls* [37].

In particular, we divide processors into two classes: *client machines* that contain client programs, and *server machines* that contain object manager programs. Each server has one or more attached storage devices, which it uses as a *repository* for the data in the objects implemented by the object managers. In our system, there are N client machines, denoted $C_1 \ldots C_N$, and one server machine, denoted S.

2.2. Caches

Caches are storage devices used in computer systems to temporarily hold those portions of the contents of an object repository that are (believed to be) currently in use. In general, we adhere to the terminology used in [50], with extensions from main memory caching to caching of generalized objects. A cache is optimized to minimize the *miss ratio*, which is the probability of not finding the target of an object reference in the cache.

The cache has three components: a collection of fixed-sized *blocks* of object storage (also known in the literature as *lines*); the *cache directory*, a list of which blocks currently reside in the cache, showing where each block is located in the cache; and the *cache controller*, which implements the various algorithms that characterize the operation of the cache.

Information is moved between the cache and object repository one block at a time. The *fetch algorithm* determines when an object is moved from the object repository to the cache memory. A *demand fetch* algorithm loads information when it is needed. A *prefetch* algorithm attempts to load information before it is needed. The simplest prefetch algorithm is *readahead*: for each block fetched on demand, a fixed number of extra blocks are fetched and loaded into the cache, in anticipation of the next reference.

When information is fetched from the object repository, if the cache is full, some information in the cache must be selected for replacement. The *replacement algorithm* determines which block is removed. Various replacement algorithms are possible, such as first in, first out (*FIFO*), least recently used (*LRU*), and *random*.

When an object in the cache is updated, that update can be reflected in one of several ways. The *update algorithm* determines the mechanism used. For example, a *write-back* algorithm has the cache receive the update and update the object repository only when the modified block is replaced. A *write-through* algorithm updates the object repository immediately.

2.3. Files and file systems

A *file* is an object used for long-term data storage. Its value persists longer than the processes that create and use it. Files are maintained on secondary storage devices like disks. Conceptually, a file consists of a sequence of data objects, such as integers. To provide the greatest utility, we consider each object in a file to be a single byte. Any further structure must be enforced by the programs using the file.

The *file system* is the software that manages these permanent data objects. The file system provides operations that will *create* or *delete* a file, *open* a file given its name, *read* the next object from an open file, *write* an object onto an open file, or *close* a file. If the file system allows random access to the contents of the file, it may also provide a way to *seek* to a specified location in a file. Two or more processes can *share* a file by having it open at the same time. Depending on the file manager, one or more of these processes may be allowed to write the shared file, while the rest may only read from it.

2.3.1. File system components

The file system software is composed of five different managers, each of which is used to implement some portion of the file system primitives.

The *access control manager* maintains *access lists* that define which users may access a particular file, and in what way – whether to read, write, delete, or execute.

The *directory manager* implements the naming directories used to implement the name space provided by the file system. It provides primitives to create and delete entries in the directories, as well as to search through the existing entries.

The *naming manager* implements the name space provided by the file system. The name evaluation mechanism is part of the naming manager, and uses the directory manager primitives to translate file names into object references.

The *file manager* interacts with the *disk manager* to map logical file bytes onto physical disk blocks. The disk manager manipulates the storage devices, and provides primitives to read or write a single, randomly accessed disk block. The disk manager may implement a cache of recently referenced disk blocks. Such a cache is called a *disk block cache*.

The file manager maintains the mapping of logical file bytes to physical disk blocks. The file manager treats files as if they were composed of fixed-sized *logical file blocks*. These logical

blocks can be larger or smaller than the hardware block size of the disk on which they reside. The file manager may implement a *file block cache* of recently referenced logical file blocks.

The file manager maintains several files which are private to its implementation. One contains the correspondence of logical file blocks to physical disk blocks for every file on the disk. Another is a list of the physical disk blocks that are currently part of a file, and the disk blocks that are free. These files are manipulated in response to file manager primitives which create or destroy files or extend existing ones.

In a distributed file system implementation, where the disk used for storage is attached to a server processor connected to the client only by a network, we distinguish between *disk servers* and *file servers*. In a disk server, the disk manager resides on the server processor, and all other file system components reside on the client. A disk server merely provides raw disk blocks to the client processor, and the managers must retrieve all mapping and access control information across the network.

In a file server, all five managers are implemented on the server processor. Client programs send short network messages to the server, and receive only the requested information in return messages. All access control computations, name translations, and file layout mappings are performed on the server processor, without requiring any network traffic.

2.3.2. The UNIX file system

The UNIX file system follows this model, with some implementation differences [6]. The access control lists are maintained in the same private file as the mappings between logical file blocks and physical disk blocks. Together, the entries in this file are known as *inodes*.

The UNIX file name resolution mechanism implements a hierarchical naming system. Directories appear as normal files, and are generally readable. User programs may read and interpret directly, or use system-provided primitives to treat the directories as a sequence of name objects. Special privileges are required to write a directory, to avoid corruption by an incorrectly implemented user program.

In UNIX, file names are composed of simple names separated by the delimiter character '/'. Names are evaluated, as outlined in the hierarchical name evaluation given above, with the current evaluation directory at each step being a directory in the naming hierarchy. Names are finally translated into unique, numerical object indices. The object indices are then used as file identifiers by the file manager. The namespace of the UNIX naming mechanism can also be thought of as a tree-structured graph.

To improved file system performance, the disk manager implements a disk block cache. Blocks in this cache are replaced according to a least recently used replacement policy [58].

2.3.3. Our view of file systems

For the purposes of this thesis, we are interested only in the operation of the file manager and operations concerning logical file blocks. We do not consider the implementation of the name evaluation mechanism or the mapping of logical file blocks to physical blocks. All names in the Caching Ring system are considered to be simple names, and the mapping from the long name

strings used in a hierarchical naming system to the numerical object identifiers used thereafter to refer to file objects is not part of the caching mechanism.

Descriptions of file system alternatives can be found in Calingaert [7], Haberman [25], and Peterson and Silberschatz [40]. Comer presents the complete implementation of a file system which is a simplification of the UNIX file system described above in [11].

3. Analysis of a Single-Processor System

There has been very little empirical data published on file system usage or performance. Obtaining trace data is difficult, typically requiring careful modifications to the operating system, and the resulting data is voluminous. The published studies tend to deal with older operating systems, and for this reason may not be applicable in planning future systems.

This chapter extends our understanding of caching to disk block caches in single processor systems. We recorded the file system activity of a single processor timesharing system. We analyzed this activity trace to measure the performance of the disk block cache, and performed simulation experiments to determine the effects of altering the various parameters of the processor's disk block cache. We also measured the amount of shared file access that is actually encountered. These measurements and simulation experiments allow us to characterize the demands of a typical user of the file system, and the performance of the file system for a given set of design parameters.

3.1. Introduction

Understanding the behavior of file block caching in a single processor system is fundamental to designing a distributed file block caching system and analyzing the performance of that caching system. Using an accurate model of the activity of a single user on a client workstation, we can build simulations of a collection of such workstations using a distributed file block caching system. By analyzing the file activity on a single-processor system, we can develop such a model. To this end, we designed experiments to collect enough information about an existing system to allow us to answer questions such as:

- How much network bandwidth is needed to support a workstation?
- How much sharing of files between workstations should be expected?
- How should disk block caches be organized and managed?
- How much performance enhancement does a disk block cache provide?

The experiments are an independent effort to corroborate similar data reported by McKusick [36], Lazowska *et al.* [32], and Ousterhout *et al.* [38], in a different environment, and with a different user community and workload. We compare our results and theirs in Section 3.6.

The basis of the experiments is a trace of file system activity on a time-shared uniprocessor running the 4.2BSD UNIX operating system [1]. The information collected consists of all read

and write requests, along with the time of access. The amount of information is voluminous, but allows us to perform a detailed analysis of the behavior and performance of the file and disk subsystems.

We wrote several programs to process the trace files – an analysis program that extracts data regarding cache effectiveness and file system activity, a data compression program, and a block cache simulator. Using these programs, we were able to characterize the file system activities of a single client, and the performance benefits of a disk block cache in various configurations.

3.2. Gathering the data

Our main concerns in gathering the data were the volume of the data and affecting the results by logging them through the cache system under measurement. We wished to gather data over several days to prevent temporary anomalies from biasing the data. We also wished to record all file system activity, with enough information to accurately reconstruct the activity in a later simulation. It quickly became obvious that it would not be feasible to log this data to disk – an hour of typical disk activity generates approximately 8.6 Mbytes of data.

The method settled upon used the local area network to send the trace data to another system, where it was written to magnetic tape. Logging routines inserted into the file system code placed the trace records in a memory buffer. A daemon read the records from the trace buffer and sent them to a logging process via a TCP connection. The logger simply wrote the records on tape. A day's activity fills a 2400 foot tape recorded at 6250 bpi.

In this manner, the buffers used in the disk subsystem are completely bypassed. The daemon that reads and sends trace records consumed approximately 3% of the CPU, and the impact on the performance of the disk subsystem is negligible.

3.3. The gathered data

We inserted probes to record activity in both the file manager and the disk manager. All event records are marked with the time at which they occurred, to millisecond resolution.

In the file manager, we recorded all file open, close, read, and write events. Each event record contains the name of the process and user that requested the action. Open events record the file index that uniquely identifies the file on disk, and the inode information, but not the name by which the user called it. Close event records contain the same data. Read and write event records identify the affected file, the point in the file at which the transfer began, and how many bytes were transferred.

In the disk manager, operations are performed on physical blocks. Only read and write events occur at this level. Each event record contains the address of the first block used in the transfer, how many bytes were transferred, and whether or not the block(s) were found in the disk block cache.

The recorded data is sufficient to link file manager activity to the corresponding disk manager activity. However, there is much disk manager activity that cannot be accounted for by file manager read and write requests. This is from I/O that is performed by the directory manager while resolving file names to file identifiers during file opens, and by the file manager when

transferring inodes between disk and memory. Also, paging operations do not use the disk block cache, and are not recorded in this trace. When a new program is invoked (via the *exec* system call), the first few pages of the program are read through the disk block cache, and are recorded in our trace data. The remaining pages are read on demand as a result of page faults, and this activity does not appear in our trace data. We can estimate the overhead involved in file name lookup by comparing the disk activity recorded in our traces and the simulated disk activity in our simulations.

3.3.1. Machine environment

We collected our trace data on a timeshared Digital Equipment Corporation VAX-11/780 in the Department of Computer Sciences at Purdue University. The machine is known as ‘‘Merlin’’ and is used by members of the TILDE project for program development and document editing, as well as day-to-day housekeeping. Merlin has 4 Mbytes of primary memory, 576 Mbytes of disk, and runs the 4.2BSD version of the UNIX operating system. The disk block cache is approximately 400 Kbytes in size.

Traces were collected for four days over the period of a week. We gathered data during the hours when most of our users work, and specifically excluded the period of the day when large system accounting procedures are run. Trace results are summarized in Table 3-1, where each individual trace is given an identifying letter. During the peak hours of the day, 24 – 34 files were opened per second, on average. The UNIX load average was typically 2 – 8, with under a dozen active users.

3.4. Measured results

Our trace analysis was divided into two parts. The first part contains measurement of current UNIX file system activity. We were interested in two general areas: how much file system activity is generated by processes and system overhead, and how often files are shared between processes, and whether processes that share files only read the data or update the data as well. The second part of our analysis, examining the effectiveness of various disk cache organizations, is presented in Section 3.5.

3.4.1. System activity

The first set of measurements concerns overall file system activity in terms of users, active files, and bytes transferred (see Table 3-2). The most interesting result is the throughput per active user. We consider a user to be active if he or she has any file system activity within a one-minute interval. Averaged over a one-minute interval, active users tend to transfer only a few kilobytes of data per second. If only one-second intervals are considered, users active in these intervals tend to transfer much more data per second (approximately 10 Kbytes per second per active user), but there are fewer active users.

In [32], Lazowska *et al.* reported about 4 Kbytes of I/O per active user. This is higher than our figure, because their measurement includes additional activity not present in our analysis such as directory searches and paging I/O, and was measured for a single user at a time of heavy usage. Ousterhout *et al.* reported about 1.4 Kbytes of I/O per user in [38]. This is lower than our figure, because their measurement does not include program loading I/O activity or the overhead of

Trace	A	B	C	D
Duration (hours)	7.6	6.8	5.6	8.0
Number of trace records	1,865,531	1,552,135	1,556,026	1,864,272
Size of trace file (Mbytes)	66	55	55	66
Total data transferred (Mbytes)	402	330	334	405
User data transferred (Mbytes)	126	110	120	135
Disk cache miss ratio (percent)	10.10	10.03	10.67	9.82
Blocks read ahead	9424	9143	10376	13933
open events	28,427	23,837	22,403	25,307
close events	28,194	23,772	22,227	25,162
read events	51,281	40,203	45,619	77,471
write events	23,689	18.972	18,834	26,073
shared file opens	5,015	3,919	4,240	3,628
shared read events	16,892	13,057	14,017	31,000
shared write events	717	695	827	995
inode lock events	911,151	778,208	762,563	906,140

Table 3-1: Description of activity traces

reading and writing inodes from disk. They also define a user as one who is active over a ten minute interval, and the throughput figure is averaged over that time period.

Several of the statistics seem to be due to the heavy reliance that the UNIX system places on the file system for storage of data. Executable versions of programs, directories, and system databases are all accessible by programs as ordinary files, and utility programs access them heavily. Information about users of the system is spread across several files, and must be gathered up by programs that would use it. Utilities are typically written to keep their data in several distinct files, rather than one monolithic one, or to themselves be made up of several passes, each stored in a separate file. Programmers are encouraged to split their programs into many smaller files, each of which may contain directives to the compiler to include up to a dozen or more files that contain common structure definitions. For example, to compile a trivial C program with the standard VAX compiler requires touching 11 files: three make up the C compiler; two more for assembler and loader; one for the standard library; the source file itself with a standard definitions file; two temporary files; and finally the executable version of the program. Invoking the optimizer adds two more files, the optimizer pass and an additional temporary. Such a trivial compile can easily take less than six seconds of real time on an unloaded system such as Merlin.

Trace	A	B	C	D
Average throughput (bytes/sec)	4600	4500	5900	4700
Unique users	20	18	19	17
Maximum active users (per minute)	7	6	7	5
Average active users per(minute)	2.05±1.10	2.75±1.05	2.70±1.25	2.91±1.32
Average throughput per active user (bytes/sec)	2243±782	1636±451	2185±691	1615±504
Average opens/sec per active user	2.4±1.8	2.2±1.7	2.1±1.6	2.0±1.5
Average reads/sec per active user	5.75±8.7	3.89±6.87	3.78±6.22	5.40±8.83
Average writes/sec per active user	1.9±5.46	1.02±3.56	0.93±2.48	1.42±3.71

Table 3-2: Measurements of file system activity

However, the low average throughput per active user suggests that a single 10Mbit/second network has enough bandwidth to support several hundred users using a network-based file system. Transfer rates tended to be bursty in our measurements, with rates as high as 140 Kbytes/sec recorded for some users in some intervals, but such a network could support several such bursts simultaneously without difficulty.

We performed simple analysis of access patterns to determine the percentage of files that are simply read or written straight through, with no intermediate seeks. Table 3-3 summarizes our results. The percentages are cumulative, *i.e.*, 80.2% of the file accesses in trace A had two or fewer seeks. These measurements confirm that file access is highly sequential. Ousterhout *et al.* report that more than 90% of all files are processed sequentially in their computing environment.

Trace	A	B	C	D
Linear access	17634 (62.5%)	15118 (63.6%)	13776 (62.0%)	16683 (66.3%)
One seek	4889 (79.2%)	4119 (80.9%)	3561 (78.0%)	3952 (82.0%)
Two seeks	272 (80.2%)	265 (82.0%)	253 (79.1%)	271 (83.1%)
Three or more	5632 (100%)	4270 (100%)	4637 (100%)	4256 (100%)

Table 3-3: Linear access of files

3.4.2. Level of sharing

After measuring the overall file system activity recorded in our traces, we turned our attention to how files are shared. Table 3-4 reports our measurements of file sharing between processes. Of the files accessed in the system, approximately 16.5% are opened simultaneously by more than one process. Of those, approximately 75% are directories. Thus, approximately 4% of all non-directory file opens are for shared access.

Directories are shared when two or more processes are opening files in the same section of the file system name space. The directories must be searched for each file open. In a network file system which locates the directory manager at the server, client processes will not share directories; rather, access to directories will be serialized at the server. The number of files shared between workstations would then be the much lower figure of about 4% of all opened files.

Trace	A	B	C	D
File open events	28427	23837	22403	25307
Shared open events	5015 (17.6%)	3919 (16.4%)	4240 (18.9%)	3628 (14.3%)
Unique files shared	352	168	344	212
Shared directories	293 (83.2%)	114 (67.9%)	280 (81.4%)	150 (70.8%)
Shared read events	16892	13057	14017	31000
Shared write events	717	695	827	995

Table 3-4: Sharing of files between processes

Of the files that are shared, approximately 4% of the accesses involved modifying the contents of the files. Removing directories from these statistics increases this to 8.8%, which is still a very small percentage of all file activity.

Furthermore, analysis of the traces indicates that sharing access mainly occurs in system files. Only approximately 10% of the shared files are user files. Files are shared by small groups of processors, as shown in Table 3-5.

Size of group	Frequency of occurrence	Cumulative frequency
2	62.8%	62.8%
3	17.4%	80.2%
4	9.4%	89.6%
5	5.4%	95.0%
6	0.9%	95.9%
7	1.4%	97.3%
8	1.6%	99.0%
9	1.1%	100.0%

Table 3-5: Size of processor groups sharing files

It is commonly argued that caching is not used in distributed file systems because updating the caches on writes is very expensive, both in terms of the data structures and communications delay required to maintain coherence. From these data, we conclude that any coherence mechanism for shared writes will seldom be invoked, and thus should have minimal impact on the overall performance of the system. We also conclude that a caching mechanism for any file system should be optimized to give the most performance benefit to the reading of file blocks.

3.5. Simulation results

In a network file system, one of the most interesting areas for study is the disk block cache. Disk and network access speeds are limited to those available from hardware vendors. A cache implemented in software, on the other hand, is extremely flexible. To optimize file system performance, the designer can vary the percentage of available memory used for caching blocks, and the algorithms to allocate and replace those blocks. With an appropriate set of algorithms, performance can be increased simply by adding to the amount of available memory, even if the algorithms are fixed in hardware.

The UNIX file system uses approximately 10% of main memory (typically 200 – 800 kbytes) for a cache of recently used blocks. The blocks are maintained in least recently used fashion and result in a substantial reduction in the number of disk operations (see Table 3-1).

For a network file system with much higher transfer latency, the role of the cache is more important than in a system with locally attached disk storage. A well-organized cache in the client workstation can hide many or all of the effects of a remote file system. With current memory technology, it is reasonable to conceive of a cache of 2 – 8 Mbytes in the client, and perhaps 32 – 64 Mbytes in a few years. Even though the general benefits of disk block caches are already well known, we still wished to answer several questions:

- How do the benefits scale with the size of the cache?
- How should the cache be organized to maximize its effectiveness?
- How effective can a cache in the client be in overcoming the performance effects of a remote disk?

3.5.1. The cache simulator

To answer these questions, we wrote a program to simulate the behavior of various types of caches, using the trace data to drive the simulation. We used only the data collected from the file manager level. This allowed us to simulate the effect of a cache on the I/O generated by user processes, without including the effects of I/O generated for maintenance of the file system and the naming system. For the measurements below, the four traces produced nearly indistinguishable results; we report only the results from trace A.

The simulated system mimics the data structures found in the UNIX file system. We simulated only the file manager operations, and included a file block cache instead of a disk block cache. This cache consisted of several fixed-sized blocks used to hold recently referenced portions of file. We used an LRU algorithm for block replacement in the cache. There is a table of currently open and recently closed files, where each entry in the table includes the file identifier, reference count, file size, statistics about how the file was accessed, and a pointer to a doubly-linked list of blocks from the file that reside in the cache.

In the UNIX system, when a file is closed, any of its blocks that may reside in the cache are not automatically flushed out. This results in a significant performance improvement, since many files are opened again shortly after they are closed, and their blocks may still be found in the cache.[1]We wished to preserve this aspect of the UNIX disk block cache in our simulated file block cache.

As the trace is processed, an open record causes an entry in the file table to be allocated. If the file is already in the file table, the associated reference count is incremented. A close record causes the reference count to be decremented. When the reference count reaches zero, the file table entry is placed on a free list. Any blocks in the file that still reside in the cache remain associated with the file table entry. So, in fact, when the simulator must allocate a file table entry to respond to an open record, it searches the free list first. If an entry for the file is found, it is reclaimed, and any file blocks that still remain in the cache are also reclaimed.

For each read or write record, the range of affected bytes is converted to a logical block number or numbers. The simulator checks to see if the affected blocks are in the cache. If so, the access is satisfied without any disk manager activity, and the block is brought to the head of the linked list that indicates the LRU order of the cache blocks. If not, a block from the cache free list is allocated. If the free list is empty, the block at the tail of the LRU list of the file block cache is freed and allocated to this file. If the cache is simulating a write-back cache, any changes to the block are written back at this time.

The principal metric for evaluating cache organization was the *I/O ratio*, which is similar to the miss ratio. The I/O ratio is a direct indicator of the percentage of I/O avoided due to the cache. It expresses the ratio of the number of block I/O operations performed to the number of block I/O operations requested. An I/O operation was charged each time a block was accessed and not in the cache, or when a modified block was written from the cache back to disk. The I/O ratio is different from the miss ratio in that it effectively counts as missed those I/O operations resulting from the write policy, even though those blocks appear in the cache.

A secondary metric was the effective access time. We assigned a time cost to each disk access and computed the total delay that user programs would see when making accesses through the cache. This allowed us to evaluate the effects of varying the access time to the disk storage on performance.

Often in the traces, programs made requests in units much smaller than the block size. We counted each of these requests as a separate access, usually satisfied from the cache. Because it more closely simulates the actual performance that programs will see, we chose not to collapse redundant requests from programs even though this results in lower miss and I/O ratios and effective access times.

The results are reported only after the simulator reaches steady state. That is, block accesses and misses that occur before the cache has filled are ignored. Modified blocks left in the cache at the end of the simulation are not forced out, because this would unrealistically increase the miss and I/O ratios.

---

[1]See Section 3.5.2 for more discussion of this aspect of the cache.

3.5.2. Cache size, write policy, and close policy

By varying parameters of the simulations, we investigated the effect on performance of several cache parameters: cache size, write policy, close policy, block size, and read ahead policy. Figure 3-1 and Table 3-6 show the effect of varying the cache size and write policy with a block size of 4096 bytes (the most common size in 4.2BSD UNIX systems). We simulated both the write-through and write-back cache policies.

Write-back results in much better performance for large caches. Unfortunately, it can leave many modified blocks in the cache for long periods of time. For example, with a 4Mbyte cache, about 20% of all blocks stay in the cache for longer than 20 minutes. If the workstation crashes, many updates may have never made it back to the server, resulting in the loss of large amounts of information.

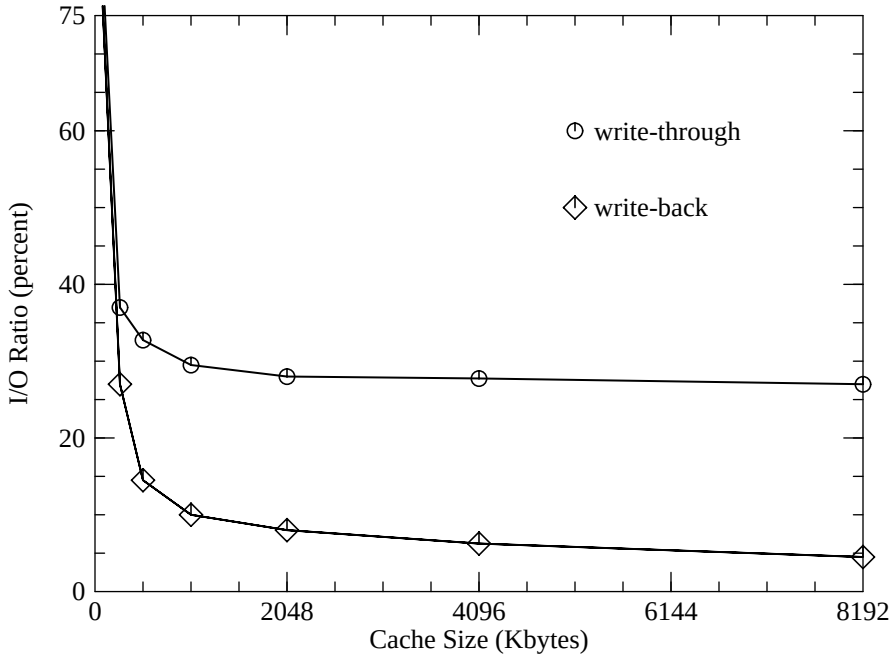


Figure 3-1: Cache size *vs.* write policy for trace A

Cache Size	Write-through	Write-back
256 Kbytes	37.1%	26.9%
512 Kbytes	32.7%	14.4%
1 Mbyte	29.5%	10.1%
2 Mbytes	28.0%	8.1%
4 Mbytes	27.7%	6.2%
8 Mbytes	27.1%	4.5%

Table 3-6: Cache size *vs.* write policy for trace A

UNIX systems generally run a utility program that flushes modified blocks from the cache every 30 seconds. This results in higher I/O ratios (though not as high as those exhibited with a write-through policy), but the amount of information lost owing to a crash is greatly reduced. Ousterhout *et al.* reported that a 30-second flush interval reduces the I/O ratio to approximately 25% below write-through, and a 5 minute flush interval results in a I/O ratio 50% below that of write-through.

We also investigated the effect of flushing all blocks associated with a file when the file is closed. Analysis of our traces indicated that many files are opened and closed repeatedly. This is most noticeable in a trace that involves many program compiles. The files containing data structure definitions are opened and closed repeatedly as they are read into each individual program file. Figure 3-2 and Table 3-7 show the effect on the overall I/O ratio of maintaining and flushing file blocks after a close for a range of cache sizes in a write-back cache.

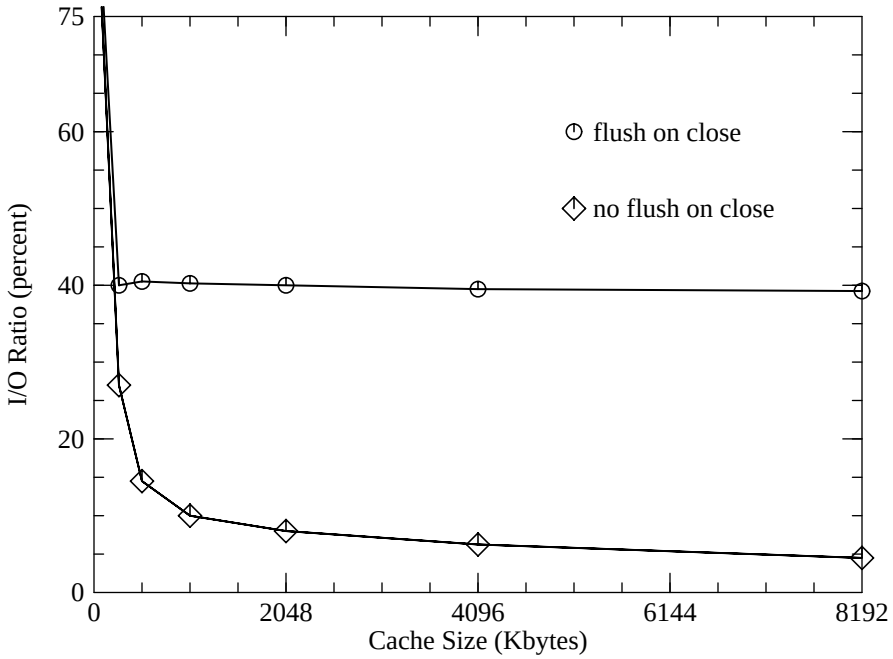


Figure 3-2: Effect of close policy on I/O ratio for trace A

Cache Size	Write-through	Write-back
256 Kbytes	40.1%	26.9%
512 Kbytes	40.5%	14.4%
1 Mbyte	40.3%	10.1%
2 Mbytes	40.1%	8.1%
4 Mbytes	39.6%	6.2%
8 Mbytes	39.3%	4.5%

Table 3-7: Effect of close policy on I/O ratio for trace A

This shows the fundamental reason that the UNIX disk block cache works so well. File access patterns are such that many files are reused before they would ordinarily leave the cache. Floyd has found that most files in a UNIX environment are re-opened within 60 seconds of being closed [21]. Maintaining blocks of closed files in the cache has a significant performance advantage over having to re-fetch those blocks a short time after the close.

3.5.3. Block size

We also evaluated the effects of differing block sizes. The original UNIX file system used 512 byte blocks. The block size has since been expanded to 1024 bytes in AT&T System V [20] and 4096 bytes in most 4.2BSD systems. Figure 3-3 and Table 3-8 show the results of varying the block size and cache size.

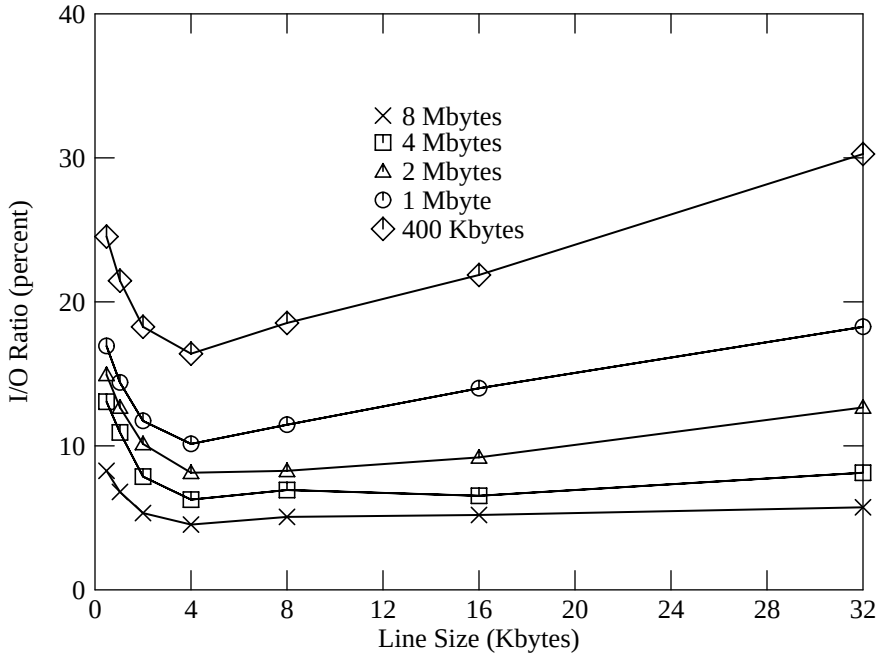


Figure 3-3: I/O ratio *vs.* block size and cache size for trace A

	Cache Size				
Block size	400 Kbyte	1 Mbyte	2 Mbyte	4 Mbyte	8 Mbyte
512 bytes	24.5%	17.0%	14.9%	13.1%	8.2%
1024 bytes	21.5%	14.4%	12.6%	10.9%	6.9%
2048 bytes	18.2%	11.7%	10.1%	7.8%	5.3%
4096 bytes	16.4%	10.1%	8.1%	6.2%	4.5%
8192 bytes	18.4%	11.4%	8.2%	6.9%	5.0%
16384 bytes	21.8%	14.0%	9.2%	6.6%	5.2%
32768 bytes	30.2%	18.3%	12.7%	8.1%	5.7%

Table 3-8: I/O ratio *vs.* block size and cache size for trace A

In general, large block sizes work well. They work well in small caches, and even better in large ones. For our traces, the optimal block size, independent of cache size, is 4096 bytes. This is an artifact of the system I/O library that rounds file system requests up to 1024 and 4096 bytes, although there are still programs that make smaller requests. For very large block sizes, the curves turn up because the cache has too few blocks to function effectively. Especially in smaller caches, large block sizes are less effective because they result in fewer memory blocks available to cache file blocks. Most of the memory space is wasted because short files only occupy the first part of their blocks.

Although large blocks are attractive for a cache, they can result in wasted space on disk due to internal fragmentation. 4.2BSD uses a mixed block size technique to minimize wasted space in short files. A cache with a fixed block size still works well with a mixed block size file system, though there may be wasted space within the cache blocks, as described above.

3.5.4. Readahead policy

The UNIX file system includes a heuristic to perform selective readahead of disk blocks. For each open file, the file manager keeps track of the last block that was read by a user program. If, when reading block *b*, the last block that was read is block *b*−1, the system fetches both block *b* and *b*+1 into the cache, if they are not already in the cache (and block *b*+1 exists in the file). This algorithm describes a readahead level of 1. We simulated with readahead levels of 0, 1, 2, and 3; *i.e.*, reading between 0 and 3 extra blocks in response to sequential access. Our results are summarized in Figure 3-4 and Table 3-9.

A readahead of one block makes a small difference; additional readahead makes no apparent difference. Large amounts of readahead, *i.e.*, several blocks with a large block size, degrade performance in a similar fashion to extremely large block sizes.

The readahead makes little difference because only a small percentage of the file references result in blocks being read ahead. A sequence of small file accesses within the same logical file block does not reference any new file blocks, thus no blocks are read ahead. This is consistent with our trace data (see Table 3-1). A process reading a file sequentially in amounts smaller than the block size will repeatedly access each of the blocks *b*−1, *b*, and *b*+1. At the transition from accessing block *b*−1 to block *b*, block *b*+1 will also be fetched in accordance with the readahead policy. However, the process will now continue to access *b* several times before reaching block *b*+1, so the effect of the extra fetch is minimal. Most UNIX files are small enough to fit in one block, so that in many cases there is no extra block to be read ahead.

The payoff of the readahead policy is based on the assumption that the time spent in reading the extra block is not noticeable to the process requesting the original disk I/O. This is likely to be true in an environment with a locally attached disk. With a remote disk, access time is approximately four to five times as great, and this assumption may not hold true.

3.5.5. Comparisons to measured data

Merlin runs with a disk block cache of about 100 − 200 blocks of different sizes, with a total cache size of approximately 400 Kbytes. The cache is flushed of modified blocks every 30 seconds. According to our simulations, this should yield a I/O ratio of approximately 20%. The

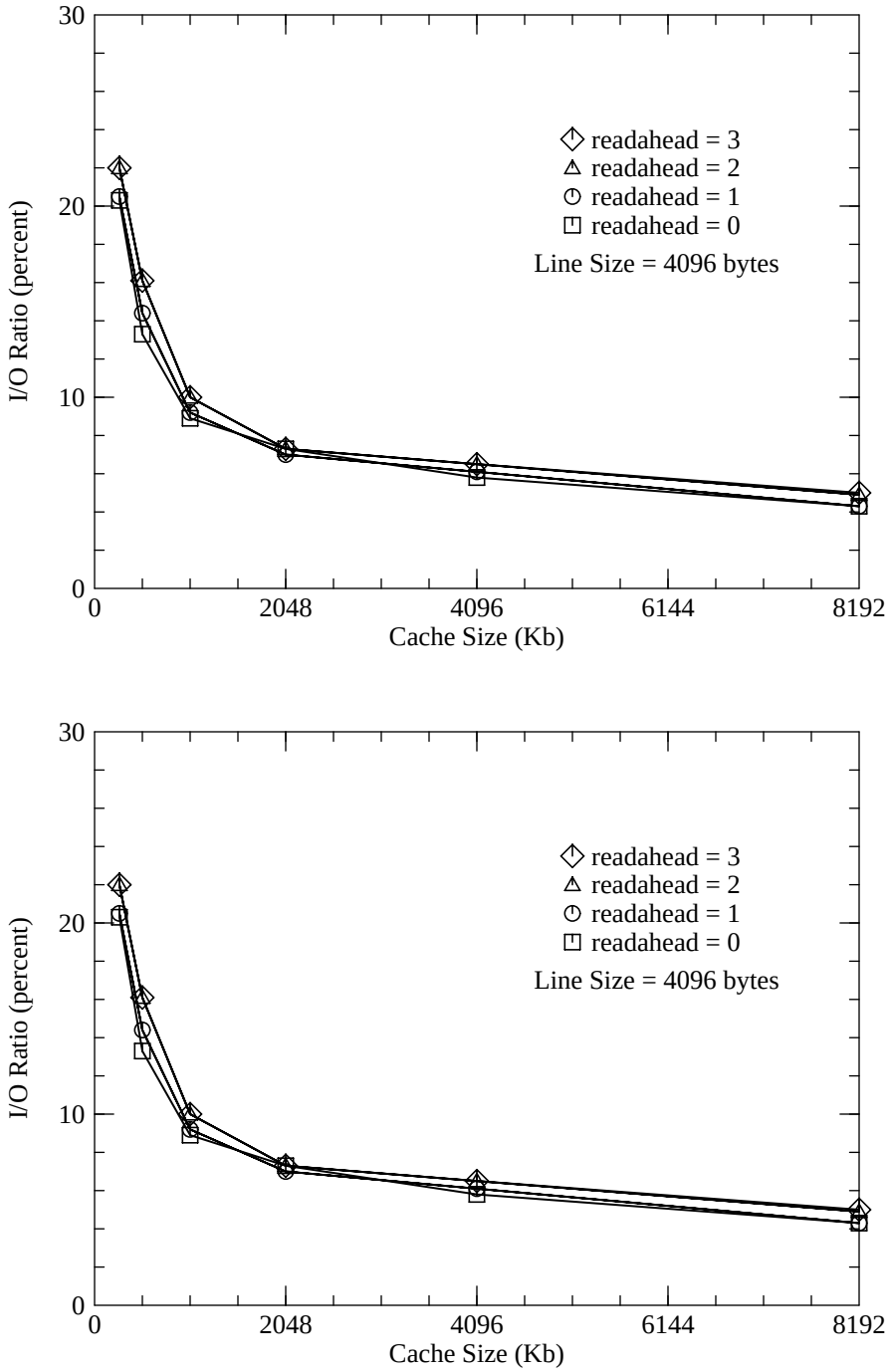


Figure 3-4: I/O ratio *vs.* cache size and readahead policy for trace A

actual I/O ratio, calculated from the trace data, is approximately 10%. Leffler *et al.* report a measured cache I/O ratio of 15% [36]. The discrepancy results from differences in the actual activity that is measured. The simulation results do not include activity for paging, searching directories during name lookups, or inode fetches. Directory lookups and inode fetches are reported to have low I/O ratios, and account for a significant amount of disk activity. The recorded trace does not include activity for paging.

512 byte blocks				
Cache Size	Blocks read ahead			
	0	1	2	3
256 Kbytes	27.0%	25.0%	24.5%	24.4%
512 Kbytes	18.9%	17.6%	17.2%	17.1%
1 Mbyte	14.6%	13.7%	13.4%	13.3%
2 Mbytes	13.1%	12.4%	12.1%	12.0%
4 Mbytes	11.8%	11.4%	11.2%	11.1%
8 Mbytes	7.4%	7.2%	7.0%	6.9%

4096 byte blocks				
Cache Size	Blocks read ahead			
	0	1	2	3
256 Kbytes	20.3%	20.5%	22.0%	22.0%
512 Kbytes	13.3%	14.4%	16.1%	16.1%
1 Mbyte	8.9%	9.2%	10.0%	10.0%
2 Mbytes	7.3%	7.0%	7.3%	7.3%
4 Mbytes	5.8%	6.1%	6.5%	6.5%
8 Mbytes	4.3%	4.3%	4.9%	5.0%

Table 3-9: I/O ratio *vs.* cache size and readahead policy for trace A

3.5.6. Network latency

The stated intention of a cache is to decrease the effective access time to a set of objects. To judge how well a large cache can improve the effective access time of cross-network disk accesses, we ran another set of simulations that varied the delay charged for each disk access from 30 ms (average time for a 4 Kbyte block from a fast disk) to 120 ms (average time for a 4 Kbyte block across a 10 Mbit/sec network). The results are shown in Figure 3-5.

The surface shows that a sufficiently large cache allows a remote disk to perform as effectively as a local disk with a smaller cache. Figure 3-6 shows the data in a different format.

From this graph, we see that a range of effective access times can be achieved at all four transfer rates. For example, an effective access time of approximate 7 ms/block can be achieved with a 700 Kbyte cache at a transfer rate of 30 ms/block, and with a 7 Mbyte cache at a transfer rate of 120 ms/block. A cache of 7 Mbytes is feasible with today's memory technology, and may become commonplace in the next few years. Performance at the level of a 400 Kbyte cache at 30 ms is available at 120 ms with only a 2 Mbyte cache, which is easily within the reach of today's technology.

3.6. Comparisons to previous work

By recording the traffic demand on the UNIX file system, we have determined that the average user, while active, uses approximately 2 Kbytes/sec of data from the file system. This amount is exclusive of any overhead involved in managing the directory or file system structure, or page replacement for memory management.

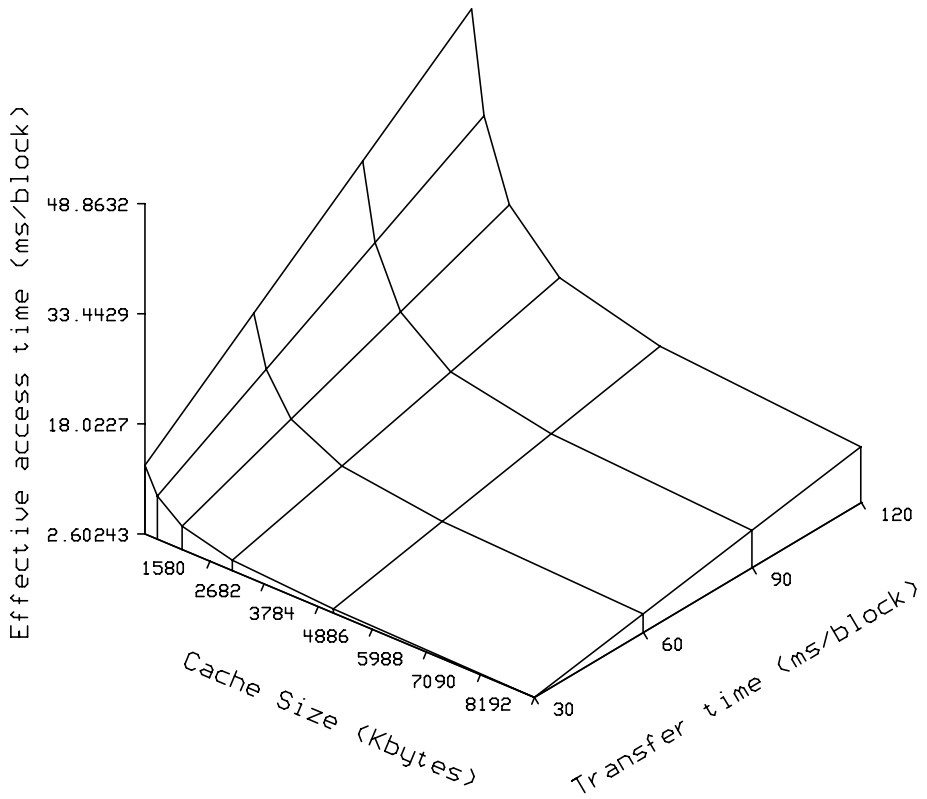


Figure 3-5: Effect of cache size and transfer time on effective access time

In our measurements of the file system, we discovered that as much as 50% of the disk activity is related to file system management: scanning directories to map file names to inodes, and locking, unlocking, reading, and writing those inodes.

Measurements of shared file access revealed that our users seldom share files. Of those files that are shared, most are shared for read-only data. Writes to shared files occur infrequently.

These results corroborate the measurements of Lazowska *et al.* [32] and Ousterhout *et al.* [38]. Lazowska *et al.* recorded about 4 Kbytes/sec of data demand per user, but their measurements include directory, file system, and paging overhead, confirming our measurement that more than 50% of disk activity is due to these operations. Ousterhout *et al.* report per-user demands similar to ours.

Our results concerning the level of sharing agree with those reported by Floyd [21]. He found that while there is extensive sharing of some few files, this sharing is restricted to standard system files. He saw very little sharing of user files.

Our simulations of various file block cache organizations indicate that a write-back cache with a block size of 4096 bytes is optimal for our environment. The simulations also indicate that a moderately sized block cache reduces disk traffic by as much as 85%. Increasing the amount of memory in the caches continues to increase the performance benefits.

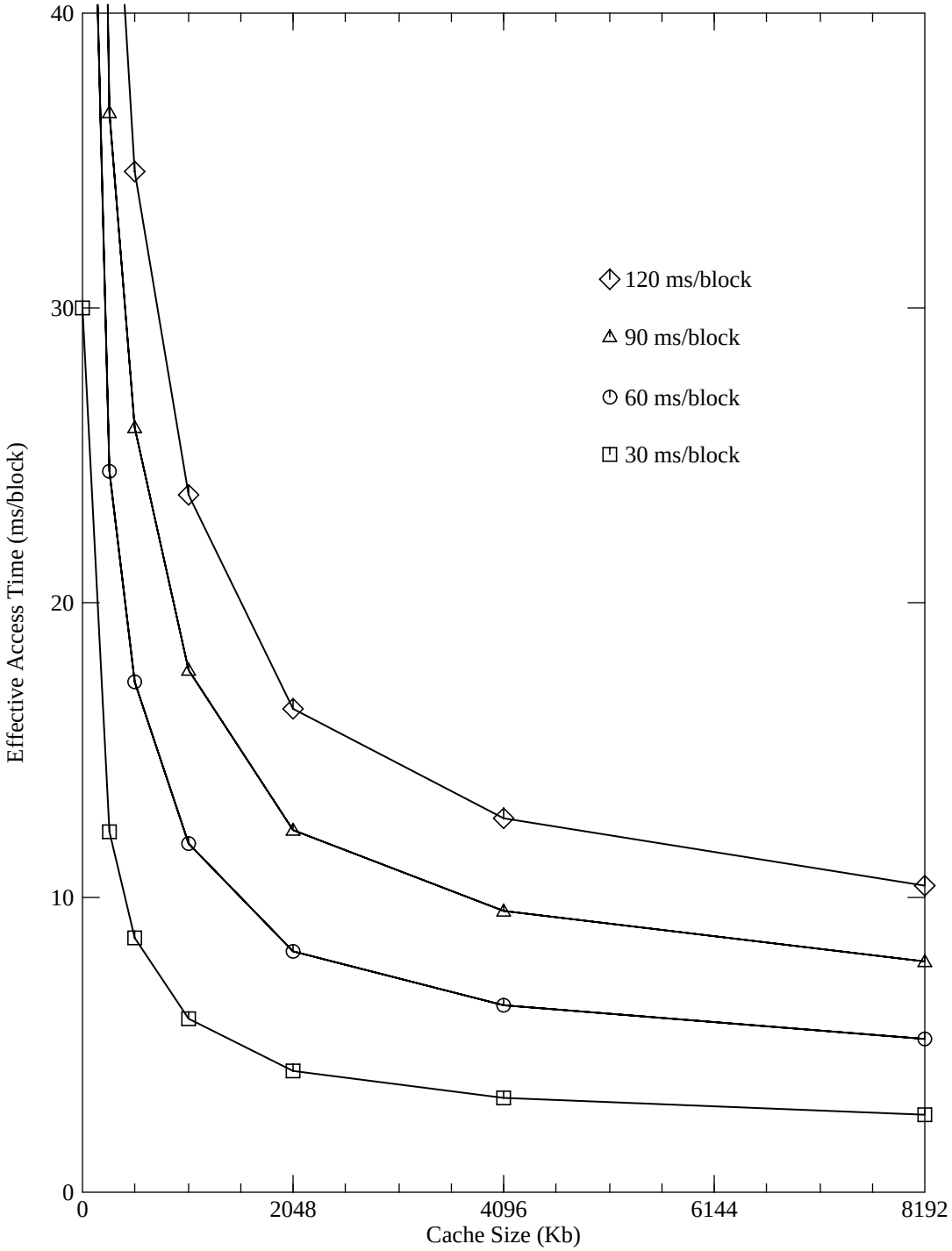


Figure 3-6: Effective access time *vs.* cache size and transfer time

These results are similar to the previous studies of the UNIX file system reported by Lazowska *et al.* and Ousterhout *et al.* Lazowska *et al.* measured a program development environment similar to ours, while Ousterhout *et al.* measured both a program development environment and a computer aided design environment.

The results also corroborate those of Smith's disk cache study, reported in [51]. His study used IBM mainframes running variants of IBM's OS operating system, and was based on physical disk blocks rather than logical file accesses. The three measured systems performed banking transactions, time sharing, and batch production work for administrative, scientific, development, and engineering support applications workloads.

3.7. Conclusions

These results from a single processor system allow us to draw several important conclusions concerning the design of a distributed file system. On the average, users demand fairly low data rates from the file system. Thus, the bandwidth available in a conventional 10 Mbit/second local area network should be sufficient to support several hundred active users, including the bursty high traffic levels sometimes experienced.

Since much of the file system activity is associated with management of the on-disk structures of the file system, a distributed file system which provides high-level file system access by clients will greatly reduce the amount of network traffic. If the server is solely responsible for management and access of these structures, network traffic can be cut by as much as 50%, compared to a distributed file system in which each client reads and writes directories, and reads, writes, locks, and unlocks inodes across the network.

A file block cache can eliminate as much as 85% of the remaining network traffic. Periodic flushing of modified blocks in the cache will limit the amount of data lost in the event of a crash, and will not severely degrade the performance benefits of the cache.

There are two ways to further increase the file system performance of a client workstation — adding local disk storage or greatly increasing the size of the cache memory. The current economy of memory costs *vs.* disk costs indicate that adding more memory is the less expensive way to increase performance.

Finally, since there is very little sharing of file data between clients, the mechanisms involved in maintaining cache coherence should be designed to perform most efficiently for non-shared data. The handling of shared data, especially writing of shared data, can be expensive without causing a significant performance penalty.

4. The Caching Ring

Based on the experiments presented in Chapter 3, we concluded that caching is essential for adequate performance in a network file system. In this chapter, we present a combination of hardware and software – the Caching Ring – that provides a generalized solution to the cache coherence problem for distributed systems.

4.1. Underlying concepts of the Caching Ring

The Caching Ring is based on several fundamental concepts:

The first is an efficient mechanism for reliable multicast. This mechanism provides inexpensive and accurate communication between those stations on the network that share an object, but does not impose any burden on stations that are uninterested in transactions about an object. Since the group of processors concerned about an object is small,[2]this eliminates a large amount of unnecessary input processing at each network station.

The second concept is that caching is solely based on the names of objects. The cache hardware and protocols need not have any specific knowledge about the structure of the objects being cached. The hardware and protocols merely assume that the objects are made up of many equal size blocks.

Third, we rely on hardware support for an efficient implementation of reliable multicast and caching by name. The *Caching Ring Interface* (*CRI*) at each station on the ring manages the communications hardware, the cache coherence algorithm, and all cache functions. Figure 4-1 shows a schematic diagram of the architecture of the CRI.

The cache directory and the memory used for cached objects are dual-ported between the CRI and the CPU, *i.e.*, the CPU can locate and access cached objects without intervention by the CRI. The CPU may not, however, modify either of these data areas. The cache controller maintains two private data stores, the status bits associated with each cache entry, and a cache of group maps. Groups are discussed further in Section 4.2.2. The ring interface provides reliable multicast communication with the other stations on the Caching Ring.

The CRI provides four primitives for the client CPU: *open*, *close*, *read*, and *write*. These are used to access objects that do not currently reside in the cache, and modify entries that already

[2]See Section 3.4.2.

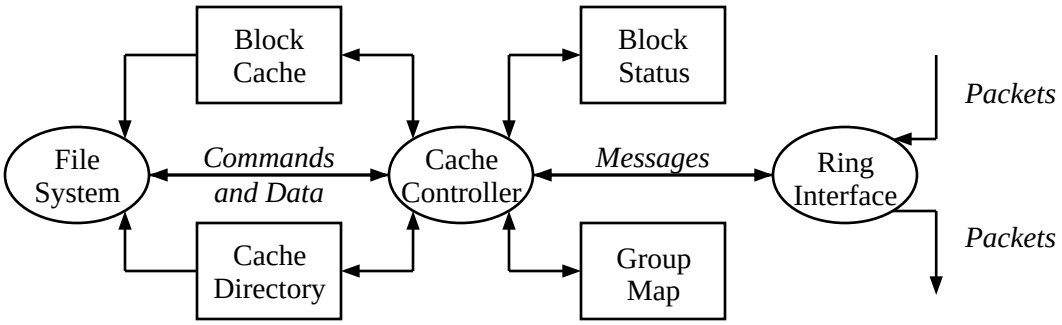


Figure 4-1: Block diagram of the Caching Ring Interface

exist. Direct modification of cached entries is not allowed, because the coherence algorithm must first be exercised.

4.2. Organization and operation of the Caching Ring

Stations on the Caching Ring fall into one of two categories: client or server. Clients, denoted $C_1 \ldots C_N$, rely on a single central server, *S*, for storage of all objects managed by the Caching Ring. Each client has a local cache of recently used object blocks, which is used, when possible, to satisfy requests before transferring objects across the network. A client need not cache all the blocks of an object that it accesses.[3]The cache coherence protocol is designed to allow the clients to share only those blocks that are needed.

The server is the repository for all objects in the system. The same object name space is shared by all clients in the system. Objects can be shared by any number of clients. The server holds all access control information locally, and performs access control operations for the clients.

4.2.1. The interconnection network

The lowest level of the Caching Ring is the interconnection network. The interconnection network is a ring, using a token-passing access control strategy [19, 18, 45]. The ring provides a synchronous communication environment in which stations on the network clearly know when it is their turn to transmit (and thus know that all other stations are ready to receive). It also provides reliable communications, because a transmitted message is guaranteed to pass each station before it returns to the originator. Each station that receives the message acknowledges receipt by marking the packet.

The ring provides unicast, multicast, and broadcast addressing. Addressing is based on an *N*-bit field of recipients in the header of the packets. Each station is statically assigned a particular bit in the bit field; if that bit is set, the station accepts the packet and acts on it. Positive acknowledgement of reception is provided to the sender by each recipient resetting its address bit before forwarding the packet. Without loss of generality, we always assign the server bit 0.

[3]In contrast to the ITC and CFS file systems.

The use of a token-passing ring provides the Caching Ring with an efficient mutual exclusion and serialization mechanism. Internal data structures are modified only while holding the ring token. A station is thus guaranteed to have seen all messages previously transmitted which can have an affect on the internal data structures. At any time, it is guaranteed that only one station will be modifying its data structures. This is necessary for correct operation of the coherence algorithm; Section 4.2.3 gives a further example. External requests have highest priorities, followed by cache responses, and finally processor requests.

Traffic on the ring is divided into three categories: cache control, cache data, and command. Cache control and cache data packets implement the cache operations and cache coherence algorithm. Command packets are available for higher level protocols (such as a distributed file system) to implement necessary functions.

4.2.2. Groups

We define a *group* to be the set of stations interested in transactions about a particular object being cached. We denote the members of the group associated with object O as $C_G(O)$. Each group is identified to the stations on the ring by a *group identifier* or *groupID* which is the bit vector that forms the network multicast address that includes the stations in the groups. Several groups may be identified by the same *groupID*. Groups are maintained strictly between the CRIs in the system.

Each member of a group knows all members of the group. The CRI contains a cache of object name to *groupID* mappings. When a message about an object is to be sent, it is multicast only to the other members of the group, by using the *groupID* as the address field of the packet containing the message. These multicasts relieve uninvolved stations of the overhead of discarding the excess messages.

We explicitly chose not to centralize the information about group members at the server, but rather to distributed it among all interested clients. With a server-centralized mechanism, the server must act as a relay for every message from a client to the appropriate group, demanding much more of the bandwidth available at the server, and increasing the overall network load.

4.2.3. The coherence algorithm

The object caches in the clients are designed to minimize server demand, since this is the limiting system resource in a distributed file system [32]. To further reduce server demand, the caches implement a write-back server update policy instead of a write-through approach.

The object caches are managed using a block ownership protocol similar to that of the snoopy cache, discussed in Section 1.2.2.3. The CRIs maintain cache coherence on the basis of information stored locally at each cache. Each cache listens to transactions on the network and takes action, if necessary, to maintain the coherence of those blocks of which it has copies.

Several stations in a group may hold cached copies of the same block, but only one cache, denoted C_O, is the owner of the block. The owner responds to read requests for a block, and is the only station that may update the block.

Initially, the server is the owner of all blocks. As clients need to update blocks, ownership is transferred away from the server. The server is not guaranteed to have an up-to-date copy of blocks that it does not own, as explained below.

A cached object block can be *public* or *private*. Public blocks are potentially shared between several client caches. Private blocks are guaranteed to exist in only a single cache.

Finally, a cached block can be modified with respect to the copy at the server. Before modifying a block, the client must become the owner of the block and make its copy private. The owner then modifies the block. The owner responds to subsequent read requests with the latest data. Responding to such a request changes the block's status from private to public. Before making further modifications to the block, the owner must once again convert the block from public to private.

Based on the concepts of ownership, public *vs.* private, and modified, we can describe the possible states of a cache entry. Table 4-1 shows the possible states.

State Description	Private	Modified	Owner	State Name
Clean, unowned, public	false	false	false	Shared-valid
Clean, owned, public	false	false	true	Valid
	false	true	false	
Modified, owned, public	false	true	true	Shared-dirty
	true	false	false	
Clean, owned, private	true	false	true	Transition
	true	true	false	
Modified, owned, private	true	true	true	Dirty

States without a name are not possible in the system.
The **Valid** state is possible only at the server.

Table 4-1: Possible states of cache entries

In addition to the six possible states above, there is a seventh: **Invalid**. This state describes a cache entry that is not in use.

Modified blocks are not written-back to the server when they become shared. A block in either the **Dirty** or **Shared-dirty** state must be written-back to the server if it is selected for replacement. It is also written-back to the server if ownership is transferred to another station.[4]A block in state **Dirty** can be in only one cache. A block can be in state **Shared-dirty** in only one cache, but might at the same time be present in state **Shared-valid** in other caches.

To outline the Caching Ring cache coherence protocol, we consider the essential actions in referencing block *b* from object *O* in the following four cases: read hit, read miss, write hit, and

[4]See Section 4.2.4.4.

write miss. We use C_R to denote the referencing cache, and $C_O(b)$ and $C_R(b)$ to denote the state of the copy of block *b* at the owner and referencing caches, respectively.

The coherence protocol works as follows (see also Figures 4-2 and 4-3):

- **Read hit:** If the object block is in the cache, it is guaranteed to be a valid copy of the data. The processor can access the data through the shared cache directory and shared cache memory with no action necessary by the CRI or the protocol.
- **Read miss:** C_O responds to C_R with a copy of *b*. If $C_O \neq S$, C_O has most recently modified *b*, and $C_O(b)$ is either **Dirty** or **Shared-dirty**. C_O sets $C_O(b)$ to **Shared-dirty**. If $C_O = S$, $C_O(b)$ remains unchanged. C_R sets $C_R(b)$ to **Shared-valid**.
- **Write hit:** If $C_R(b)$ is **Dirty**, the write proceeds with no delay. If $C_R(b)$ is **Shared-valid** or **Shared-dirty**, a message is sent to $C_G(O)$. This message instructs all members of $C_G(O)$ to change the state of their copy of *b* to **Invalid**. After this message has circulated around the ring, $C_R(b)$ is set to **Transition**. The write is immediately completed, and $C_R(b)$ is changed to **Dirty**.
- **Write miss:** Like a read miss, the object block comes directly from C_O. If $C_O(b)$ is **Shared-dirty** or **Dirty**, a copy of the block is also written-back to the server. All other caches in $C_G(O)$ with copies of *b* change the state of their copy to **Invalid** and $C_R(b)$ is set to **Dirty**.

4.2.3.1. Client state transitions

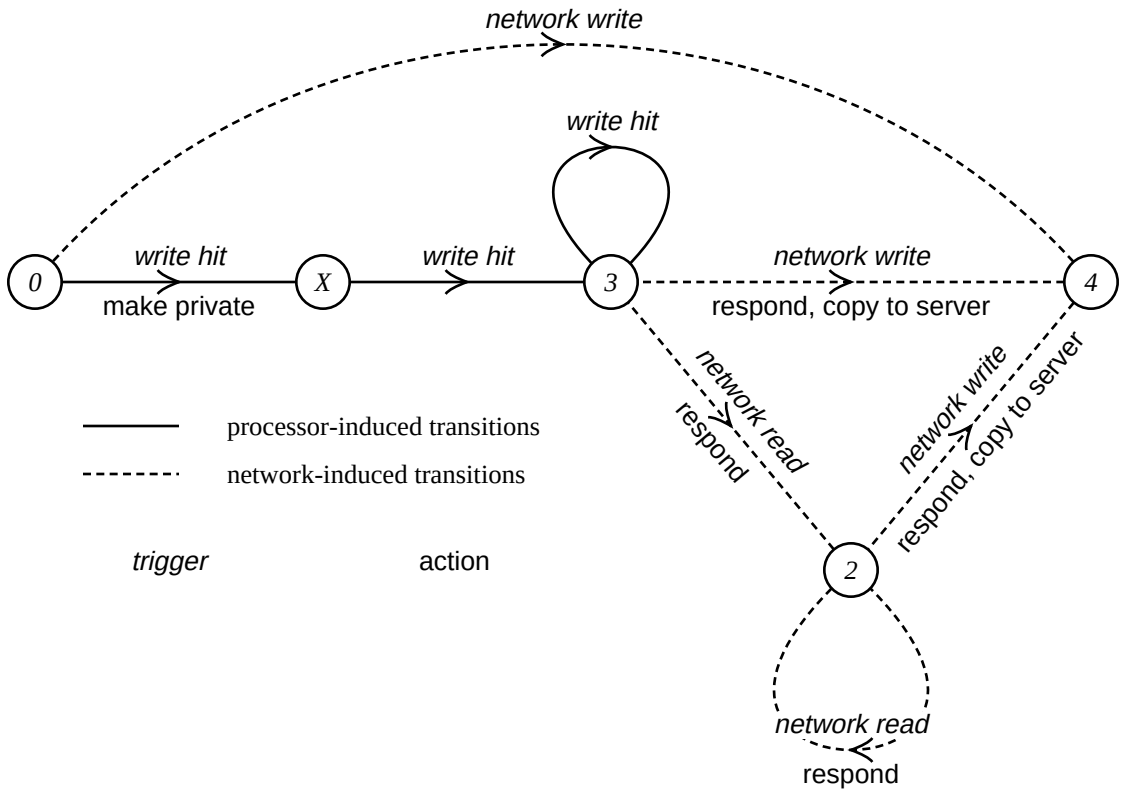
The cache in a client station implements all the states except **Valid**. A client cache entry will never be in the **Valid** state because a client only becomes the owner of a block when the block is being updated. Thus, the block will be left in the **Shared-dirty** or **Dirty** state. If the client does not own the block, the block is in the **Shared-valid** state. Figure 4-2 shows the possible transitions for entries in a client cache. Client cache entries are never in the **Valid** state because client caches only request ownership when modifying a block. A modified cache entry remains in the **Shared-dirty** or **Dirty** states until it is selected for replacement. At that time, ownership and the modified contents of the block are returned to the server. When closing an object, the cache returns to the server ownership of any associated cached blocks. To retain the performance improvements of retaining blocks after close discussed in Section 3.5.2, these blocks are retained in the cache in state **Shared-valid**.

When a cache entry that is in either the **Shared-dirty** or **Dirty** states is replaced in a client cache, the modified data are written back to the server, and ownership of the block is transferred to the server. When a cache entry that is in the **Shared-valid** state is replaced, no special action need be taken.

4.2.3.2. Server state transitions

The cache in the server implements all the states, including **Valid**. Figure 4-3 shows the possible transitions for entries in a server cache.

Because the server is the repository for all shared objects in the system, the server must also respond to requests for object blocks that do not appear in any cache. In the cache coherence



State 0: **Shared-valid**
State 2: **Shared-dirty**
State X: **Transition**
State 3: **Dirty**
State 4: **Invalid**

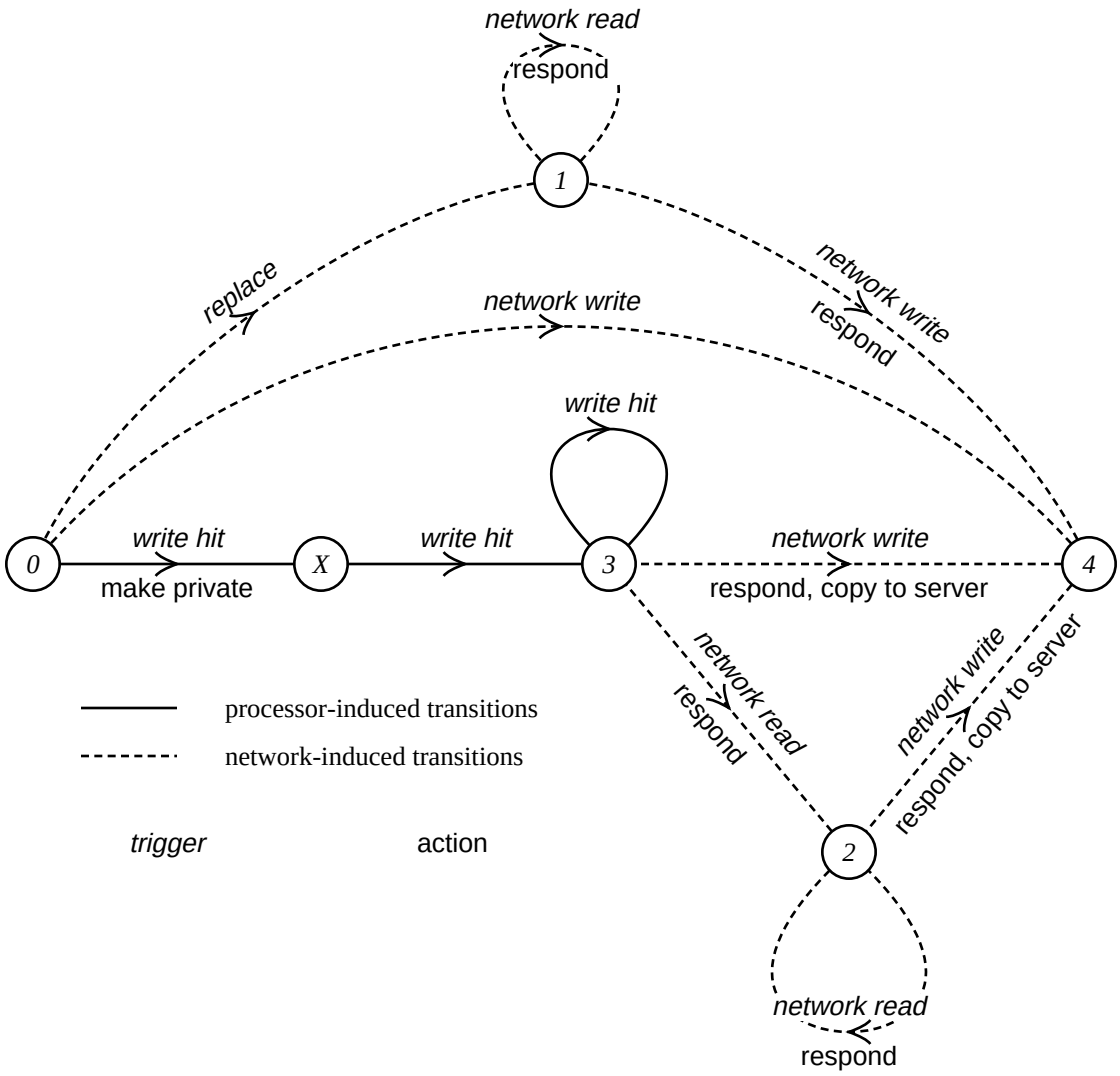
Figure 4-2: States of cache entries in a client

protocol, these object blocks can be viewed as residing in the server cache, in state **Valid**. The server responds to requests for these blocks by moving a copy from the object store into its cache, and then sending a copy to the requesting cache.

When the server cache must replace a cache entry that is in either the **Shared-dirty** or **Dirty** states, the modified copy is written-back to the object store. When replacing a cache entry that is in either the **Shared-valid** or **Valid**, no special action is required. In either case, ownership of the block remains at the server.

4.2.4. Cache-to-cache network messages

The block ownership protocol is implemented with eight cache control messages that are encapsulated in network packets and transmitted on the ring. The messages are designed so that responses are fast to compute; packet formats provide space for the response to be inserted in the original packet. The network interface contains enough delay that the packet is held at each station while the response is computed and inserted. Responses that involve longer computation are sent in a separate packet, as noted below. The original packet is marked by the station that will respond stating that a response is forthcoming.



State 0: **Shared-valid**
State 1: **Valid**
State 2: **Shared-dirty**
State X: **Transition**
State 3: **Dirty**
State 4: **Invalid**

Figure 4-3: States of cache entries in a server

Messages are divided into three categories: group and access control, cache control, and data responses. Each category is described below.

4.2.4.1. Group and access control messages

To access an object, a client must satisfy the server that it has access rights to the object, and join the pre-existing group concerned with that object. The *open* message accomplishes this.

open(*objectName, userName, accessMode*)
The client sends an *open* message to the server to indicate that it wishes to access object *objectName* in a manner described by *accessMode*. The server determines if the supplied *userName* has sufficient rights to access *objectName*. If not, access is denied. If so, the server returns the *groupID* of the group currently sharing *objectName*, and a unique *objectID* which is subsequently used to refer to the specified object.

The server requires some time delay to check perform object name to *objectID* conversion and access rights. The response message is always sent as a separate *dataResponse* message. The server multicasts this response packet to all clients in the group associated with the resulting *objectID*. Each recipient immediately adds the requesting client to the group. This avoids any possibility of a client leaving the group in the interval between receipt of the *dataResponse* message by the client and that client acquiring the token to send a message to the other clients in the group indicating that it is joining the group.

When a client is finished referencing an object, it leaves the group. This is accomplished with the *leaveGroup* message:

leaveGroup(*objectID*)
The client multicasts the *leaveGroup* message to the group associated with *objectID*. Each recipient deletes the originating client from its stored copy of the group for *objectID*.

Because client caches retain blocks associated with an object after the last process closes the object, there is no explicit message to close off access to an object. When the server receives a *leaveGroup* message, it deletes the sender from the indicated group. If there are no more members in the group, the object may not be accessed until the server receives another *open* message.

4.2.4.2. Cache control messages

Four messages implement the cache coherence protocol. Figures 4-2 and 4-3 show the actions of the client and server caches, respectively, on receipt of any of these messages.

sfetch(*objectID, blockNumber*)
A client multicasts an *sfetch* (for *shared fetch*) message to the *groupID* associated with the object referenced by *objectID* to receive a *public* copy of the logical block *blockNumber*. The current owner of that block marks the original packet indicating that it will respond, and sends a copy of the block in a separate *dataResponse* message. The owner converts the block to *public* status if the block is *private*.

pfetch(*objectID, blockNumber*)
A client multicasts a *pfetch* (for *private fetch*) message to the *groupID* associated with the object referenced by *objectID* to receive a *private* copy of the logical block *blockNumber*. The current owner of that block marks the original packet indicating that it will respond, and sends a copy of the block in a separate *dataResponse* message. If the block is modified, the *dataResponse* message is multicast to both the requestor and the server, effecting a write-back of the new contents to the object store. Ownership of the block is transferred to the requestor. All caches other than the requestor invalidate the block, if it exists in their caches.

replace(*objectID*$_1$, *blockNumber*$_1$,*objectID*$_2$, *blockNumber*$_2$)
The owner of logical block *blockNumber*$_1$ of the object referenced by *objectID*$_1$ multicasts a *replace* message to the *groupID* associated with the object referenced by *objectID*$_2$ to replace block *blockNumber*$_1$ with logical block *blockNumber*$_2$ of object *objectID*$_2$ in its cache. Ownership of block *blockNumber*$_1$ is transferred to the server. (Recall that the server is part of every *groupID*.) If block *blockNumber*$_1$ is *modified*, the new contents are appended to the original packet for the server to read, effecting a write-back to the object store. The owner of block *blockNumber*$_2$ marks the packet indicating that it will respond and returns the contents of block *blockNumber*$_2$ to the requestor in a separate *dataResponse* packet. Clients use a special case of the *replace* message, with both *objectID*$_1$ and *blockNumber*$_1$ set to zero, to return ownership of blocks to the server when closing an object.

The group associated with object *objectID*$_1$ need not be involved in this transaction. Only the client and server are concerned with the transfer of ownership of this (or any) block, since the owner of a block is responsible for responding to such requests. At the completion of this message, the server owns block *blockNumber*$_1$, and will respond to any future requests for it.

Alternatively, clients could replace owned blocks by sending a message to give ownership back to the server, followed by an appropriate fetch message. The *replace* message saves one message every time a block is replaced.

private(*objectID, blockNumber*)
A client multicasts a *private* message to the *groupID* associated with the object referenced by *objectID* to convert the copy of block *blockNumber* that it holds from a *public* copy to a *private* copy (which may subsequently be modified by the requestor). Ownership is transferred to the requestor. If block *blockNumber* is *modified*, the current contents are sent to the server in a separate *dataResponse* packet, effecting a write-back to the object store. All caches that hold a copy of the block, other than the requestor, invalidate the block.

4.2.4.3. Data response messages

All responses that can not be included in the original message, because the response takes a long time to compose, are returned in a separate *dataResponse* packet. An example is when the server must respond with an object block that does not reside in the server's cache, but must be fetched from the backing store.

When a client sends a *pfetch* message, the client cache assumes ownership of the block as soon as the message has completed transit of the ring. The client is now responsible for answering further fetch requests, but may not yet have the data for the object block. When this is the case, the client marks the request to indicate that it will respond, and keeps track of which other clients have requests pending for that block. When the owner finally receives the data for the block, it sends a *dataResponse* message to all pending clients, indicating that they should resend their request. At this point, the owner has the data and can satisfy the requests immediately.

Ownership transfer via a *private* message from another client during this pending period does not cause a problem. Upon receipt of the original *pfetch* message, all caches that held copies of the requested block, but were not the owner of the block, invalidated their copies. The time delay in response occurs only because the server is the owner. While waiting for the data for the block

to be returned, it is guaranteed that there are no other copies of the block in any cache. Thus, no *private* message will be sent during this interval.

4.2.4.4. Emergencies

In an ideal situation, the above messages suffice to implement the cache coherence protocol of the Caching Ring. However, the stations on the network can fail, and may fail while holding ownership of some blocks. Thus, we add a ninth message:

bailout(*objectID, blockNumber*)
A client sends a *bailout* message to the server when it receives no response from the owner of block *blockNumber* of the object referenced by *objectID*. The server also attempts to contact the owner, and, if this fails, issues a *private* message to become the owner of the block, and falls back to the most recently written-back copy of the block. The server returns this copy of the block to the requestor in a separate *dataResponse* packet.

The server then sends *private* messages for all blocks that it knows are owned by any client. After this, the server removes the failed station from the group by sending a *leaveGroup* message on the behalf of the failed client.

Because the protocol includes a write-back of a modified block every time the ownership of that block changes, the amount of data lost if a client crashes is, at most, the updates made by the current owner.

We expect that stations will fail due to processor failure rather than network failure. The architecture of ring networks does not allow continued communications when any ring interface has failed. Clients can check that received messages are from a station that is recorded as a member of the group. If the source is not a member of the group, the receiver can disregard the message and notify the source that it is not part of the group. This prevents the client processor that survives a network failure from disrupting the coherence protocol.

4.2.4.5. Cost of the messages

The absolute time delay for the delivery of a network message is dependent upon the number of stations in the network, because each station adds a fixed delay to delivery times. However, we can compare the cost of the various messages by expressing the delays in terms of ring delays—the time required for a message to make one complete circuit of the network. Thus, we can estimate the expected delay for various operations on the ring.

Again, we consider four possible cases: read hit, read miss, write hit, and write miss. Expected delays for each case are as follows:

- **Read hit:** There is no network delay.
- **Read miss:** The expected delay for a read miss is one or two ring times. The client first issues a *sfetch*, with a delay of one ring time. If the owner is another client, the block resides in the owner's cache, and the contents can be appended to the original packet. If the server is the owner, and the block is not in the server's cache, there will be a delay while the block is fetched from the object store. The server marks the packet stating that it will respond, and fetches the block. The server then sends a *dataResponse*, with a delay of one ring time. The delay of fetching the block from

the object store can be partially overlapped with the delay for the *sfetch*, depending on the relative locations of the client and server on the ring.

- **Write hit:** The expected delay for a write hit is zero or one ring time. If the block is in the **Dirty** state, no message need be sent. The copy of the block is known to be the only one in the system, and can be modified immediately. If the block is in the **Shared-dirty** state, other caches may hold a copy of the block, and a *private* message must be sent, with a delay of one ring time, to invalidate those other copies.
- **Write miss:** The expected delay for a write miss is one or two ring times. The client first issues a *pfetch* message, with a delay of one ring time. If another client is the owner, the block must reside in the owner's cache, and the contents are appended to the original packet. If the server owns the block, and the block is not in the server's cache, there will be a delay while the block is fetched from the object store. The server marks the packet stating that it will respond, and fetches the block. The server then sends a *dataResponse*, with a delay of one ring time. The delay of fetching the block from the object store can be partially overlapped with the delay for the *pfetch*, depending on the relative locations of the client and server on the ring.

4.2.5. Semantics of shared writes

Since there can be a delay of up to one ring time between the time a client issues a *private* or *pfetch* message and the holder of a copy of the block receives it, there is an interval during which the holder of a copy may provide stale data. Consider two clients C_1 and C_2, sharing block *B* of an object. C_2 is the current owner of block *B*. A process on C_1 writes into block *B*. In response, the CRI at C_1 sends a *private* message to C_2, requesting ownership of *B*, and subsequent removal of *B* from the cache at C_2. Between the write and the time C_2 receives the message, a process on C_2 can read the cached block, and receive (now) stale data.

A desirable solution would be for the write at C_1 to fail in this case, since the write is stalled pending transfer of ownership. This requires that messages be timestamped, which further requires that the stations in the distributed system have synchronized clocks. We prefer not to require these attributes. In essence, the semantics of writing on the Caching Ring are such that the write is not complete until the *private* or *pfetch* message has completed its transit around the ring. This is different from the immediately complete writes that many simple UNIX programs assume, and can lead to problems as described here.

Since our evidence shows that shared access makes up only about 10% of the measured activity on our UNIX system, and only 5% of those accesses involve writes, we require that a higher-level locking protocol be enforced between the sharing clients. This is consistent with the standard UNIX file system. Other systems built on top of the Caching Ring must adapt to these semantics, perhaps by building a higher-level locking protocol using the command packets provided by the CRI.

Time	C_1	C_2
T_0		pfetch(*B*)
T_0+1		
T_0+2		
T_0+3	write(*B*)	
T_0+4	send private(*B*)	
T_0+5		
T_0+6		read(*B*)
T_0+7		
T_0+8		
T_0+9		receive private(*B*)
T_0+10		invalidate *B*
T_0+11		
T_0+12	private(*B*) returns	
T_0+13	complete write(*B*)	

Figure 4-4: Timeline showing contention problem

4.2.6. Motivation for this design

This algorithm reflects the results discussed in Chapter 3. Most importantly, it places the naming, access control, directory, file, and disk managers at the server. This eliminates the network traffic that would be required to transfer the contents of and serialize access to the data structures used to implement the file system.

We concluded that performance of reads should be of the highest importance, so the CRI imposes no communications or synchronization delay on read hits. This, in turn, leads to the shared write semantics discussed above. If shared writes were expected to occur more frequently, the coherence algorithm could force clients to synchronize through the CRI on a read hit. Since processor requests have the lowest priority in the CRI, all pending invalidation messages would be processed first, and the stale data would not appear in the cache. However, this leads to a higher average read access time.

Similarly, if writes occurred more often, we would consider transferring ownership with every fetch message, be it *pfetch* or *sfetch*. This would eliminate some of the time spent waiting to attain ownership of a block before writing it. This would lead to higher write-back traffic to the server, since modified blocks are written back with every ownership change. Alternate strategies to predict the need to make a block *private* would also need to be explored.

4.3. Summary

We have presented the design of the Caching Ring, an intelligent, network-based caching system for use in a distributed system. Utilizing the intelligent interface, a client workstation can access objects from a central repository and share portions of these objects with other workstations. To improve performance, the intelligent interface contains memory to act as a cache of recently used objects. This cache is used to satisfy requests to access objects whenever possible.

The intelligent interface implements a network protocol to maintain consistency among the caches of clients sharing an object, and to minimize the amount of data lost if a client fails.

5. A Simulation Study of the Caching Ring

To investigate the performance of the Caching Ring, we designed a simple distributed file system in which disk files are the shared objects. Using the activity traces discussed in Chapter 3, we simulated the performance of the Caching Ring for several different system configurations.

This chapter extends our understanding of caching to the performance in distributed systems. Using simulation, we were able to determine the effects of cache size and placement, and locate the expected performance bottlenecks in a distributed caching system.

5.1. A file system built on the Caching Ring

We now present the design and analyze the performance of a distributed file system built using the facilities provided by the Caching Ring. The file system has the semantics of the 4.2BSD UNIX file system, although we did not implement all the primitives. We did design mechanisms for opening, closing, reading, writing, and seeking on files, as well as accessing and maintaining the directory structure. The implementation of the remaining status and maintenance primitives is straightforward using command packets, but these primitives do not play a part in measuring the performance of the Caching Ring.

5.1.1. Architecture of the Caching Ring file system

The file system is based on a set of diskless clients and a single central file server. The server maintains a complete file system that is shared by all clients. The same file name space is shared by all clients in the system. Clients use file names to identify files when opening them. Thereafter, the server and clients use *objectID*s to identify open files.

The server also maintains the naming manager and the directory system, providing primitives for atomic directory access and maintenance across the network, using command packets. In using these primitives, the client operating systems do not need to know the structure of the directory system, and do not transfer large amounts of data across the network while searching or modifying the directory system. Updates to directories are guaranteed to be synchronized, because they are serialized at the server.

5.1.2. Implementation of the file system

The implementation of the file access primitives is a straightforward mapping onto the primitives provided by the Caching Ring. To open a file, a client sends an *open* packet with the file name, authentication information, and intended access mode. The server checks for access rights, and returns an *objectID* that describes the file and a *groupID* that describes the group of clients currently sharing the file. This response is multicast to the group, as described in Section 4.2.4.1. If the open fails for any reason, the server returns a failure message to the client indicating the reason for the failure.

To read a block from the file, the client operating system first checks the CRI cache directory to see if the file is already in the cache. If so, the cached copy is used with no intervention by the CRI. If not, a read request for the block is passed to the CRI. The CRI multicasts an *sfetch* message to the *groupID* associated with the file, which is recorded in the group mapping cache in the CRI, and waits for the response from the Caching Ring.

To write a block of the file, the client operating system issues a write command to the CRI. If a copy of the block is already in the cache, and not in the **Transition** or **Dirty** state, the CRI multicasts a *private* message to the file's *groupID*, and sets the state of the entry to **Transition**. If there is no copy of the block in the cache, the CRI sends a *pfetch* message to the corresponding *groupID* and waits for the response. When the response arrives, it contains the current data for the block. The CRI places this data in the cache, sets the entry state to **Transition**, writes the new data, and sets the state to **Dirty**. Only then is the client operating system notified that the write has completed.

To close a file, the client operating system decrements the file reference count. When the reference count reaches zero, the client operating system issues a close command to the CRI. The CRI returns ownership of all cached blocks associated with the file to the server, as described in Section 4.2.4.1. When all of these blocks are subsequently invalidated or replaced, the CRI multicasts a *leaveGroup* to the associated group, and is no longer party to messages concerning that file.

The object names passed in an *open* message to the server are marked as names that come from the file system portion of the shared object name space that the server provides to the system. Once the file object has been opened, all transactions are in terms of *objectID*, and the server need not differentiate between file objects and other objects that it provides to clients on the Caching Ring.

5.1.3. Operating system view of the file system

To the higher level operating system routines that use the file system, this distributed file system has the identical semantics as a file system implemented on a locally attached disk. Files appear as an unformatted stream of bytes.

Files are named by text strings, and once open, referred to by small integers, known as *file descriptors*. Internally, the file system maps these small integers into the *objectID*s used by the CRI.

Requests to read and write from the file contain a file descriptor, the starting address of a buffer in which to place or from which to copy the data, and the number of bytes to transfer. The file system maintains the current offset in the file at which to begin operations. This offset can be altered by the *seek* primitive. The file system computes the logical block numbers of the range of bytes affected by the request, and uses the CRI to access those blocks.

5.2. Simulation studies

To evaluate the performance of the Caching Ring, we wrote a program to simulate this distributed file system. We drove it with the activity traces described in Chapter 3, and, by varying parameters in the simulated system, gained insight into the performance effects of different portions of the system. These simulations also allow us to compare the performance of a Caching Ring distributed file system to the performance of other file systems.

5.2.1. Description of the simulator

The simulator is written in the CSIM simulation language [26]. CSIM is a process oriented simulation language based on the C programming language. It supports quasi-parallel execution of several software processes. The basic unit of execution is a process, and each process can initiate sub-processes. Each process has both a private data store and access to global data. Processes can cause events to happen, wait for events to happen, and cause simulated time to pass. Much of the data gathering associated with simulation models is automated, and is easy to extend. The language is implemented primarily as calls to procedures in a runtime library, so the full power of the host operating system is available to support features such as I/O and dynamic memory management.

The process model of the simulation language serves as a convenient environment in which to do concurrent programming [27, 3]. The simulated processes can be thought of as executing in parallel, sharing a common address space. CSIM directly provides events that are similar to *semaphores*, and can be used for process synchronization [14, 15]. Some cooperating processes need to pass data back and forth as well as synchronize. For them, we implemented message passing queues on top of the primitives provided by CSIM.

The model of the caching ring contains a process for each client processor, each CRI, and the server. These processes communicate through message passing queues, and synchronize by waiting for access to the ring or for synchronization events to be set. Access to the ring is controlled by a token, which is passed from cache to cache. When the ring is idle, any CRI can take the token and send on the ring. The message is placed on the input queue of the next station on the ring, and the token is passed to that station. Each CRI executes in the loop of ‘‘receive packet, act on packet, forward packet’’ until the associated processor has some work to be done. A CRI removes an inbound packet when it notices that it is the originator of that packet. The originator of the packet advances simulated time by the appropriate amount for the length of the packet. Unless mentioned otherwise below, the simulated transmission rate on the ring is 10 Mbits/sec, the delay through each CRI is 32 bit times, and there are 32 stations on the ring.

A processor with a request for the CRI places a message on the CRI’s incoming request queue, and resets and waits for a completion signal from the CRI. If the ring is currently idle, the CRI forms an appropriate message, acquires the token, and transmits the message on the ring. When

the response appears in the CRI input queue, the result is placed in the block cache, and the processor is signalled to continue. If the ring is not idle, the CRI forms the message and waits for the token to appear. After processing and forwarding all inbound messages, the CRI adds its message to the next station's input queue, and only then passes the token. This simple model of the processor does not accurately model a multiprogrammed operating system. Once a processor begins waiting for the CRI to respond, it is blocked from completing other operations that might be satisfied from the cache. We have, however, found this model sufficient to discover many factors affecting the performance of the Caching Ring.

The server is implemented as two processes with message queues between them. One process simulates the server CRI, and the other manages the simulated disk at the processor. The CRI places incoming open and block fetch requests on the disk queue, and continues network operations. The disk process delays for an amount of time appropriate to the request, and places a response on the CRI incoming request queue. At the next opportunity, the CRI acquires the ring token, composes a response, and sends it to the requesting client.

Each CRI maintains data structures similar to those used for the simulations in Chapter 3. There is an open object table and a set of cache blocks that the CRI manages according to an LRU replacement policy. These are shared by the processor and CRI. In addition, there is a table mapping *objectID*s to *groupID*s, for composition of network addresses, and a table containing the state of each entry in the cache block (**Invalid, Shared-valid**, etc.)

5.2.2. Using the trace data

To compare the simulated performance of the Caching Ring file system with the simulated results in Chapter 3, we generated processor requests from the same activity traces. Once again, we used only trace records generated by the file manager, thus simulating only I/O activity generated by user processes. The original traces recorded the user and process identifier of the process making each request, so it is straightforward to split the traces into separate streams, one stream per simulated client. Unfortunately, there is a fair amount of file system activity due to various system tasks. This activity must be accounted for in some manner. It cannot simply be ignored, nor can it necessarily be duplicated and charged to each simulated client. We have used the traces in the simulations reported below in ways that attempt to treat this system activity fairly. As in Chapter 3, the four original traces produced nearly indistinguishable results; we report only the results from trace A.

5.2.3. Miss ratio *vs.* cache size

In order to characterize the basic performance of the Caching Ring, we simulated a simple system with one active client. We varied the cache size and network transmission parameters to determine their effects on the miss ratio, effective access time, and utilization of the disk and network. The block size is 4096 bytes and the cache is write-back, both of which were shown in Chapter 3 to be optimal for this set of traces. The disk service time, exclusive of any network delay, is 30 ms/block. To isolate the effects of the client cache and network, there is no cache at the server. The results of the cache size *vs.* miss ratio experiments are shown in Figure 5-1 and Table 5-1.

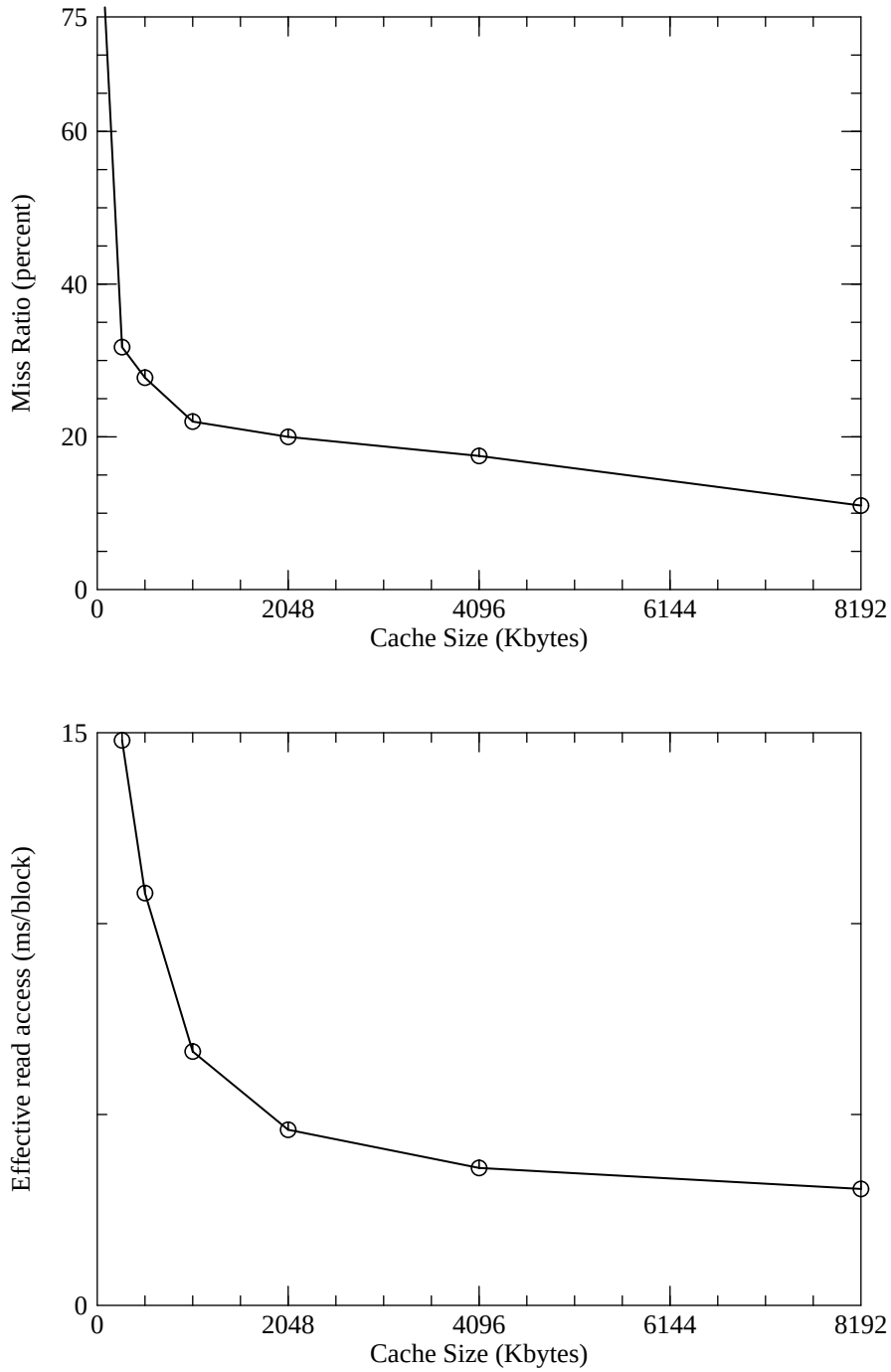


Figure 5-1: Miss ratio and effective read access *vs.* cache size

The miss ratios are higher than those reported in Table 3-1. This is the effect of the block replacement policy enforced by the CRI on file closes. Every modified block is written-back to the server when the file is closed. Thus the write miss ratio is close to 100%, which affects the overall miss ratio we report. Another cause of the increased miss ratio is the action of the protocol on write misses. to be written would fully replace an existing block. In our simulation in Chapter 3, if a block to be written was not in the cache, we did not charge a write miss or add

Cache Size	Miss ratio	Effective read access time (ms/block)
256 Kbytes	31.7%	14.8
512 Kbytes	27.8%	10.8
1 Mbyte	22.0%	6.66
2 Mbytes	19.9%	4.61
4 Mbytes	17.5%	3.58
8 Mbytes	10.9%	3.07

Table 5-1: Miss ratio and effective read access *vs.* cache size

any delay to the total write delay; the cache merely allocated a block and filled in the new data. In the Caching Ring coherence protocol, it is necessary to send a *pfetch* message and wait for a response before writing the block. These differences add to both the overall delay experienced and to the write miss ratio.

Because we compute the effective access time as the total delay seen by the processor when reading divided by the number of bytes read, this policy has minimal effect on the effective access time. The only effect would come from the slightly increased network traffic from returning modified file blocks to the server on close. For comparison, the access time of the disk is 30 ms/block, and the access time of a typical network disk is typically 120 ms/block.

Our results show that the Caching Ring provides a great performance improvement over a network disk access mechanism. We attribute this to several factors.

First, by placing the directory, access control and disk managers completely at the server, we eliminate the communications overhead associated with accessing manipulating the data used by these managers across the network. The managers can use a memory cache of recently used objects to further enhance performance.

Second, the server contains the portion of the file manager that maps logical file blocks to physical disk blocks and interacts with the disk manager. This eliminates the remaining overhead associated with maintenance of the file system. This placement decision eliminates approximately 50% of the traffic to the server[5].

Furthermore, the network protocols involved have very little transmission and processing overhead. In contrast, a network disk access mechanism such as Sun Microsystems' ND is layered over several general-purpose protocols [48]. This layering incurs a costly checksum computation at several points in the procedure of accessing a disk block. The Caching Ring avoids this overhead by using a specialized lightweight protocol.

Lastly, the computing equipment servicing the protocol is required to do nothing else. In ND, the server machine is running a timesharing operating system and has a slow response time to

[5]See Section 3.6.

processing incoming network packets. The server in the Caching Ring need only concern itself with processing client packets as quickly as possible.

We also measured the utilization of the network and server disk for the various cache sizes. The results are shown in Table 5-2.

Utilization		
Cache Size	Disk	Network
256 Kbytes	21.7%	0.5%
512 Kbytes	17.4%	0.5%
1 Mbyte	14.7%	0.4%
2 Mbytes	13.2%	0.4%
4 Mbytes	12.3%	0.4%
8 Mbytes	11.9%	0.4%

Table 5-2: Disk and network utilizations for various cache sizes

The disk at the server is the likely bottleneck, though no queueing occurs at this load. The network is very lightly loaded.

5.2.4. Network latency

To determine the effects of the network subsystem, we repeated the previous experiment, varying the delay through each processor and the number of stations on the ring.

The performance of the Caching Ring depends on the ability to respond to requests in the original packet containing the request. The acceptable delay through the station has a great impact on the technology used to implement the CRI and the resultant complexity and cost. For this experiment, we found that a delay of 8 bits at each station instead of 32 reduced the utilization of the network by 20%, and a delay of 64 bits increased the utilization by 20%. In either case, total network traffic still demands less than 1% of the available network bandwidth. The effect on the effective read access time in each case was less than 0.1%.

The number of stations on the ring affects the total delay experienced in a ring transit time. We varied the number of stations on the ring from 1 to 32, and noticed effects on the ring utilization and effective access time similar to those produced by varying the delay at each station.

We conclude that the transmission characteristics of the network are not a large consideration in the performance of the Caching Ring. Adding more active stations to the network will of course generate more traffic, which will increase utilization and delay at each point in the system. We explore this more fully below.

5.2.5. Two processors in parallel execution

To test the robustness of the cache coherence algorithm and load all components of the system, we simulated a system in which two client processors execute exactly the same activity. Operation of the system proceeded as follows: C_A would send an *open* to *S* and wait. C_B would send an *open* to *S* and wait. *S* would complete the first open and send the result to C_A. C_A would issue the first fetch operation and wait. *S* would complete the second open and send the result to C_B. C_B would send its first fetch operation. If the first operation was a write, C_A has assumed ownership of the block, and would indicate that it will respond to C_B. Otherwise, *S* indicates that it will respond. *S* completes the fetch and sends the block the C_A. If C_A is writing the block, it does so and sends a *resend* to C_B. C_B reissues the *pfetch*. C_A responds, invalidating its copy and sending a copy to the server. C_B receives the block and writes it. This continues until the end of the trace is reached.

Utilization of the disk is considerably higher, as high as 44% for a 256 Kbyte cache, primarily because almost every write involves a write miss, and the frequent change of ownership of blocks causes many more updates to be sent to the server. Network utilization is also higher, for the same reason. This experiment shows that the coherence protocol is sound and does not reveal any timing races under high sharing loads.

5.2.6. Simulating multiple independent clients

We now consider the performance of the Caching Ring system under a more typical load from several independent users. Ideally, we would like to separate the individual activity streams into the activity generated by each user of the system. However, as discussed earlier, we must account for file activity generated by system-owned processes. We also wish to place a heavier load on the system than is recorded in our original traces.

To generate activity streams, we split the original stream into as many streams as necessary to drive the desired number of simulated processors. We duplicated existing streams as necessary. To avoid uncharacteristic levels of sharing activity, processors started at different places in the activity stream and wrapped around to the beginning.

Our analysis of the traces in Chapter 3 showed that the majority of file sharing is in system files. Those files live on a particular device on Merlin’s disks, and can easily be identified in the trace. To further support the independence of the generated activity streams, referenced files that do not lie on the system disks were renamed, so that the only candidates for sharing between processors are the system files.

We generated simulations with two different classes of processors. The first replicated the activity traces without separating out individual users. This simulates the effect of several timesharing systems on the network, and accurately depicts the load offered by the system processors. The second also separated out the individual activity traces, and assigned each to a separate processor. The activity from the system tasks was randomly assigned to a processor. The results from these two sets of simulations were nearly indistinguishable; we report only the results of simulating timesharing systems, as they seem to be a more accurate use of the traces.

We simulated systems with one to six client processors, representing approximately two to twelve active users. Our simulation results are shown in Figure 5-2.

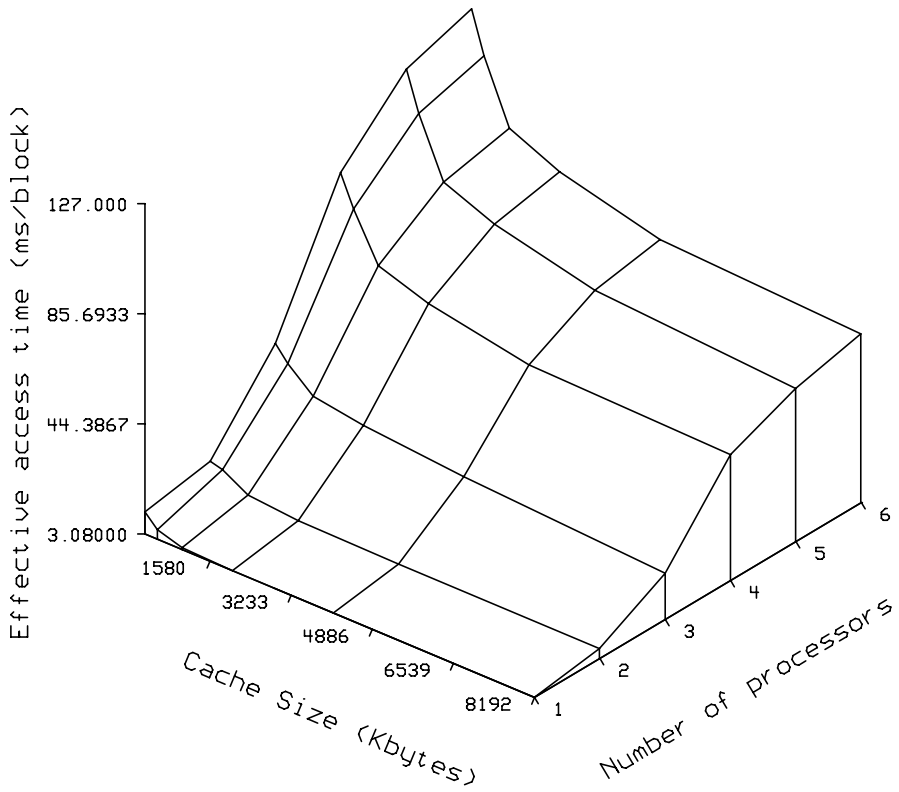


Figure 5-2: Multiple timesharing systems on the Ring

The effective disk block access time rises sharply after adding the fourth system to the ring. It flattens out almost completely after adding the fifth system to the ring. We inspected the utilizations of the various components of the system, and found the utilization of the disk to be high at these points, with queue lengths at the disk growing to be larger than 1. Figure 5-3 shows the server disk utilization percentages.

Comparing these two graphs, we see that as the utilization of the server disk increases, and with it the queue length, the effective disk service time increases to above the hardware delay. This causes the effective access time to rise. The effects of the client cache are still visible, since the miss ratio at the cache does not change. Because the service time at the server has become dependent on the load, the effective access time necessarily also changes.

At high loads, the size of the cache does not affect the server disk utilization. This is because the disk is being saturated by open and close requests, and the cache can not affect the I/O required to perform these.

Depending on the cache size chosen for the client processors, we conclude that the system can support approximately ten users on the ring before the server disk becomes a significant bottleneck. At maximum load, the utilization of the network is only 1.8%. Thus, we expect that ring throughput will not be a performance bottleneck without much heavier loads.

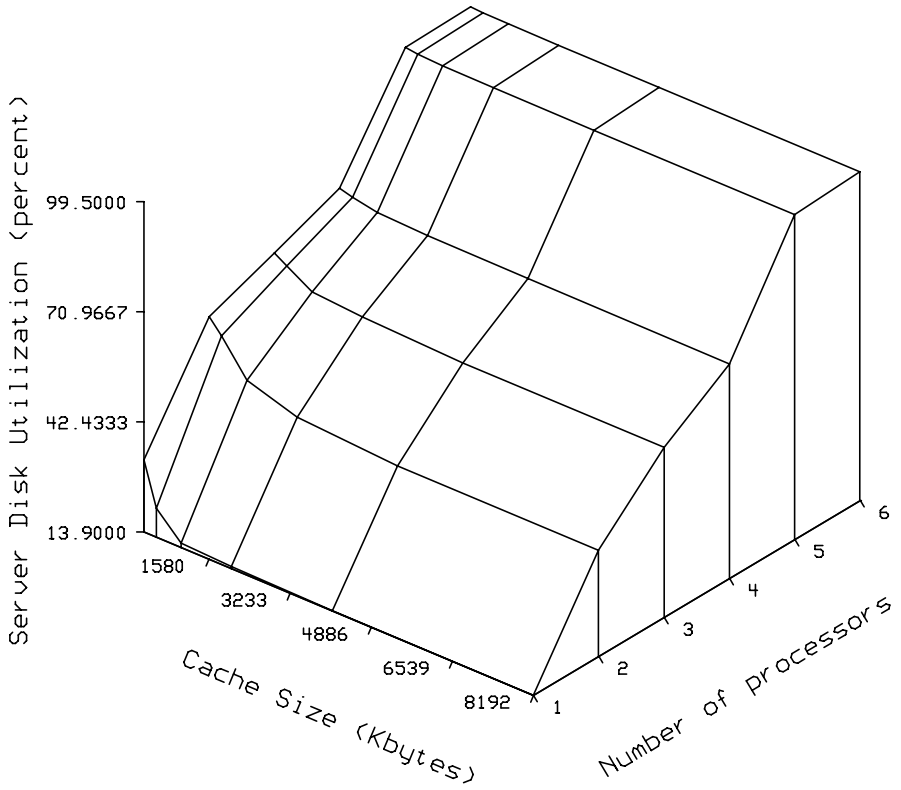


Figure 5-3: Server disk utilization *vs.* timesharing clients

5.2.7. Size of the server cache

How can we shift the bottleneck of the server disk? The service time can be decreased in several ways. The first is by adding second disk to share the load. If properly managed, this can reduce the disk service time by as much as 50%, but this figure can only be achieved if files are arranged so that the load is always evenly balanced. Similarly, the disk can be replaced with a faster one; however, there are limits to the transfer rate of current disk technology, and the figure of 30 milliseconds per block used in this simulation is based on one of the fastest currently available devices.

Another way to decrease the disk service time is to add a disk block cache at the server. We repeated the experiments of the previous section with a range of cache sizes at the server, and discovered that even a modest cache has a significant effect on the location of the bottleneck.

We found that even a modest cache of 256 Kbytes provides a large improvement in performance, pushing the bottleneck out to approximately 14 systems, or 28 users. A large cache places the bottleneck beyond the largest system we were able to simulate (a system with twenty timesharing systems on the ring). The cache at the server provides the same relative performance enhancement that it provides at the client.

These simulation results correspond well to those reported by Lazowska *et al.* in [32]. In the most efficient configuration that they measured, they expect a single server to support approximately 48 clients before exhibiting response times above those that we consider typical for a network disk. By adding the effects of large caches in the system, we can expect performance to improve to the levels presented here.

Neither of these methods reduces the I/O load generated by the naming manager. This can be reduced in several ways: moving the on-disk data structures used by the naming manager (the directories) to a separate disk, locating the naming manager on a separate server machine that does not handle file disk traffic, or using a cache of name lookup information. As we did not simulate the performance of the naming manager, we did not investigate the effects of changes of this type.

Again, we do not consider paging traffic to the server. A typical Caching Ring would probably use the server for both file activity and paging activity, and utilizations would then be higher. It may also be unwise to consider a configuration in which so many clients are dependent on a single server for reliability reasons.

5.2.8. Comparison to conventional reliable broadcast

We now consider implementing the Caching Ring coherence protocol on a conventional broadcast network such as the Ethernet. Both the Ethernet and the Caching Ring network have transmission rates of 10 Mbits/sec, so the performance comparison is an interesting one to make. Because the coherence protocol of the Caching Ring depends on reliable broadcast transmission of packets, we must ensure reliable transmission on the Ethernet. The Ethernet has a ‘‘best-effort’’ delivery policy; it will make a good effort to deliver packets reliably, but does not guarantee delivery. To guarantee reliable transmission of broadcast, an additional protocol mechanism must be introduced.

Chang and Maxemchuck describe an algorithm that is typical of such protocols used on an Ethernet in [10]. The protocol constructs a logical ring of the hosts that wish to reliably broadcast messages to one another, and uses point-to-point with acknowledgements to messages to move the broadcast packets around this ring. For a group of N hosts, the protocol requires at least N messages to be sent on the network for each reliable broadcast.

The Ethernet is known to have poor performance on small packets [47]. This is largely because the Ethernet channel acquisition protocol enforces a 9.6 μsec delay between the end of one packet and the beginning of the next. Since the control packets in the Caching Ring are all small, we expect this to have a serious effect on the performance.

The Ethernet limits packet sizes to 1536 bytes, including protocol overhead. Since the Ethernet is designed to accomodate a wide variety of network applications, there is a large amount of space for protocol headers in a packet, with room for approximately 1 Kbyte of data. For our optimal block size of 4 Kbytes, it will take four Ethernet packets to transmit a single block. We compose the following bit count for one of these packets: 1024×8 bits of data, 64 bits of *objectID*, 96 bits of addressing information, 16 bits of packet type, and 32 bits of block number, for a total of 8400 bits. Each Ethernet packet is preceded by a 64 bit preamble, and followed by a 96 bit delay, for a total of 8560 bits, or a total delay of 856 μsec. Four packets are needed to transmit a cache block, for a total delay of 3424 μsec.

To transmit the same amount of data on the Caching Ring, we build a single packet consisting of: 4096×8 bits of data, 64 bits of *objectID*, 96 bits of addressing information, 16 bits of packet type, and 32 bits of block number, for a total of 32976 bits, or a total delay of 3297.6 μsec. In addition, the packet will experience a 32 bit delay through each of the 32 stations on our ring, adding 102.4 μsec for a total delay of 3400 μsec. Thus, we see that the transmission delay for large packets in the two network technologies is quite similar.

However, after this delay, all stations on the ring have seen the packet, and only one on the Ethernet has. We must multiply the total delay on the Ethernet by the number of stations in the group of clients sharing the file. The largest number of processors that had a file open simultaneously in any of our traces is nine. Sharing levels of two or three are more typical, making up over 80% of all shared accesses.

We thus conclude that the Caching Ring protocol can be implemented on an Ethernet using a reliable broadcast protocol. We expect that this implementation would perform at one-half to one-third of the simulated performance reported here.

5.3. Conclusions

We have presented the design of a distributed file system that uses the Caching Ring to allow workstations to access remote files efficiently and share portions of files among themselves. This file system implements the semantics of an existing well-known operating system, 4.2BSD UNIX.

We used a simulation of this file system to examine the performance characteristics of the caching ring. The primary result is that the performance of the disk at the server is the bottleneck. A server with a high-performance disk can serve approximately 24 active users before those users see a degradation in throughput to levels below those of a typical remote disk. This can be extended to 32 users by adding small a disk block cache at the server, and 40 or more users by adding a large disk block cache at the server.

The simulations show that the Caching Ring hardware, with a sufficiently large cache at each CRI, can provide clients with access to a remote disk at performance levels similar to those of a locally attached disk with a small cache.

When the server is not the bottleneck, the performace of the file system appears to the client only slightly worse than the performance of a local disk accessed through a cache of the same size. When the server bottleneck is reached, performance quickly degrades to the performance of a typical remote disk, as described in Chapter 3. A large cache at the client keeps even performance of this configuration similar to the performance of a locally attached disk with a cache the size of that used in our VAX timesharing systems.

We attribute the performance of the Caching Ring to the large caches at each client, and the low overhead imposed by the protocol and hardware implementing the cache coherence algorithm. The amount of network traffic generated by the cache coherence messages is small enough that we believe there to be enough communication bandwidth to support a hundred or more workstations. This is primarily because there is so little sharing of files; most communications are simply between a client and the server, and the low overhead communications add little penalty to the basic operation of reading or writing the disk.

Furthermore, we conclude that the Caching Ring protocol can be implemented on a conventional broadcast network such as the Ethernet, but we expect that performance will be two to five times less than the performance on the Caching Ring.

6. Summary and Conclusions

Our research has been directed toward an understanding of caching in distributed systems. After surveying previous work, we measured and analyzed the effects of caching in the UNIX file system to extend our understanding of caching beyond just caching in memory systems. We then designed a caching system, the Caching Ring, that provides a general solution to caching of objects in distributed systems. We analyzed the performance of the Caching Ring under simulated loads based on trace data taken from an existing system, and discussed how the Caching Ring addresses the problems of caching in distributed systems. The remainder of this chapter summarizes the contributions of this research and proposes future directions for the investigation of distributed caching systems.

6.1. Caching in the UNIX file system

On the average, active UNIX users demand low data rates from the file system. But the traffic pattern is bursty. UNIX depends heavily on objects stored in the file system, so an active user can have short periods of activity that demand as much as ten times the average data rate. A moderately sized disk block cache greatly reduces the total disk traffic required to satisfy the user requests.

More than 50% of the total disk traffic is file system overhead. This is the cost of managing on-disk data structures that map logical file blocks to physical disk blocks, implement the naming hierarchy, and provide access control information.

We used simulation to show that the economics of current memory and disk technologies provide two alternatives for increasing the file system performance of a processor. By adding a large memory to be used as a file block cache, the effective performance of a disk with a slow access time can be increased to compare to that of a disk with a faster access time.

UNIX users rarely share disk files. Of all file accesses in our sample, only 4% were to files opened simultaneously by more than one user. Most of those were read-only uses of system files; of the accesses to shared files, only 8.8% were writes.

6.2. Caching in distributed systems

Based on the low demand placed on the file system by active users, we would expect that the network bandwidth available from a conventional network such as an Ethernet would support many users. Further simulation showed that the bandwidth implied by the transmission speed of

the network is not necessarily available for the transmission of information. Delay through interfaces or in the medium access protocol can greatly reduce the bandwidth available to a particular workstation, with a significant effect on the performance of a distributed file system.

6.2.1. Location of managers

Our model of a file system divides the activity into five separate managers, each handling a different type of object related to the operation of the file system as perceived by the clients. The placement of the processes implementing each of these managers, either local to the client or remote on a server, can have a significant impact on the amount of traffic on the network, the CPU overhead on the client, and the performance of the server.

6.2.2. Cache coherence

The low general level of sharing of file blocks among client workstations lead us to conclude that a cache coherence mechanism is not as expensive to provide as was previously thought. In our data, most applications use private files, and reads outnumber writes by 2:1. Sharing, when it occurs, is largely for files that are read only. Thus, we presented the design of an efficient cache coherence protocol, based on earlier multiprocessor cache coherence mechanisms, which is optimized for the reading of private files. This protocol guarantees that blocks that exist in a cache are valid, and therefore incurs no communications cost on a read hit. Miss operations are inexpensive in terms of communications. Caching and writing of shared objects is fully supported, with no special locking action required by the client.

6.3. The Caching Ring

We present the Caching Ring as a high performance, general purpose solution for the caching of objects in a distributed system. The Caching Ring uses current technology to implement the coherence protocol discussed above, thus relieving the client processors of the overhead of maintaining the cache and network. The Caching Ring network hardware itself presents a novel addressing scheme, providing low-cost reliable broadcast to the clients.

In contrast to the other systems discussed in Chapter 1, the Caching Ring provides a general purpose mechanism for caching objects in distributed systems. It supports partial caching of objects, no delay on read hits, and the ability to cache shared objects. Updates to shared objects are immediately available to the other processors.

6.4. Future directions

Traces from other environments. Our trace data was taken from large processors running a timesharing system. We recorded the activities of each individual user, and used the activity of each user as an indication of the demand generated by a single workstation in a distributed system. There is some reason to believe that this is not a totally accurate representation of a client in a distributed system environment. For example, it ignores all the traffic generated by tasks that are considered system overhead, such as mail, news, and routine file system housekeeping tasks.

It would be instructive to obtain a set of activity traces taken from a set of workstations, and use the traces to drive this simulation of the Caching Ring, and compare the results to those presented here.

It would also be valuable to obtain traces from an environment in which large databases are used by many programs. In such an environment, we would expect readahead to have a more dramatic effect on the miss ratio, and perhaps see higher miss ratios over all.

Distribution of file system tasks. The experimental results presented in Chapter 3 led us to conclude that a distributed file system that uses file servers will perform more effectively than one that uses disk servers. This conclusion is based on the goal of minimizing network traffic, because we believe that network throughput is one of the most likely bottlenecks in a distributed system. The results show that more than 50% of the disk traffic in the UNIX file system is overhead due to user requests, rather than data needed for user programs. Eliminating this overhead traffic from the network allows more users to be served before the network reaches saturation.

This places an additional computational burden on the file server machines. Lazowska *et al.* concluded that in a distributed system using disk servers, the CPU of the disk server is the bottleneck [32]. With a higher performance CPU at the disk server, the disk then becomes the bottleneck. In a file server, we expect disk traffic to be reduced, because the server can keep a local cache of the information used by the access control, directory, and file managers. A large cache of recently accessed disk blocks used for files also reduces the total disk traffic.

A complete trace of the disk traffic generated by the file system managers and by paging would allow the tradeoffs of locating the different managers locally or remotely to be studied. A simulation could be built that implements each manager and allows them to be individually placed at the client or server. Driving this simulation with the complete trace would provide a total characterization of the necessary disk and network traffic in a distributed system.

Caching of other objects. Our study has centered around the sharing of file objects. We believe that the Caching Ring can be applied to the caching and sharing of other objects, such as virtual memory pages. A virtual memory system designed around the Caching Ring would provide the interconnection mechanism for a large scale, loosely coupled multiprocessor. The semantics of writing shared objects would have to be carefully defined, because the write semantics of the Caching Ring are not those typically found in memory systems. Li and Hudak indicate that the performance of a shared virtual memory system where the physical memory is distributed across a network should be adequate for general use [33].

Since the cache only knows the names of objects, it can be applied to objects of any type. In addition, the quick response of the ring for objects in the cache may lend itself to other distributed algorithms that need to collect small pieces of data or status information from several cooperating processors. For example, a distributed voting algorithm could be easily implemented using the basic packet mechanism provided by the CRI, with some changes in the message semantics to collect several responses in one packet.

Network issues. The current design of the Caching Ring network hardware and software is rather simplistic. It makes no provision for security of the communications on the ring. Intrusion to a ring network is more difficult than on a passive bus, but a malicious user of a workstation on the Ring could easily spoof the server into accessing any desired data. Mechanisms for insuring the authentication of clients and the accurate transmittal of data should be studied.

The protocols as described allow only one server node in the system. The server is the likely bottleneck in a large system, and it would be desirable to have more than one, to provide both additional performance and some measure of reliability or fault tolerance. One possibility is to include multiple servers in the system, each responsible for a subset of the object store. This is easily achieved by adding functionality to the naming manager. The naming manager currently translates object names to object identifiers. It could be extended to keep track of which portion of the object name space is handled by which server. The naming manager would then forward the open request to the appropriate server. The server would include itself in the original group, and the algorithm then proceeds as before.

The system is currently limited in size to the number of bits in an address vector. Since interface delay is such a large factor in the upper bound of system throughput, we would like the size of an address vector to be no larger than necessary, yet provide sufficient room for future expansion of an initial system. It is difficult to conceive of a Caching Ring system as large as the loosely coupled distributed systems built on Ethernets, where the available address space is 2^{47}, yet a moderately sized research facility might have a hundred or so researchers that desire to share data. Mechanisms for scaling the system other than increasing the number of stations on the ring should be investigated.

An investigation into implementing the Caching Ring protocols on a conventional network such as the Ethernet would be most interesting. We envision a dedicated interface similar to the CRI described in Chapter 4, with a reliable broadcast protocol implemented in the interface. We feel that the use of multicast addressing is important to performance of the system, as a high percentage of the network traffic concerns only a few processors at any time. Burdening all the other processors with broadcast of this information would impact them heavily. Some efficient mechanism for allocating and managing the Ethernet's multicast address space must be developed.

In Chapter 5, we estimated the expected difference in performance between the Caching Ring and an implementation of the Caching Ring coherence protocol on the Ethernet using a reliable broadcast protocol. This estimate is based solely on the transmission characteristics of the two communication networks. The error characteristics of the Ethernet and ring networks are also different. In particular, high traffic loads on the Ethernet tend to cause a higher rate of collisions. This results in more delay in acquisition of the network channel for sending packets, and more errors once the packets have been placed on the network. A complete simulation of the Ethernet-based Caching Ring that included collision and other error conditions would be interesting.

Cache issues. Several multiprocessors are under development with cache subsystems that allow multiple writers – the data for each write to a shared memory block is transmitted to each cache that has a copy of the block [4]. As a result, there is no **Invalid** state in the coherence protocol. The performance of a Caching Ring using this type of protocol, with various object repository update policies, would be of great interest. We expect that performance would be improved, because the all misses resulting from the invalidation of updated blocks would be eliminated from the system.

Schemes for more frequent updates of modified blocks from caches to the object repository are also of interest. The simple scheme of periodically flushing the modified blocks from the buffer can increase the miss ratio by as much as 35%, which can have a significant impact on

network and server load during periods of high file traffic. A mechanism that sensed network load and only flushed modified blocks when the network is idle would be useful.

An alternate architecture for the CRI would make the shared cache directory and block cache private to the CRI, and require the processor to request blocks through the CRI. The CRI could synchronize with the system by acquiring the ring token before responding to the processor, thus ensuring that the data received by the processor is not stale. We believe that this will increase the effective access time, perhaps appreciably in a large system with many stations and much network traffic. A simulation study of this CRI architecture would be instructive.

A more efficient crash recovery mechanism could be developed. The server is perfectly located to act as a recorder of all transactions that take place on the ring. A station that experiences a network failure but not a total crash can, on being reconnected to the network, query the server for all missed transactions. Powell describes such a mechanism, known as *Publishing*, in [43]. In this mechanism, all transactions are recorded on disk, that would add to the disk bottleneck already present at the server. We propose, instead, a set of messages that allow the recovering CRI to query the server about the status of all blocks that it holds in the cache. A straightforward design would have the recovering CRI flush all non-owned blocks, and determine which of the blocks it believes are owned are still owned. Perhaps a more efficient solution can be found, which allows the recovering CRI to flush only those blocks that are no longer valid copies.

6.5. Summary

In this thesis, we have explored the issues involved in caching shared objects in a distributed system. We designed a mechanism for managing a set of distributed caches that takes advantage of current hardware technology. The advent of powerful inexpensive microprocessors, memories, and network interfaces leads us to conclude that we can devote a large amount of computing power to the cache coherence algorithm for each workstation in a distributed system.

The Caching Ring combines a powerful hardware interface, software, and network protocols to efficiently manage a collection of shared objects in a distributed system. Our experiments with this design have extended the understanding of caching in a distributed system, and we propose the Caching Ring as an alternative to the less general solutions previously used in distributed systems.

References

[1] *UNIX Programmer's Manual, 4.2 Berkeley Software Distribution, Virtual VAX-11 Version* Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, 1983.

[2] O. P. Agrawal and A. V. Pohm. Cache Memory Systems for Multiprocessor Architectures. In *Proceedings AFIPS National Computer Conference*. Dallas, TX, June, 1977.

[3] Gregory R. Andrews and Fred B. Schneider. Concepts and Notations for Concurrent Programming. *ACM Computing Surveys* 15(1):3-43, March, 1983.

[4] James Archibald and Jean-Loup Baer. *An Evaluation of Cache Coherence Solutions in Shared-Bus Multiprocessors*. Technical Report 85-10-05, Department of Computer Science, University of Washington, Seattle, WA 98195, October, 1985.

[5] James Bell and David Casasent and C. Gordon Bell. An Investigation of Alternative Cache Organizations. *IEEE Transactions on Computers* C-23(4):346-351, April, 1974.

[6] Dennis M. Ritchie and Ken Thompson. The UNIX Time-Sharing System. *Communications of the ACM* 17(7):365-375, July, 1974. Revised and reprinted in the *Bell System Technical Journal*, (57)6:1905-1929.

[7] P. Calingaert. *Operating System Elements: A User Perspective.* Prentice-Hall, Englewood Cliffs, New Jersey, 1982.

[8] Warren Teitelman. *The Cedar Programming Environment: A Midterm Report and Examination*. Technical Report CSL-83-11, Xerox Palo Alto Research Center, June, 1984.

[9] Michael D. Schroeder and David K. Gifford and Roger M. Needham. *A Caching File System for A Programmer's Workstation*. Technical Report 6, Digital Systems Research Center, Palo Alto, CA, October, 1985.

[10] Jo-Mei Chang and N. F. Maxemchuk. Reliable Broadcast Protocols. *ACM Transactions on Computer Systems* 2(3):251-273, August, 1984.

[11] Douglas Comer. *Operating System Design, the XINU Approach.* Prentice-Hall, Englewood Cliffs, NJ, 1984.

[12] C. J. Conti and D. H. Gibson and S. H. Pitkowsky. Structural Aspects of the System/360 Model 85 (I) General Organization. *IBM Systems Journal* 7(1):2-14, January, 1968.

[13] Philip A. Bernstein and Nathan Goodman. Concurrency Control in Distributed Database Systems. *Computing Surveys* 13(2):185-221, June, 1981.

[14] Edsger W. Dijkstra. Cooperating Sequential Processes. In F. Genuys (editor), *Programming Languages*. Academic Press, New York, 1968.

[15] Edsger W. Dijkstra. The Structure of the 'THE' Multiprogramming System. *Communications of the ACM* 11(5):341-346, May, 1968.

[16] Paul J. Leach and Paul H. Levine and James A. Hamilton and Bernard L. Stumpf. The File System of an Integrated Local Network. In *Proceedings 1985 ACM Computer Science Conference*, pages 309-324. March, 12-14, 1985.

[17] J. B. Dennis and E. C. Van Horn. Programming Semantics for Multiprogrammed Computations. *Communications of the ACM* 9(3):143-155, March, 1966.

[18] David J. Farber and K. Larson. The Structure of a Distributed Computer System - the Communication System. In Jerome Fox (editor), *Proceedings of the Symposium on Computer Communications Networks and Teletraffic*, pages 21-27. Polytechnic Press, New York, April, 1972.

[19] W. D. Farmer and E. E. Newhall. An Experimental Distributed Switching System to Handle Bursty Computer Traffic. In *Proceedings of the ACM Symposium on Problems in Optimization of Data Communication Systems*, pages 31-34. Pine Mountain, GA, October, 1969.

[20] J. Feder. The Evolution of UNIX System Performance. *Bell Laboratories Technical Journal* 63(8):1791-1814, October, 1984.

[21] Rick Floyd. *Short-Term File Reference Patterns in a UNIX environment*. Technical Report 177, Computer Science Department, The University of Rochester, Rochester, NY 14627, March, 1986.

[22] John Fotheringham. Automatic Use of a Backing Store. *Communications of the ACM* 4:435-436, 1961.

[23] David R. Fuchs and Donald E. Knuth. Optimal Prepaging and Font Caching. *ACM Transactions on Programming Languages and Systems* 7(1):62-79, January, 1985.

[24] J. Goodman. Using Cache Memories to Reduce Processor-Memory Traffic. In *10th Annual Symposium on Computer Architecture*. Trondheim, Norway, June, 1983.

[25] A. N. Haberman. *Introduction to Operating System Design.* Science Research Associates, Palo Alto, California, 1976.

[26] H. Schwetman. *CSIM: a C-based, process-oriented simulation language*. Technical Report PP-080-85, MCC, September, 1985.

[27] R. C. Holt and E. D. Lazowska and G. S. Graham and M. A. Scott. *Structured Concurrent Programming with Operating Systems Applications.* Addison-Wesley, 1978.

[28] M. Satyanarayanan and John H. Howard and David A. Nichols and Robert N. Sidebotham and Alfred Z. Spector and Michael J. West. The ITC Distributed File System: Principles and Design. *Operating Systems Review: Proceedings of the Tenth ACM Symposium on Operating Systems Principles* 19(5):35-50, December, 1985.

[29] Anita K. Jones. The Object Model: A Conceptual Tool for Structuring Software. In R. Bayer and R. M. Graham and G. Seegmuller (editor), *Lecture Notes in Computer Science*: *Operating Systems*, pages 7-16. Springer-Verlag, New York, 1978.

[30] Robert M. Keller and M. Ronan Sleep. Applicative Caching. *Transactions on Programming Languages and Systems* 8(1):88-108, January, 1986.

[31] T. Kilburn and D. B. G. Edwards and M. J. Lanigan and F. H. Sumner. One-level Storage System. *IRE Transactions EC-11* 2:223-235, April, 1962.

[32] Edward D. Lazowska and John Zahorjan and David R. Cheriton and Willy Zwaenepoel. *File Access Performance of Diskless Workstations*. Technical Report 84-06-01, Department of Computer Science, University of Washington, June, 1984.

[33] Kai Li and Paul Hudak. *Memory Coherence in Shared Virtual Memory Systems*. Technical Report, Yale University, Department of Computer Science, Box 2158 Yale Station, New Have, CT 06520, 1986.

[34] J. S. Liptay. Structural Aspects of the System/360 Model 85 (II) The Cache. *IBM Systems Journal* 7(1):15-21, January, 1968.

[35] Gene McDaniel. An Analysis of a Mesa Instruction Set Using Dynamic Instruction Frequencies. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 167-176. Association for Computing Machinery, SIGARCH, March, 1982.

[36] M. Kirk McKusick and Mike Karels and Sam Leffler. Performance Improvements and Functional Enhancements in 4.3BSD. In *Proceedings of the Summer 1985 Usenix Coference*, pages 519-531. 1985.

[37] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *Transactions on Computer Systems* 2(1):39-59, Feb, 1984.

[38] John K. Ousterhout and Hervé Da Costa and David Harrison and John A. Kunze and Mike Kupfer and James G. Thompson. A Trace-Driven Analysis of the UNIX 4.2BSD File System. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 15-24. 1985.

[39] Douglas Comer and Larry L. Peterson. *A Name Resolution Model for Distributed Systems*. Technical Report CSD-TR-491, Department of Computer Sciences, Purdue University, West Lafayette, Indiana, February, 1985.

[40] J. Peterson and A. Silberschatz. *Operating System Concepts.* Addison-Wesley, Reading, Massachussetts, 1983.

[41] Peter J. Denning. Virtual Memory. *Computing Surveys* 2(3):1-2, September, 1970.

[42] Peter J. Denning. Working Sets Past and Present. *IEEE Transactions on Software Engineering* , January, 1980.

[43] Michael L. Powell and David L. Presotto. Publishing: A Reliable Broadcast Communication Mechanism. *Operating Systems Review: Proceedings of the Ninth ACM Symposium on Operating Systems Principles* 17(5):100-109, 10-13 Oct 1983.

[44] Lucien M. Censier and Paul Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers* C-27(12):1112-1118, December, 1978.

[45] Jerome H. Saltzer and Kenneth T. Pogran. A Star-Shaped Ring Network with High Maintainability. In *Proceedings of the Local Area Communications Network Symposium*, pages 179-190. Boston, May, 1979.

[46] Mahadev Satyanarayanan. A Study of File Sizes and Functional Lifetimes. In *Proceedings of the 8th Symposium on Operating Systems Principles*, pages 96-108. Asilomar, CA, December, 1981.

[47] J. F. Shoch and J. A. Hupp. Measured performance of an Ethernet local network. *Communications of the ACM* 23(12):711-721, December, 1980.

[48] Microsystems, Inc., Sun. ND(4P) - Network Disk Driver. In *System Interface Manual for the Sun Workstation*. January, 1984.

[49] Dan Walsh and Bob Lyon and Gary Sager. Overview of the Sun Network File System. In *Proceedings of the Winter 1985 Usenix Conference*, pages 117-124. Portland, OR, January, 1985.

[50] Alan Jay Smith. Cache Memories. *Computing Surveys* 14(3):473-530, September, 1982.

[51] Alan Jay Smith. Disk Cache - Miss Ratio Analysis and Design Considerations. *ACM Transactions on Computer Systems* 3(3):161-203, August, 1985.

[52] Alan J. Smith. Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms. *IEEE Transactions on Software Engineering* SE-7(4):403-417, July, 1981.

[53] Edward P. Stritter. *File Migration*. PhD thesis, Stanford Linear Accelerator Center, Stanford University, January, 1977.

[54] Steven J. Frank. Tightly Coupled Multiprocessor System Speeds Memory-Access Times. *Electronics* :164-169, January, 12, 1984.

[55] Andrew S. Tanenbaum. Network Protocols. *Computing Surveys* 13(4):453-489, December, 1981.

[56] C. K. Tang. Cache System Design in the Tightly Coupled Multiprocessor System. In *Proceedings National Computer Conference*, pages 749-753. October, 1976.

[57] Jon Postel, ed. *Transmission Control Protocol*. Request for Comments 793, Network Information Center, SRI International, September, 1981.

[58] Ken Thompson. UNIX Implementation. *The Bell System Technical Journal* 57(6):1931-1946, July-August, 1978.

[59] J. Von Neumann and A. W. Burks and H. Goldstine. Preliminary Discussion of the Logical Design of an Electronic Computing Instrument. *Collected Works* V, 1963.

[60] Cheryl A. Wiecek. A Case Study of VAX-11 Instruction Set Usage for Compiler Execution. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 177-184. Association for Computing Machinery, SIGARCH, March, 1982.

[61] *The Ethernet: A Local Area Network. Data Link Layer and Physical Layer Specifications, Version 1.0* Xerox Corporation, 1980.

Acknowledgements

I gratefully acknowledge the support and help of my advisor, Doug Comer; his advice, criticism, and insights have consistently been on the mark. Most importantly, he steered me back to science when I was tempted to stray off into hacking. I would also like to thank the other members of my thesis committee: Tim Korb, for exposing me more fully to the non-UNIX world and coaching me in formal prose; and Tom Murtagh and Bharat Bhargava, for their suggestions and ability to view the problems and solutions presented here from a more detached viewpoint.

It is not possible for me to include all of the special people whose lives have crossed my path during my time at Purdue. I would, however, like to make special mention and give special thanks to: Forest Baskett for asking the question that got me started thinking about caching in distributed systems, and for acting as an unofficial fifth member of my committee; Bob Brown for the late nights of hacking, Defender, and the Doctor; my father, Theodore Kent, né Bozhidar Kantarjiev, for his devotion and encouragement, and for providing a vision for my life; my mother, Auguste Kent, for her unflagging support and understanding in both good times and bad; Marny Livingston for her help during qualifiers; Brian Reid for Scribe wizardry; Mark Shoemaker for all the support, insanity, and special times, especially the drives to Chicago.

In addition, the following people made invaluable contributions to my time at Purdue, without which I probably would not have survived: Paul Albitz, Wilma Allee, Roger Armstrong, Bruce Barnett, Laura Breeden, Georgia Connaroe, Kevin Cullen, Jane Douglas, Ralph Droms, Werner Erhard, Paul de Haas, Julie Hanover, Steve Holmes, Mark Kamen, Balachander Krishnamurthy, Cathy La Luz, Linda McCord, Wendy Nather, Francie Newbery, Paige Pell, Paula Perkins, Gary S. Peterson, John Riedl, Katherine Rives, Sandy Robbins, Harry Rosenberg, Dave Schrader, Art Schuller, Steve Shine, Malcolm Slaney, Herb Tanzer, Dan Trinkle, the members of the ‘‘systems group’’, the Arni’s crowd, the staff at the East Coast 6-Day Advanced Course, and many others to whom I apologize for the lack of space and my momentary lapse of memory. Each contributed something tangible, valuable, and irreplaceable to my path through this life and my completion of this work.

Herb Schwetman of the Microelectronics Computer Technology Corporation of Austin, Texas graciously provided me with a copy of the CSIM simulation package, which greatly simplified the production of the simulation results reported in Chapter 5.

And finally, I must acknowledge and thank Christy Bean Kent, my partner in this and all things, for her patience and understanding, and for continually reminding and showing me that I had what I needed to see this monumental task through.

Table of Contents

List of Figures

List of Tables