HINT-BASED COOPERATIVE CACHING

by

Prasenjit Sarkar

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF COMPUTER SCIENCE

In Partial Fulfillment of the Requirements

For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

1 9 9 8

THE UNIVERSITY OF ARIZONA ®
GRADUATE COLLEGE

As members of the Final Examination Committee, we certify that we have read the dissertation prepared by Prasenjit Sarkar

entitled Hint-based Cooperative Caching

and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy

[illegible]	6/24/98
John Hartman	Date
Larry Peterson	6/17/98
Larry Peterson	Date
[illegible]	6/17/98
Rick Schlichting	Date
	Date
	Date

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copy of the dissertation to the Graduate College.

I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

[illegible] — 6/24/98

Dissertation Director John Hartman — Date

STATEMENT BY AUTHOR

SIGNED: [illegible]

[illegible]

ACKNOWLEDGMENTS

I would like to thank my advisor, John Hartman, for the advice, support and understanding he has given during my graduate studies. His insight into the various subtleties of system design and his meticulous performance analysis has had a profound impact on me and was invaluable for the completion of my dissertation. My thanks go to Mary Bailey for the insight, understanding and support provided during the initial stages of my graduate career.

I would also like to thank the members of my dissertation committee, Larry Peterson and Rick Schlichting, for reading the drafts of this dissertation and providing valuable feedback that improved its content and presentation. I would also like to thank my minor advisor, Sarma Vrudhula, for his guidance in my field of minor studies in ECE.

My sincere thanks also go to Phil Kaslo for providing invaluable assistance and support in installing the cooperative caching file system prototype in the department. I would also like to thank Wanda Chiu and Rajesh Sundaram for volunteering their workstations during the early development phases of this prototype, a very sincere gesture on their part. Without their active support, it is doubtful whether the dissertation would be complete at this time.

I would like to thank Mike Dahlin for providing the N-chance simulator and the Sprite trace information, and Mike Feeley for clarifications on the GMS system. I would also like to thank Olaf Kirch for clarifications on the NFS client implementation in Linux. Wanda Chiu, Matti Hiltunen, Mudita Jain, Dave Lowenthal, Anup Kuzhiyil, Tammo Spalink, Rajesh Sundaram, and Todd Proebsting all provided much-needed comments on this research. Finally, we thank Andy Bavier, Patrick Bridges, Ian Murdock, Larry Peterson and Oliver Spatscheck for participating in the measurements of the prototype.

Last, but not the least, I would like to thank my fiancee Shailaja and my parents, Subir and Archana, for their support and encouragement in making this dissertation possible.

TABLE OF CONTENTS

TABLE OF CONTENTS - *Continued*

TABLE OF CONTENTS - *Continued*

TABLE OF CONTENTS - *Continued*

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

This dissertation focuses on caching in distributed file systems, where the performance is constrained by expensive server accesses. This has led to the evolution of cooperative caching, an innovative technique which effectively utilizes the client memories in a distributed file system to reduce the impact of server accesses. This is achieved by adding another layer to the storage hierarchy called the *cooperative cache*, allowing clients to access and store file blocks in the caches of other clients.

The major contribution of this dissertation is to show that a cooperative caching system that relies on local hints to manage the cooperative cache performs better than a more tightly coordinated fact-based system. To evaluate the performance of hint-based cooperative caching, trace-driven simulations are used to show that the hit ratios to the different layers of the storage hierarchy are as good as those of the existing tightly-coordinated algorithms, but with significantly reduced overhead. Following this, a prototype was implemented on a cluster of Linux machines, where the use of hints reduced the average block access time to almost half that of NFS, and incurred minimal overhead.

CHAPTER 1
INTRODUCTION

1.1 Distributed File Systems and Caching

Technology trends of the past two decades has led to the evolution of distributed systems as a key computing paradigm. Low-cost microcomputers now provide the computing power for a vast range of applications and have replaced the need for a centralized mainframe-based computing environment in organizations. In tandem, local area networking has also evolved to provide high-bandwidth connections between individual computers[Forum93, ANSI87, Metcalfe76]. This has led to the development of distributed systems where many computers share resources and services over a network.

While distributed systems provide advantages over a collection of isolated microcomputers, they also introduce new challenging issues[Tannenbaum96]. For example, while distributed systems allow incremental growth by simply adding a new computer to the network, the presence of this computer can saturate the services provided by the distributed system and lead to poor performance.

A key service provided by a distributed system is that of files. The file service is typically provided by a process or a thread pool which runs on a machine known as the *server*. A distributed system may have one or several servers which serve a set of files to the other machines in the system. The remaining machines, or *clients*, use a predetermined protocol such as NFS[Sandberg85] to access the file service provided by the servers.

The principal challenge in designing a distributed file system is to achieve high performance with minimal overhead. Clients should be able to access their files as fast as possible and the average time to access a file block, or *average block access time*, should be minimized. At the same time, the work done by the distributed system for providing file service, or *overhead*, should be kept low, as high overheads can swamp the network and disrupt client activity.

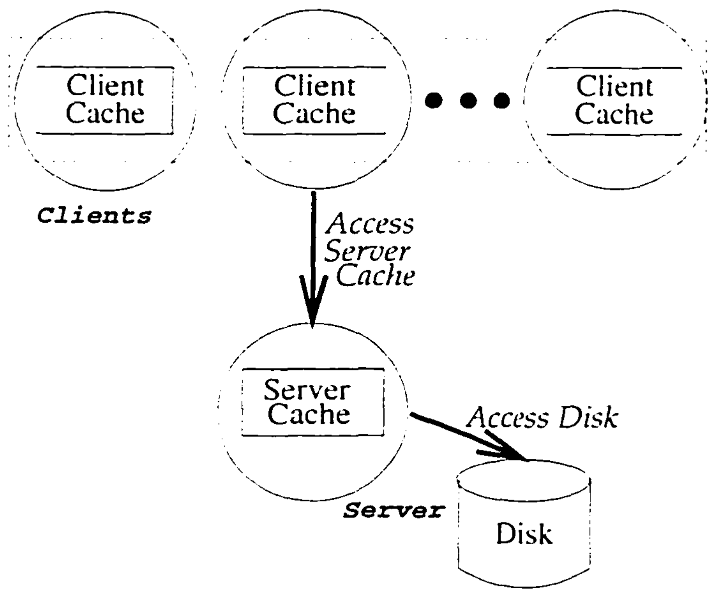


Figure 1.1: **Caching in Distributed File Systems.** This figure shows the storage hierarchy in distributed file systems. Whenever a client needs to access a block, the client first looks in the *client cache* in its local memory or disk. For the purpose of this dissertation, we assume that the client cache is in memory because file access to memory is several times faster than that to disk. If the lookup fails, the client forwards the block request to the server. The server performs a lookup in the *server cache* and in case of failure, forwards the block request to its *disk* subsystem.

Caching is one of the principal mechanisms of achieving this goal[Smith82, Leach83, Sandberg85, Popek85, Schroeder85, Bach87, Howard88, Nelson93]. The use of caches creates a storage hierarchy on top of the file service to filter out accesses to the slower layers of the hierarchy. As seen in Figure 1.1, the topmost layer in the hierarchy is the *client cache* which resides in the local memory or disk of each client. For the purpose of this dissertation, we assume that the client cache is in memory because file access to memory is several times faster than that to disk. A client's file access that hits in its cache avoids the cost of communicating over the network to a server. Otherwise, the client forwards the request to the appropriate server. The server then checks for the block in the *server cache*, where a hit prevents an expensive *disk* access. A simulation using the traces of the BSD UNIX operating system[Ousterhout85, Nelson93] revealed that client caches can greatly reduce server accesses.

1.2 The Server Bottleneck

Although client caches are effective at filtering out server accesses, the performance of a distributed file system is often limited by the poor hit rate on the server cache[Dahlin94]. The server cache's poor hit rate increases the number of disk accesses, which are an order of magnitude slower than server cache accesses. As a result, the average time to access a file block increases due to expensive disk accesses, limiting performance even though the hit rate to the local client cache might be as high as 80%[Dahlin94]. Past studies have also shown that despite client caching, the server is the principal bottleneck for distributed file system performance[Lazowska86, Satyanarayanan85]. A recent study points out that such a phenomenon has been observed even when the server is lightly loaded[Riedel96].

There are two principal reasons for this poor hit rate on the server cache: *capacity* and *locality*. First, as the number of clients increases, the number of accesses to the server increases proportionately. However, if the server cache is fixed in size, it will not have the capacity to filter out the increasing number of server accesses. Second, hits to the local client caches take advantage of locality in file references. Consequently, server accesses have very poor locality.

One possible way to improve the server cache's poor hit rate is to add more memory to the server. However, studies have shown that not only is it much less expensive to distribute this memory over all the clients, but the average block access time improves by 33% if the added memory is moved from the server to the clients[Dahlin94]. This is mainly because the amount of memory needed to offset the locality and capacity constraints in a server cache is prohibitively large and requires expensive DRAM technology.

1.3 Technology Trends

The performance problems in a distributed file system are rooted in the disparity in performance between the different layers of the storage hierarchy: disk accesses are an order of magnitude slower than server cache accesses. Trends in the development of processors, networks and storage can indicate whether or not this problem will get worse and point to possible solutions.

The rates of improvement in network, memory and disk performance are a reasonable indicator of future trends in the relative performance between networks, memory and storage. However, it is difficult to predict improvements because of uncertainty in the adoption of new standards as well as the effect of business cycles. The interesting statistic is the growth in performance of networks and memory compared to disks. While network performance grew by 25-45% annually in the past three years primarily due to the introduction of Fast Ethernet, it is expected to grow even faster as Gigabit Ethernet and Fibre Channel start becoming more popular[Peterson96]. A similar trend can be seen in memory speeds and bandwidth[Henessey96]. In contrast, disk performance rose at 10-20% annually from 16 ms in 1991 to 8 ms today[Dahlin95]. Furthermore, storage research is focused more on capacity rather than on latency[Henessey96]. To put this into perspective with the problems in distributed file systems, disk accesses are going to be even more expensive relative to server and client cache accesses and will increasingly dominate the average block access time. Therefore any proposed solution to the server bottleneck in distributed file systems must reduce server accesses.

1.4 Cooperative Caching

The goal of this research is to improve the performance of distributed file systems by reducing the impact of the server cache's poor hit rate. This is achieved by adding another layer to the storage hierarchy, positioned between the local client cache and the server. This layer resides in the memories of all clients and is known as the *cooperative cache*. The technique of *cooperative caching* allows clients to access and store file blocks in the caches of other clients, so that local cache misses can be satisfied without accessing the server. If the hit rate to the remote client caches is high, the number of server accesses will be reduced and performance improved.

Incorporating the cooperative cache in a storage hierarchy is challenging because the issues concerning cooperative caching go beyond those for the other layers in the storage hierarchy:

- *Coordination.* The cooperative cache is distributed among multiple clients, which implies that cooperative caching operations could require coordination between these clients.
- *Overhead.* The overhead of coordinating the cooperative cache should be minimized so as not to saturate the network or disrupt client activity.
- *Resource Sharing.* As the cooperative cache and local client caches share the same physical memory in the clients, there should be a resource-sharing mechanism to ensure that the performance of the local cache is not be affected by the cooperative cache.

1.5 Contributions

The major contribution of this dissertation is to show that the use of imprecise local information (*hints*) to manage the cooperative cache performs better than a more tightly-coordinated system with precise global state (*facts*). The intuition is simple: hints are less expensive to maintain than facts, and as long as hints are highly accurate, they will improve performance. However, inaccurate hints increase overhead and degrade performance, negating the benefits of a hint-based approach. Thus the key challenge in designing a hint-based system is to ensure that the hints are highly accurate.

This dissertation describes a cooperative caching system that uses hints instead of facts whenever possible:

- Hints are used to locate blocks in the client caches, and maintained to achieve both high accuracy and low overhead.
- Hints are also used to replace accurately the least valuable blocks in the client caches without incurring high overhead.
- The server memory is used in an innovative way to augment the use of hints.

To evaluate the performance of hint-based cooperative caching, trace-driven simulations are used to compare the hint-based algorithm with existing and ideal algorithms. Then, a prototype file system was implemented using hint-based cooperative caching on a cluster of Linux machines, and the prototype's performance was measured with actual user activity over one week. The results distinguished the hint-based algorithm from its tightly-coordinated fact-based peers:

- Simulations show that the average block access times of the hint-based algorithm are the same as those of the existing and ideal algorithms.
- Simulations also reveal that the manager load in the hint-based algorithm is lower by as much as 30 times when compared to that in the existing algorithms.
- Measurements of the prototype showed that the average block access time was almost half that of NFS.
- Measurements also indicated that the average and maximum overhead rate of cooperative caching was negligible ($< 7.5\%$) compared to the available bandwidth of a 10 Mbps Ethernet network.
- Finally, hint accuracy was high ($> 98\%$) both in the simulations and the prototype.

1.6 Overview of the Dissertation

Cooperative caching is a relatively new concept and differs from traditional forms of caching. The dissertation begins with an introduction to the components of cooperative caching, the issues unique to cooperative caching and a discussion of the available approaches. To evaluate the hint-based algorithm, existing fact-based and ideal algorithms are elaborated next. Then the hint-based algorithm is described along with a detailed discussion of how hints are used in cooperative caching. To evaluate the algorithm, simulations compare its performance and overhead with existing and ideal algorithms. This is followed by the details of the prototype implementation as well as the subsequent measurements of the prototype. The dissertation ends with conclusions on hint-based cooperative caching.

CHAPTER 2
COOPERATIVE CACHING

This chapter deals with cooperative caching in greater detail. The first section defines the principal components of cooperative caching. Following this, the issues specific to cooperative caching are discussed along with a summary of possible approaches to each issue.

2.1 Definitions

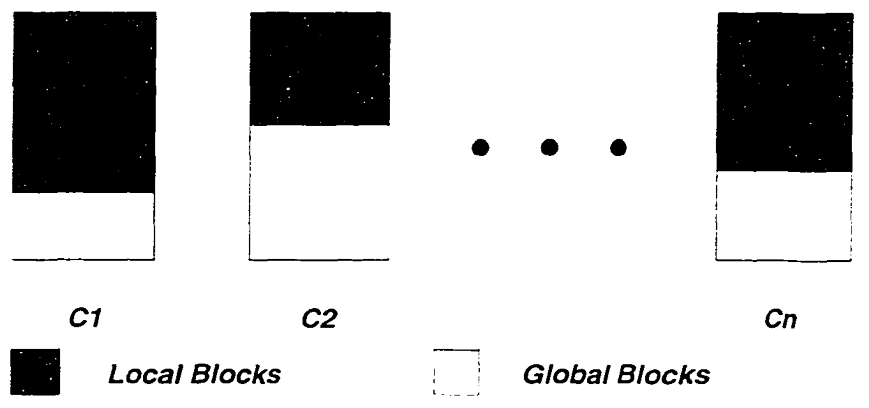


Figure 2.1: **The Cooperative Cache in Client Memories.** The cooperative cache is the layer of the storage hierarchy positioned between the local client caches and the server. The introduction of a new layer in the storage hierarchy introduces two types of blocks in a client's cache: (i) *local* blocks, which are those being accessed by the client itself; (ii) *global* blocks, which are stored in the client's memory by other clients. As seen above, the local and global blocks share the client memories in $C1..Cn$ and the ratio of these two types of blocks varies from client to client. For example, the fraction of local blocks in the cache of an active client like $C1$ is larger than that in an idle client like $C2$.

Dahlin defines cooperative caching as a technique to "improve file system performance and scalability by coordinating the contents of client caches and allowing requests not satisfied by a client's local in-memory cache to be satisfied by the cache of another client"[Dahlin95].

The cooperative cache is the layer of the storage hierarchy positioned between the client cache and the server. Cooperative caching is a technique that allows a client to access the caches of other clients when the client misses a block in its cache. Thus, cooperative caching allows clients to use effectively the entirety of the client memories in the distributed file system. Cooperative caching is equivalent in concept to the Global Memory Service[Feeley95], though it is specific to distributed file systems.

The introduction of a new layer in the storage hierarchy introduces two types of blocks in the client caches. The first type of blocks in a client's cache are the *local* blocks, which are those being accessed by the client itself. The remaining blocks in a client's cache are stored in the client's memory by other clients and are referred to as *global* blocks. Thus, a local block in a client's cache becomes a global block when the block is forwarded to the cooperative cache on another client. Similarly, a global block in a client's cache becomes a local block if the client starts accessing the block. The ratio of local and global blocks in a client's cache is determined by the activity level of a client. The caches of active clients have a large fraction of local cache blocks because the clients access most of their memory. In contrast, the caches of idle clients are underused and therefore contain a larger fraction of global cache blocks.

2.2 Components

Cooperative caching involves three logical entities: *clients*, *servers*, and *managers*. As described in Chapter 1.1, clients access file blocks stored on the servers. Servers do not maintain the state of the distributed file system: this role is reserved for managers. Thus, clients do not access the server to access the state of the distributed file system, reducing the server load as a result. However, this also means that servers are not aware of the contents of the client caches.

The presence of managers is necessitated by the distributed nature of the cooperative cache. The role of managers in cooperative caching is to maintain distributed file system state: clients access this state to locate blocks in and replace blocks from the cooperative cache. The role of managers varies among cooperative caching algorithms and serves as a major distinction between them. For example, a manager in N-chance[Dahlin94] has a very limited role in replacing blocks from the client caches compared to a manager in GMS[Feeley95]. There is no restriction on the number of managers in a cooperative caching system. A manager is likely to be an application daemon running on a machine in the distributed file system. A schematic of cooperative caching is shown in Figure 2.2.

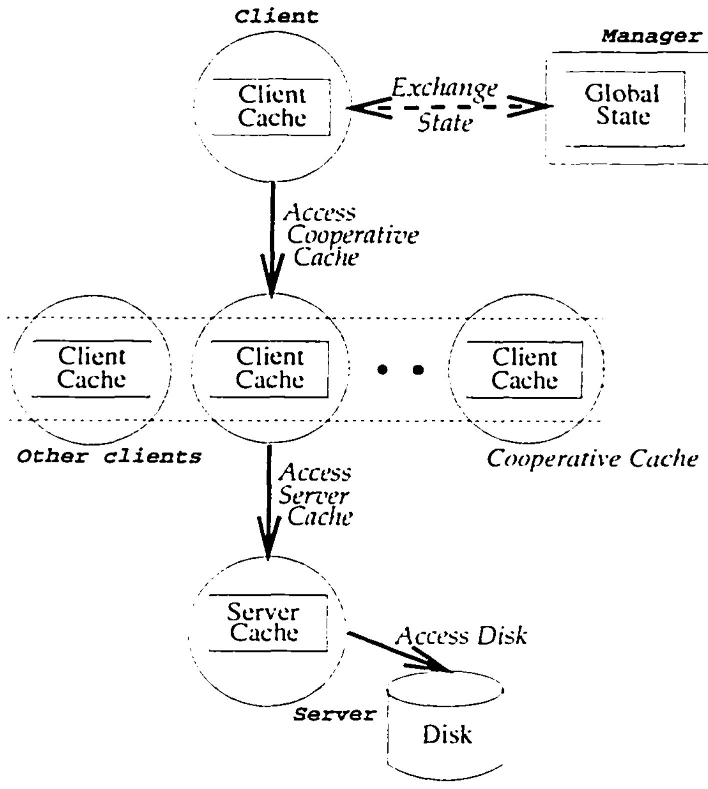


Figure 2.2: **Framework of Cooperative Caching.** This figure shows the framework of cooperative caching. Cooperative caching consists of *clients*, *managers* and *servers*. Clients access file blocks stored on the servers. Managers help coordinate the cooperative cache and maintain global state. Clients exchange state with the managers to update the global state. On a local miss on a file block, the clients use the state information to access the missing block from the caches of other clients. If the block is not found in the client caches, the client accesses the server.

2.3 Issues

The issues concerning the cooperative cache go beyond those for the other layers in the storage hierarchy because the cooperative cache is distributed over multiple clients:

- *Coordination.* The distributed nature of the cooperative cache means that the state of the cooperative cache is not confined to a single client and any operation on the cooperative cache could potentially require coordination among clients. The *local state* of a client in cooperative caching refers to the state of the blocks in the client's cache. An operation on the cooperative cache is said to be *globally optimal* if the operation takes the best possible decision based on the local states of all the clients in the distributed file system. Therefore, a client cannot invoke an operation on the cooperative cache purely based on its own local state and expect the operation to approximate the globally optimal. In fact, various studies have shown that cooperative cache operations based solely on local state can seriously hurt performance[Dahlin94, Feeley95, Sarkar96], emphasizing the need for coordination among clients.

- *Overhead.* The performance of a cooperative caching system depends on the level of coordination required to manage the cooperative cache. Coordinating the cooperative cache incurs the overhead of exchanging information between clients and managers. It is important to minimize this overhead as otherwise it may negate the benefits of cooperative caching. This underlines the importance of *efficient* cooperative caching by ensuring high hit ratios to the cooperative cache without incurring significant overhead. Feeley et al. also outline the need for efficiency in cooperative caching[Feeley96].

- *Resource Sharing.* The local and global blocks shares the client memories, implying the need for a resource sharing protocol. The ratio of local and global cache blocks in a client's cache must be maintained such that the cooperative cache does not affect performance adversely. For example, too large a cooperative cache can tend to reduce the hit ratios to the local caches of clients, which is the fastest layer in the storage hierarchy.

These issues arise when one considers the *management operations* for the cooperative cache. Each layer in a memory or storage hierarchy must support management operations for resource allocation and deallocation. These management operations also move data between the layers of the hierarchy for reasons of performance or consistency. In file systems, data is usually transferred in fixed size segments called *blocks*, though there are exceptions as in AFS where the unit of transfer is files[Howard88].

2.4 Cache Access

2.4.1 Access Policy

The first operation in caching is the *access policy* which decides when and how a layer fetches blocks from a lower layer in a hierarchy. The first part of an access policy is to decide when to fetch the block from the lower layer. A common method is *demand paging* in which a fetch is initiated from the lower layer when a client wants to access a block which is not present in its cache. The disadvantage of this method is that it incurs the full latency of retrieving a block from the lower layer. This problem is solved by adding *prefetching*, which brings blocks into a caching layer in anticipation of an access in the near future.

Prefetching introduces two issues into cache management. First, prefetching is speculative as it must be able to predict future accesses with enough precision to improve performance. Second, prefetching must integrate well with replacement policies so that prefetched blocks do not replace valuable blocks from the cache. These two issues have been well studied in literature and practice[McKusick84, Kotz91, Griffioen94, Patterson95, Cao95]. Current prefetching techniques do not take cooperative caching into account, though researchers are beginning to address this issue[Voelker98].

2.4.2 Block Lookup

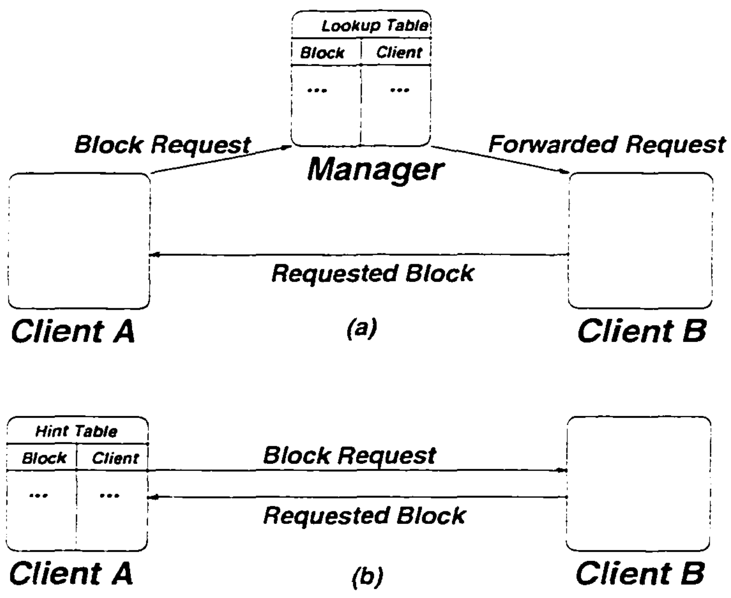


Figure 2.3: **Locating Blocks in the Client Caches.** This figure shows two possible approaches for locating blocks in the client caches. Part (a) shows the *manager-centric* approach, where a client contacts a manager on a local cache miss. The manager consults its lookup table and forwards the block request to the client caching the block. This client provides the block to the requesting client. Part (b) shows the hint-based approach, where on a local cache miss, a client consults its hint table and directly contacts the client caching the block. This client provides the block to the requesting client, avoiding the overhead of contacting the manager. Of course, the success of the hint-based approach depends on the accuracy of hints.

The second part of the access policy requires locating a missing block from the lower levels of the storage hierarchy. In traditional forms of caching, locating and retrieving a block is trivial as the block is confined to the server. However, this is not the case in cooperative caching as the block could be anywhere in the client caches and a lookup decision would require information about the contents of the client caches.

Existing block lookup strategies use a collection of managers to keep track of which blocks are present in the client caches, as shown in Figure 2.3(a). To ensure that managers have up-to-date block location information, all block movement in and out of the client caches must be reported to the managers. Each manager is responsible for managing the location information for a distinct subset of the file system's blocks. A client performs a block lookup by sending a request to the manager for the block. The manager forwards this block request to a client caching the block, if one exists, or to the server otherwise. Following this, the block is provided to the client from the appropriate source.

The advantage of this strategy is that a block lookup is accurate and always returns the correct location of the block in the client caches. The disadvantage is the high overhead which slows down cache accesses. Not only must clients contact a manager on every block lookup, the movement of all blocks in and out of the client caches must be reported to the managers. This strategy also greatly increases the manager load when the working set size of client applications starts to approach the size of the client memories. Under such a condition, blocks move rapidly in and out of the client caches, resulting in frequent communication with the manager and greatly increasing manager load.

The alternative strategy, seen in Figure 2.3(b), trades off accuracy for minimizing overhead. The clients maintain their own block location hints and do their own block lookup, avoiding the overhead of contacting a manager. However, if a block location hint is incorrect, then the client will have to perform extra accesses in order to retrieve the block from the storage hierarchy, negating the advantage of not contacting the manager. The key to making this strategy work is to maintain highly accurate block location hints and to provide a mechanism to deal with inaccurate hints.

2.4.3 Placement

A secondary management policy related to block lookup is *placement*, or where to place the fetched block in a layer. This is not an issue in the local client cache, as fetched blocks are put in random-access memories where placement is immaterial. However, the placement problem is a matter of study in those layers of a hierarchy where access to data is not uniform within a layer. For example, if the clients in a distributed file system were to be placed in a non-uniform network where there was a significant variation in average block access time depending on the location of a client, then placement of blocks in the client caches would become an issue. However for the purpose of this dissertation, the average block access times of the clients in a distributed file system are assumed to be uniform with little variation.

2.5 Replacement Policy

2.5.1 Local Cache

Another important management operation on the cooperative cache is that of *replacement*, which determines the order in which blocks are removed from the client caches. The replacement policy decides which block to discard from a layer to make room for a block fetched into the layer. A replacement policy is invoked when a block is fetched into a layer that is full.

The principal goal of a replacement policy is to minimize the average block access time. This means that an optimal replacement policy must keep blocks that are accessed soon and remove those that are accessed in the distant future[Belady66]. More precisely, the goal of an optimal replacement policy is to replace the block that is accessed the most distant in the future. However, in a practical system, it is not possible to know the future pattern of block accesses and hence any algorithm can at best try to approximate the optimal.

The LRU (least recently used) algorithm is the most common basis for practical replacement policies. The algorithm is based on the heuristic that blocks that are accessed furthest in the past are also those that are accessed the most distant in the future. This algorithm provides reasonable performance over a wide variety of applications. Less common algorithms include frequency counting algorithms such as LFU and MFU, which replace the least and most frequently used block respectively. However, these algorithms do not approximate the optimal very well[Silberschatz95].

Another practical consideration in designing a replacement policy is overhead. The overhead of replacing a block must not negate the benefits attained. Thus, while many algorithms reasonably approximate the optimal policy, their overheads are high and they are rarely implemented. For example, the LRU algorithm is very expensive to implement without the presence of hardware counters. As a result, practical algorithms tend to approximate the LRU algorithm itself. An example of this type of algorithm is Global Clock[Easton79], which organizes the blocks in a cache into a circular list. The algorithm uses a single reference bit per block and traverses through the list of blocks like a clock hand, inspecting a block at every turn. A block that has been referenced since the last inspection is made a candidate for replacement by clearing the reference bit for the block. A block that has not been referenced since the last inspection is evicted from the cache.

2.5.2 Cooperative Cache

Cooperative caching adds a new dimension to replacement policies because the cooperative cache is distributed over all the client machines. While practical algorithms like Global Clock are able to achieve a second order approximation of the optimal replacement policy, the distributed nature of the cooperative cache makes the problem of devising an algorithm with the same degree of approximation very challenging.

The cooperative cache replacement policy determines the order in which blocks are removed from the client caches. The replacement policy is activated when a client decides to replace a block from its cache. Thus, in contrast to traditional caching, multiple replacement decisions can go on in parallel in different clients. The implication is that the overhead of a replacement decision must be low as otherwise the network can be swamped with the overhead of concurrent replacement messages.

In general, a replacement policy tries to value a block using measures such as age(LRU) or frequency(LFU) with respect to other blocks in a caching layer when it decides whether or not to replace the block. In cooperative caching, a similar valuation is applied to the block being replaced from a client's cache. Cooperative caching algorithms usually do not distinguish between local and global cache blocks[1]. This allows a global block to replace a less valuable local block in an idle client.

[1]The GMS algorithm, which aims at cooperative caching for both virtual memory and distributed file systems, values local blocks more for considerations related to virtual memory paging[Feeley95].

increasing the fraction of global blocks in the client's cache. Similarly, a local block can replace a global block in an active client and increase the fraction of local blocks in the client's cache.

There are two choices when a client decides to replace a block from its cache. If the block is less valuable than any other block in the cooperative cache, then there is no reason to forward the block to the cooperative cache and the block is *discarded*. Otherwise, the block is forwarded to the cooperative cache and replaces another block in the client caches, which is then discarded.

The distributed nature of the cooperative cache adds another interesting factor to the valuation of a block. This factor in determining the value of a block is derived from the issue of resource sharing and is known as *duplicate avoidance*. The cooperative cache is a shared resource requiring coordination of which blocks are forwarded to the cooperative cache. Without any coordination, it is possible for clients to forward several copies of the same block to the cooperative cache. While duplicate local blocks in the client caches cannot be avoided as several clients might be accessing the same block, duplicate global blocks in the client caches reduce the effective sizes of both the local and cooperative caches and lower their hit rates. As a result, cooperative caching algorithms must avoid duplicate global blocks in the client caches.

Once the client decides to forward a block to the cooperative cache, the client must then decide the *target client*, or the client that will receive the block. The target client choice is important in determining the effectiveness of the replacement policy. Thus, the target client should be chosen such that the forwarded block replaces a less valuable block in the target client. Otherwise, if the replaced block in the target client is more valuable, then a future access to the replaced block will result in a local cache miss and in the worst case, a disk access.

2.5.3 Examples and Alternatives

There are many choices for a cooperative caching replacement policy. While an optimal replacement approach is not realizable in practice, LRU for the client caches is also impractical as it is expensive to determine the globally LRU block. Even Global Clock is also expensive to implement in the client caches as it would require extensive coordination to maintain a circular list of blocks distributed over the client caches. Clearly, a good approach is to approximate LRU, but keeping the overhead low at the same time.

Practical approaches to replace blocks from the client caches can broadly be generalized into two categories: centralized and distributed. The key difference is the role of the manager in making replacement decisions. In the centralized approach, a manager collects or maintains information about the contents of the client caches. The manager either distributes a summary of this information to all the clients or provides information on demand to the clients. The clients then use this information to replace blocks from the client caches. While this approach can yield results closely approximating that from the previously described LRU approach, it adds to the overhead of cooperative caching and is sensitive to manager failures.

In the distributed approach, clients make a replacement decision based on local replacement hints. This approach avoids the overhead of contacting the manager. However, the approach is sensitive to hint accuracy as there is a performance penalty if the hints are not accurate and a valuable block is replaced from the client caches.

2.6 Cache Consistency

Consistency is a management operation on a caching layer that is dictated by how multiple users can read and write the same file or block at the same time. One of the key decisions in a consistency protocol is to specify when modifications of a file or block by one user are observed by other users of the same file or block. For example, in the UNIX local file system, writes to an open file by a user are visible immediately to other users that have this file open at the same time.

2.6.1 Update Policy

The first key decision in a consistency protocol is the update policy, which determines how soon a write to a client cache is propagated to the server. Distributed file systems use a wide range of update policies. The simplest policy is *write-through*. where a client write to a cached file or block is immediately reflected to the server. The advantage of this scheme is its reliability. No file or block data is lost even if a client crashes because the server is always consistent with the contents of the client cache. The disadvantage is performance because synchronous writes impose a heavy penalty in terms of latency and overhead. An alternate policy called *delayed write* trades off reliability for performance. This policy updates the server only at specified intervals or times. lowering the overhead of each write and allowing overwrites of data at significantly less cost than the write-through scheme. The problem with delayed write is the reduced reliability as a client crash can lead to the loss of dirty data that has not yet been written through to the server.

The update policy in cooperative caching schemes is guided by the motivation to simplify the management of the cooperative cache. As a result. cooperative caching algorithms either use write-through or a restricted version of delayed write. where blocks are written out to the server whenever they are shared or forwarded to the cooperative cache. In either case. the goal is to make sure that all global blocks are clean. An important consequence of this policy is that if a manager enforces strong consistency. the server always has an up-to-date copy of the block should a cooperative cache lookup fail. On the other hand. if a more relaxed form of consistency is maintained (as in NFS). the copy in the server may be out-of date: however. with relaxed consistency. the block lookup mechanism also may fetch an out-of-date copy from the cooperative cache.

2.6.2 Consistency Mechanism

The consistency mechanism decides whether a locally cached copy of a file or block is older than the most up-to-date copy. If a file or block is determined to be stale. the client caching the file or block retrieves the most up-to-date copy from the appropriate source.

There are two principal approaches in maintaining consistency in a distributed file system[Silberschatz95]. The first approach is client-driven and puts the onus of detecting the validity of a file or block to the client caching the file or block. The client sends a message asking for the status of the file or block to the server. which replies in the positive only if the file or block is up-to-date. The performance of a client-driven scheme is determined by the frequency with which the client contacts the server to validate whether a file or block is up-to-date. A validation scheme that checks on every block access would be always consistent but the overhead would also be high. An alternative would be to check on every file open. lowering overhead but at the same time leaving open the possibility that files or blocks might become temporarily inconsistent. In general. the problem with a client-driven approach is that the overhead is high for any reasonable level of consistency[Howard88].

The second approach relies on a consistency manager to implement a *write-invalidate* policy. Whenever the manager is informed about an update to a file or block from a client. it informs the other clients caching the file or block to invalidate its contents. A subsequent access to the invalidated file or block causes a server access.

2.6.3 Granularity

The granularity of cache consistency is important to cooperative caching because it directly affects the level of coordination required for cooperative caching.

A *block-based* protocol maintains consistency on each individual block. If write-invalidate is used. clients communicate with the manager on every local cache miss to ensure consistency. With client-driven consistency. clients must frequently check with the manager to determine the validity of a locally cached block. As a result. this granularity of consistency is ideal for cooperative caching algorithms which use manager-centric lookup and replacement strategies. On the other hand. there is little point in using hints to avoid contacting the manager for block lookup and replacement.

A *file-based* protocol is similar to the block-based except that it maintains consistency on entire files rather than blocks, potentially allowing clients to handle local cache misses without contacting the manager. One drawback is that file-based consistency does not handle concurrent write-sharing of a file by multiple clients as efficiently as block-based consistency, but this pattern of file access is rare in distributed file systems[Baker91].

2.7 Server Caching

In a traditional distributed file system, the server maintains a cache of blocks that have been accessed by the clients. Although the server cache is lower in the storage hierarchy than the client caches and therefore has a lower hit rate, studies have shown that the server cache is still effective at reducing disk accesses and improving performance[Baker91].

The benefits of a server cache are less apparent with cooperative caching because local cache misses are first serviced by the caches of other clients. This reduces the server cache's effectiveness and raises the issue of using the server memory in a different manner so as to benefit cooperative caching.

One possibility is to use the server cache as part of the cooperative cache. The advantage of this is that it increases the effective size of the cooperative cache. As the hit ratio to the client caches is expected to be better than that to a traditional server cache, the increased use of the server memory would benefit cooperative caching. However, the disadvantage of this proposal is that the server would be target of forwards from clients, increasing its load and possibly hampering performance.

Yet another possibility is to augment the use of hints in cooperative caching. Hints can be inaccurate and it might be helpful to use the server memory to offset the effect of incorrect hints. These alternatives to traditional disk caching in server memory are evaluated using trace-driven simulation in Chapter 5.2.6.

2.8 Summary

The cooperative cache is the layer in the storage hierarchy positioned between the local client caches and allows a client to access and store file blocks in the caches of other clients. Cooperative caching is challenging because the cooperative cache is distributed over the client caches and must be coordinated with minimal overhead to reap performance benefits.

A block lookup mechanism in cooperative caching must be able to locate effectively a block in the client caches. One approach would be to use a manager to keep track of the location of blocks in the client caches. While this approach would be accurate, the overhead would be significantly lowered if clients maintained their own block location hints.

A cooperative caching replacement strategy must decide the order of replacement of blocks from the client caches. A manager-centric approach in valuating the blocks in the client caches works well, but lower overhead can be obtained if clients maintained hints about the values of blocks in the caches of other clients.

Two other design issues in cooperative caching are the granularity of the consistency mechanism and the possible uses of the server memory.

CHAPTER 3
EXISTING AND IDEAL ALGORITHMS

This chapter introduces existing and ideal algorithms against which the proposed hint-based algorithm is compared. The performance and overhead of existing algorithms provide insight into the strengths and weaknesses of a fact-based approach, as well as peer benchmarks for practical cooperative caching algorithms. On the other hand, the ideal algorithms provide a bound on achievable performance by a cooperative caching algorithm.

In this chapter, two existing cooperative caching algorithms are introduced - N-chance and GMS. The algorithms are discussed with respect to the management operations on the cooperative cache: block lookup, replacement and consistency. Following this, two ideal algorithms are discussed - Global LRU and Optimal - and the performance bounds established by each of these algorithms are shown.

The two existing algorithms developed from successive attempts to harness the remote memory in workstation clusters[Comer90, Felten91, Franklin92, Leff91]. These attempts established important results, which were subsequently used by cooperative caching algorithms. These results are elaborated on in Chapter 7.1.

3.1 N-chance

Dahlin et al. pioneered the use of cooperative caching by motivating the need to offset server accesses with those to the client caches[Dahlin94]. The authors inspected a wide variety of algorithms for using idle memory in a workstation cluster and settled on the N-chance algorithm for providing the best performance with the lowest overhead. The algorithm was further refined to separate the roles of the server and the manager in the *xfs* file system[Anderson95].

3.1.1 Lookup

The N-chance algorithm uses the server to take on the responsibilities of a manager and locate blocks in the client caches. When a client has a local cache miss on a block, the client sends a request to the server, which then forwards this request to a client caching the block. If the block is not present in the client caches, the server responds with the block itself. To maintain the locations of blocks in the client caches, the server is informed whenever a block moves in and out of the client caches. Thus, whenever a client discards a block or forwards it to another client, the client must inform the server. Similarly, when a client is forwarded a block from another client, the client too must inform the server. The only exception where a notification to the server is not necessary is when a client fetches a block from the server itself. A generalized schema describing manager-centric block location is shown in Figure 2.3(a).

However, the original N-chance algorithm does not reduce the number of messages to the server, adding to the server load. While the algorithm achieves near-optimal results with a high hit ratio to the cooperative cache, each remote cache access also involves a message to the server and adds to the server load. This means that the number of messages to the server will not be reduced even if clients have a high hit ratio to the cooperative cache.

The *xfs* file system refined the N-chance algorithm by separating the role of the manager from the server, so that the server and the manager reside on separate machines[Anderson95]. This refinement reduces the load on the server as communication with the manager can now avoid the server. Additionally, in an attempt to reduce the overhead of contacting the manager, the *xfs* file system also tries to co-locate managers with clients. The goal is to ensure that a client wanting to access a block is physically in the same machine as the manager for the block, and can thus communicate with the manager without incurring the overhead of a network message. The co-location policy used by *xfs* assigns the management of a file's blocks to the client which last wrote the file, the underlying assumption being that this client is most likely to access the file in the future.

3.1.2 Replacement

The N-chance algorithm uses a combination of duplicate avoidance and randomness to replace blocks from the client caches. Whenever a client replaces a block from its local caches and does not know whether the block is a *singlet*, or the only copy in the client caches, the client contacts the appropriate manager to check if the block is a singlet. If the manager responds in the affirmative, then the client forwards this block to a randomly chosen target client. In other words, the N-chance algorithm gives a preference to singlets, as discarding them may cause a future server access.

The N-chance algorithm also takes measures to ensure that these singlets are discarded from the client caches when they becomes less valuable. The N-chance algorithm imposes a limit n on the number of times such blocks can be forwarded from one client cache to another. The authors use measurements as well as empirical evidence to suggest that a value of 2 for n would provide optimal performance for this algorithm.

3.1.3 Consistency

The N-chance algorithm specifies a manager-centric block-based consistency scheme. Whenever a client updates a block, the client informs the appropriate manager which then invalidates the remaining copies of the block in the client caches (i.e. write-invalidate). The N-chance algorithm also assumes a write-through policy for updates, ensuring that only clean blocks are stored in the cooperative cache.

3.1.4 Discussion

The N-chance algorithm uses managers for block lookup, duplicate avoidance and consistency, and therefore incurs the overhead of frequent communication with the manager to maintain the exact global state. Moreover, while trying to co-locate a client accessing a file with the manager for the file's blocks can potentially reduce overhead, the disadvantages of such a scheme are many. First, it adds the complexity of locating the manager for a block because the manager may move from one machine to another. In other words, the client must do a manager lookup to locate a manager for a block. Second, the overhead of co-location is high if files are write-shared across multiple clients. A co-location mechanism would have to move the management of the shared blocks from one client to another, the overhead of which would negate any benefits obtained. All existing evaluations of co-location mechanisms have assumed a single client accessing blocks and have neglected to measure the overhead in the case where multiple clients access shared blocks[Feeley95, Anderson95]. A potential improvement to the above scheme is to maintain block location hints in every client, and thus avoid both the cost of communication with a manager and the complexity of a co-location mechanism.

For replacement, N-chance gives more weight to duplicate avoidance, and has poor performance if block access patterns do not conform to this assumption. For example, if a block is forwarded to a randomly selected client actively accessing its cache, then the block might replace a valuable local block. Moreover, if the same block is successively forwarded n times to such active clients, then the block is discarded from the client caches even though it may be a valuable cooperative cache block. Studies have shown this phenomenon frequently occurs in practice when the fraction of active clients is high and the idle memory is concentrated in the few remaining clients[Feeley95, Sarkar96].

3.2 GMS

The GMS or *Global Memory Service* algorithm is also motivated by the need to use the idle memory in workstation clusters, but does not limit idle memory usage to file systems. Feeley et al. stress the need for treating the client memories in a distributed file system as a global resource and demonstrate how the GMS algorithm is effective at attaining this objective[Feeley95]. Further modifications to GMS deal with client loads and network overhead[Jamrozik96, Voelker97].

3.2.1 Lookup

The GMS algorithm is similar to N-chance in that it specifies a manager-centric approach to block lookup (Figure 2.3(a)). Block movement in and out of the client caches is reported to managers which maintain exact information about the location of blocks in the client caches. A client contacts a manager on a local cache miss for a block, which then redirects the request to a client caching the block, or to the server if the block is not present in the client caches.

3.2.2 Replacement

The goal of the replacement policy in GMS is to approximate LRU in replacing blocks from the client caches. A key piece of the GMS algorithm is the use of managers to collect and distribute information about the ages of blocks in the client caches.

The GMS algorithm avoids duplicate global blocks in the client caches. Thus, only singlets are forwarded to the cooperative cache. The managers in GMS keep track of the number of copies of a block, and whenever a block becomes a singlet in a client's cache, the GMS manager informs the client. In contrast to N-chance, the responsibility of duplicate avoidance is on the manager rather than on the client. This lowers overhead in GMS as the manager sends a message to a client only when a block becomes a singlet in the client caches, whereas in N-chance the message exchange happens in 98% of all block replacements when a replaced block is not known to be a singlet.

To approximate LRU, a manager in GMS initiates the collection and distribution of age information from clients at dynamically determined time intervals called *epochs*. The duration of an epoch is principally determined by the number of replacements during the epoch. At the beginning of an epoch, clients send a summary of the ages of the blocks in the client caches to the manager coordinating the replacement algorithm. The manager then evaluates this summary to determine the locations of the n oldest blocks in the client caches, where n is determined from the length of an epoch. Subsequently, the manager computes w_i for each client i, where

w_i is the fraction of the n oldest blocks present in a client i. The manager then distributes the w_i array to all the clients along with the expected duration of the epoch. Whenever a client replaces a block from its cache and needs to forward the block to a target client, the client chooses a client i as the target with probability w_i. Measurements showed that the algorithm was able to reasonably approximate LRU over a wide range of memory-intensive benchmarks[Feeley95].

The GMS replacement policy was refined further to deal with high loads in clients. While the original policy was designed to approximate LRU, Voelker et al. experimented with a combination of both client loads and block ages in replacing blocks from the client caches[Voelker97]. The authors found that a load-balanced LRU replacement did not affect performance as the deviations from LRU were moderate, and demonstrated that a 8-node memory server could handle as many as 70 clients with load balancing.

3.2.3 Consistency

The GMS algorithm does not specify any consistency mechanism for the client caches. Instead, the algorithm states that the consistency semantics are the responsibility of the application that created the sharing.

The GMS algorithm adopts a restricted version of delayed write for its update policy. Whenever a dirty block is forwarded to the cooperative cache or shared across multiple clients, the block is written out to the server to ensure that only clean blocks exist in the cooperative cache.

3.2.4 Discussion

The GMS block lookup scheme suffers from the same drawback of high overhead as N-chance because of the reliance on managers to maintain exact block location state, as discussed in Section 3.1.4. In contrast, an approach based on local hints on each client would not incur the overhead of contacting the manager on every local cache miss.

While the GMS replacement policy is able to provide good results, it suffers from three important drawbacks. First, the policy is very complex as the policy decisions in replacements involve hard-to-compute parameters such as the statistical distribution of the oldest blocks in all the clients, the expected rate of replacements and the rate at which age information is expected to become inaccurate. Second, the replacement policy incurs the overhead of exchanging information with managers at every epoch. This creates a tradeoff between overhead and accuracy where optimizing one adversely affects the other. For example, a long epoch would no doubt minimize overhead, but then the age information would grow increasingly inaccurate with time during a long epoch. On the other hand, a short epoch would result in more accurate results but at the cost of increased overhead. The authors have not performed any experiments on this aspect of the GMS replacement policy. Third, the GMS algorithm depends on the manager to coordinate the replacement process and is thus sensitive to manager failures. The GMS algorithm does not specify a mechanism to deal with manager failures. On the other hand, a replacement policy based on local hints would have lower overhead as no communication with the manager is necessary, and would not be susceptible to manager failures.

3.3 Ideal Algorithms

The ideal algorithms provide a performance bound for cooperative caching algorithms. While these algorithms may not be practical to implement, they provide a lower bound on the average block access time of a cooperative caching algorithm. The overhead of an ideal algorithm is ignored because of their impracticality and as a result, the exact mechanisms for implementing cooperative cache management operations are not consequential.

The ideal algorithms assume accurate block lookup in the client caches and maintain block-level consistency in the client caches. The algorithms avoid duplicate global blocks in the client caches by forwarding only singlets and discarding the rest. The only difference in the two ideal algorithms is in the policy of choosing the target client when a client decides to forward a block to the cooperative cache.

3.3.1 Optimal

The replacement policy in the Optimal algorithm always replaces the block in the client caches whose next access is farthest in the future. It has been shown that this replacement policy is optimal because it minimizes the number of cache misses and therefore has the minimal block access time[Belady66]. Thus, whenever a client decides to forward a block to the cooperative cache, the target client is the one that contains the block whose next access is farthest in the future. If several clients contain blocks which are never accessed in the future, Optimal chooses one at random. In addition, if the block to be forwarded to the cooperative cache is itself the one whose next access is the farthest in the future, the block is discarded.

3.3.2 Global LRU

The Global LRU algorithm uses the distributed version of LRU as the basis of its replacement policy. Global LRU approximates Optimal by replacing the globally LRU block from the client caches. Whenever a client needs to forward a block to the cooperative cache, it chooses the target client with the globally LRU block. As in the Optimal algorithm, if several clients contain the globally LRU block, the algorithm chooses one at random. Also, if the block to be forwarded to the cooperative cache is itself the globally LRU block, it is discarded. In summary, Global LRU removes blocks from the client caches in order of age.

In a real implementation, this algorithm would be expensive as the overhead of locating the globally LRU block is high. The degree of approximation achieved by Global LRU depends on the access patterns of a workload and is determined by the degree of similarity between optimal replacement for this workload and Global LRU [Voelker97].

3.4 Summary

This chapter discusses existing and ideal algorithms to provide the basis for comparison with the proposed hint-based algorithm discussed in the next chapter. The existing algorithms provide a yardstick for comparing performance and overhead. while the ideal algorithms define the limits to performance.

The pioneering paper on cooperative caching by Dahlin et al.[Dahlin94] described an algorithm for implementing the cooperative cache called *N-chance*. Managers are responsible for maintaining consistency and block location information in N-chance. The replacement policy in N-chance uses a combination of duplicate avoidance and randomness to decide the best block to replace. Blocks with more than one copy are discarded. while singlets are forwarded to another client chosen at random.

A subsequent paper by Feeley et al.[Feeley95] described the *Global Memory Service* (GMS) which provided better performance than N-chance as well as reduced overhead. GMS is similar to N-chance in that it uses managers to locate blocks in the client caches. One difference is that it does not specify a consistency mechanism.

GMS, like N-chance, forwards a block to the cooperative cache only if the block is a singlet. GMS differs from N-chance in that a manager keeps track of the number of copies of each block and notifies the appropriate client when a block in its cache becomes a singlet. GMS also differs from N-chance in that a centralized manager-based algorithm chooses a block for replacement. The algorithm reasonably approximates LRU over all the client caches.

A summary of the existing algorithms, as well as the hint-based algorithm is shown in Table 3.1. The key features of the hint-based algorithm are elaborated on in the next chapter.

Algorithm	N-chance	GMS	Hint-based
Consistency	Block-based	None	File-based
Lookup	Manager-based	Manager-based	Hints
Replacement	Random	Manager-based LRU	Best-guess LRU
Server Caching	Traditional	Traditional	Discard

Table 3.1: **N-chance, GMS and the hint-based algorithm.** This table lists the key features of the N-chance, GMS, and hint-based algorithms.

The ideal algorithms provide a lower bound on the average block access time of any algorithm. The overhead of ideal algorithms is ignored as they are not practical to implement. The ideal algorithms assume accurate block lookup. block-level consistency and forward only singlets to the cooperative cache.

The two ideal algorithms discussed here. Optimal and Global LRU. only differ in the choice of the target client when a block is forwarded to the cooperative cache. The Optimal algorithm replaces the block which is accessed the farthest in the future and chooses the target client having this block. In contrast. Global LRU emulates the distributed version of LRU and chooses the target client with the globally LRU block.

CHAPTER 4
A HINT-BASED ALGORITHM

4.1 Hints

Hints. in general. can be defined as an imprecise state of a given system. A hint is initially obtained from a system's actual state. However. a hint is not kept consistent with the system's exact state. and may sometimes be updated to reflect the exact state only if the benefit of updation outweighs the cost. As a result. hints cannot be guaranteed to be accurate at any given time.

The use of hints has its advantages and disadvantages. The lack of consistency with the exact system state implies that hints are easier to maintain than is the system state. On the other hand. inaccurate hints can degrade performance and increase overhead because they present an incorrect idea about the state of the system. Consequently. a hint-based system is useful only when the vast majority of hints are accurate and the benefit of using hints is greater than the penalty of a few inaccurate hints.

Different categories of hints can be found in the design of computer systems. The use of hints can be found in networking protocols like TCP/IP[Postel81]. where getting the exact state of a network connection is very difficult: in operating system functions such as disk block lookup in the Pilot operating system[Redell80]. where locating a block on the disk has high cost: and even in programming languages such as Smalltalk[Deutsch82] to resolve the types of polymorphic methods. Detailed information about the use of hints can be found in Chapter 7.2. The goal of this dissertation is the successful application of hints to cooperative caching.

4.1.1 Hints in Cooperative Caching

The goal in designing a cooperative caching system was to develop a distributed algorithm that provides high performance as well as low overhead. One way to do this is to make clients as independent of managers as possible, and allow the clients to take decisions based solely on information available locally. A local decision-making approach has two advantages: not only is there a performance benefit of not contacting the manager, but there is also reduced overhead as all decisions are local.

The greatest challenge in making independent decisions in clients is to make the decision approximate one based on the global state, which is the sum of the local states of all the clients in the distributed file system. As a result, an independent client decision cannot achieve this objective based purely on that client's local state alone. Thus, a client must also maintain information about the local states of the remaining clients in the distributed file system in the form of hints.

As implied by the definition, cooperative caching hints refer to a probable global state of the distributed file system. As a result, hints allow clients to make independent decisions about the cooperative cache based on approximate global state without contacting a manager. Hence, the use of hints fits in perfectly with the goal of devising a cooperative caching algorithm with high performance and low overhead.

The challenge of using hints is to make them accurate. Inaccurate hints can not only hurt performance by causing decisions that do not approximate one based on global state, but also force the system to rely on a costly mechanism to overcome their effect. For example, if a hint says that a particular block is in a client's cache, and in reality, it is not, then the system must have a mechanism to return the correct location of the block in the client caches. Thus, the goal is to make sure that the benefit obtained by using hints more than offsets the cost of making mistakes due to inaccurate hints.

4.2 Block Lookup

This section focuses on the use of hints in locating blocks in the client caches. When a client suffers a local cache miss on a file block, a lookup must be performed on the client caches to determine if and where the block is cached. In the previous algorithms, the client contacts the manager responsible for the block to perform the lookup. Although this approach will always return the correct location of the block in the client caches, it also increases both the block access time and the manager load.

An alternative is to let the client itself perform the lookup, using its own hints about the locations of blocks within the client caches. These hints allow the client to access the caches of other clients directly without contacting a manager on every local cache miss. However as discussed in Section 2.4, the hints need to be accurate for this approach to be effective. In addition, a hint-based block lookup scheme must provide the correct result even if hints are incorrect. In other words, the lookup mechanism must be able to return the correct location of the block in the storage hierarchy even if the block's location hints are not correct.

The key focus of this hint-based approach to perform block lookup is to make the location hints accurate and provide a lookup mechanism to deal with inaccurate hints.

4.2.1 Hint Accuracy

Hint accuracy largely depends on the choice of blocks to which location hints refer. This choice is important because it affects how fast hints get out-of-date in the client caches. If the set of blocks referred to by hints changes location rapidly, the hints would have to be frequently updated to ensure accuracy. However, changes to block location are determined primarily by access patterns which are not known in advance, making it difficult to predict the natures of changes to the locations of blocks. To maximize hint accuracy, clients maintain hints for the part of the location state that changes the least rapidly. The reasoning is that hints for this

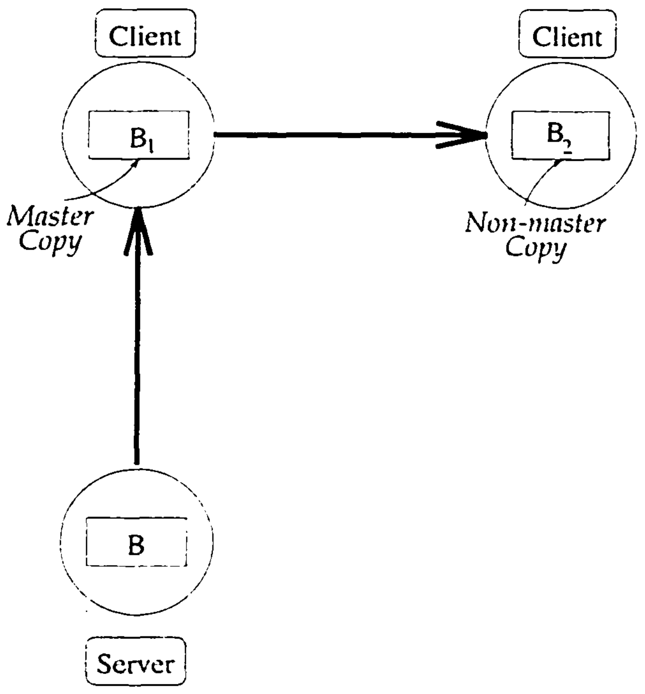


Figure 4.1: **Master Copies.** The *master copy* of a block is that which is obtained from the server. In this figure, blocks B_1 and B_2 are copies of the block B on the server. Block B_1 is a master copy because it was obtained from the server (as shown by the arrow). On the other hand, block B_2 is not a master copy because it was obtained from the client having B_1.

part of location state are not likely to get out-of-date as fast as those for the parts that change more rapidly. The key to the success of this approach is to find this part of the location state of a distributed file system.

One alternative is to keep track of all the copies of all blocks in the client caches as is done in a manager-centric approach. However, the aggregate rate of movement of all the copies of a block in and out of the client caches is larger than the rate of movement of a single copy. Thus, the overhead of maintaining hints for all the copies of a block in the client caches is more than that for a single copy of a block, making the latter a more attractive option.

The next question is to decide which copy of a block should a block location hint refer to. One possibility is to refer to the newest copy of a block in the client caches because this copy is likely to be present in a client's cache and consequently, a location hint that refers to this block is also likely to be accurate. On the other hand, the location of the newest copy is likely to change more rapidly than that for a fixed copy of a block. This makes a fixed-copy scheme more advantageous for accurate hints.

One concern about a fixed copy scheme is that it increases the load on a client caching the fixed copy of a block as this client is the target of accesses from clients needing the block in the future. However, as shown in Section 6.4.3, client loads were not an issue.

To allow hints to keep track of the location of a fixed copy of a block, the concept of a *master copy* of a block is introduced. The first copy of a block to be cached by any client is called the master copy, as shown in Figure 4.1. The master copy of a block is distinct from the block's other copies because the master copy is obtained from the server. Block location hints only contain the probable location of the master copy of a block, simplifying the task of keeping the hints accurate:

1. When a client opens a file, the client contacts a consistency manager which gives the client a set of hints that contain the probable location of the master copy for each block of the file. Here, we assume that a manager coordinates the consistency mechanism in the distributed file system, though we place no restriction on the mechanism itself. The manager obtains the set of hints for the file from the last client to have opened the file. The assumption is that the last client to open the file has the most accurate location hints for this file. While this assumption may not hold for some patterns of block access, they conform to typical UNIX access patterns[Baker91] and yield highly accurate hints, as demonstrated in Chapters 5.2.2 and 6.4.4.

2. During the process of replacement (described in Section 4.3), when a client forwards a master copy of a block to another client, both clients update their hints to reflect the new location of the master copy in the client caches.

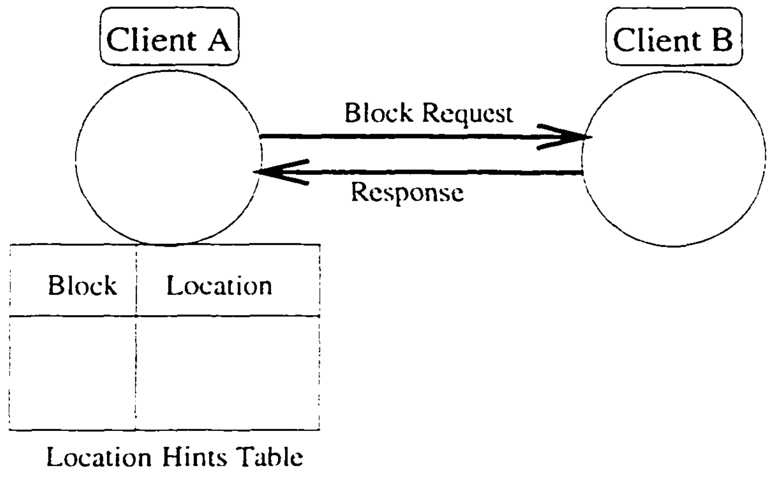


Figure 4.2: **Lookup Mechanism.** This figure illustrates the use of location hints in the lookup mechanism. Once client A obtains location hints, the client uses the information in the table of location hints to determine that client B is caching a needed block. Following this, client A sends a request to client B, which responds with the block.

4.2.2 Lookup Mechanism

The hints contain the probable location of the master copy of a block. Thus the lookup mechanism must ensure that a block lookup is successful, regardless of whether the hints are right or wrong. Fortunately, in cooperative caching algorithms, dirty blocks are written out to the server whenever they are shared or forwarded to the cooperative cache. As discussed in Chapter 2.5, an important corollary of this assumption is that the server can always be relied to have a valid copy of a block and can satisfy requests for the block should the hints prove false. This simplifies the lookup mechanism:

1. When a client has a local cache miss for a block, it consults its hint information for the block.

2. If the hint information contains the probable location for the master copy of the block, the client forwards its request to this location. Otherwise, the request is sent to the server.

3. The client which receives a forwarded request for a block checks to see the block is present in its cache. If so, the client responds with the block to the

requesting client. Otherwise, the client consults its hint information for the block and proceeds to Step 2.

Measurements show this algorithm works well when the working set size of client applications is less than the size of the client memories, as shown in Chapter 5.2.7.1. However, if several clients share a working set of blocks larger than the client memories, forwarding a block to a client increases the probability that a master copy will be replaced. This, in turn, will cause the master copy to be forwarded to another client, greatly increasing the rate of movement of blocks in and out of the client caches. As a result, the location hints for blocks will also lose their accuracy at a faster rate and consequently, these inaccurate hints will decrease the hit ratio to the cooperative cache.

However, a similar degradation occurs in other cooperative caching algorithms. When the working set size of client applications starts to approach the size of the client memories, block movement in and out of the client caches increases as described above. While this does not cause any loss of accuracy for block location, frequent communication with the manager to report this block movement increases the manager load to as much as 30 times that of the hint-based algorithm, as shown in Section 5.2.7.1. The high load increases the manager response time, which in turn degrades performance.

4.3 Replacement

The replacement policy in cooperative caching makes space for a new block entering the client caches by discarding the least valuable block in the client caches. The distributed nature of the cooperative cache makes this challenging as the client fetching the new block might be different from the client holding the least valuable block, requiring communication among clients. The degree and nature of communication among clients determines the performance and overhead of a cooperative caching replacement algorithm.

A client invokes the replacement policy when a block is replaced from its local cache. As discussed in Chapter 2.5, the client has to decide first whether or not to forward this replaced block to the cooperative cache. If the client decides to forward the block, it must choose the target client to which to forward the block.

4.3.1 Forwarding

The previous algorithms rely on the manager to determine whether or not a block should be forwarded. A block is forwarded only if it is a singlet. Maintaining this invariant is expensive: in addition to reporting all block movement in and out of the client caches to the manager, it also requires an N-chance client to contact the manager whenever it wishes to replace a block, and the GMS manager to contact a client whenever a block becomes a singlet.

An alternative is to avoid the use of a manager in determining the copy of a block to be forwarded to the cooperative cache. To avoid overhead, the copy to be forwarded to the cooperative cache is predetermined and does not require communication between the clients and the manager. In particular, only the master copy of a block is forwarded to the cooperative cache and all other copies are discarded. As only master copies are forwarded, and each block has only one master copy almost all the time, there are no duplicate global blocks in the client caches.

Blocks may have more than one master copy in the client caches if location hints are inaccurate. For example, out-of-date hints in the client caches can cause a client's block lookup request to get forwarded to the server even though the block is present in the cooperative cache. Consequently, the client will fetch the block from the server, increasing the number of master copies of the block by one. Fortunately, measurements in Chapter 5.2.2 indicate that less than 0.01% of the server accesses fetch a second master copy of a block into the client caches. Moreover, after a client fetches a second master copy of a block, the location hint for the block will now refer to the second master copy and this hint will propagate to other clients which want to access the block in the future. Therefore these clients will reference the second master copy instead of the first. Consequently, the first master copy of the block will be discarded from the client caches at a faster rate than if there was only one master copy in the client caches.

A potential drawback of the master copy algorithm is that it has a different forwarding behavior than the previous algorithms. Instead of forwarding the last local copy of a block in the client caches as in GMS or N-chance, the master copy algorithm forwards the first (master) copy. In some cases, this may lead to unnecessary forwards. As the previous algorithms forward the last copy of a block, the block will not be forwarded at all if it is deleted while there are multiple copies in the client caches. The hint-based algorithm may unnecessarily forward the master copy prior to the delete. Fortunately, our measurements in Chapter 5.3 show that few (1.97%) of the master copy forwards are unnecessary.

4.3.2 Best-Guess Replacement

Once the replacement policy has decided to forward a block to the cooperative cache, it must choose the least valuable block in the client cache to replace. This block is present in the cache of a target client, and forwarding a block to the target client aims to replace this least valuable block.

N-chance chooses a target client at random, while GMS relies on information from the manager. A policy of randomly choosing a target client may not replace the least valuable block in the client caches and this results in poor performance, as seen in Chapter 5.2.1. Relying on information from the manager allows GMS to reasonably approximate Global LRU. However, this incurs the overhead of frequent communication with the manager and increases the complexity of a replacement algorithm.

An attractive alternative is to design a replacement algorithm without involving a manager, thus avoiding the overhead of contacting the manager. However, this implies that clients have to exchange information among themselves to implement the replacement algorithm, incurring overhead in the process. Hence, the key focus of such an algorithm should be to exchange the minimal amount of information between clients necessary for a highly accurate replacement policy.

A distributed replacement algorithm must aim for high accuracy as well as adaptability to changes in the behavior of clients. Thus such an algorithm must able to identify the *active* clients (clients accessing the cooperative cache) and *idle* clients (clients that are not) in a distributed file system at any given point of time. In addition, an algorithm must also be able to adapt to situations where an idle client suddenly becomes active and vice versa. This would imply changes in the distribution of idle memory in the client caches, requiring adjustments in the replacement process.

Finally, a distributed replacement algorithm should not be affected by client failure. In case a client fails, the remaining clients must be able to proceed with the replacement algorithm. Fortunately, this is achievable in a hint-based algorithm as each client maintains its own replacement hints.

4.3.2.1 Algorithm

A hint-based algorithm chooses a target based on local information about the state of the client caches. This is referred to as *best-guess* replacement because each client chooses a target client that it believes has the system's oldest block. The objective is to approximate Global LRU, without requiring a centralized manager or excessive communication between the clients. The challenge is that the block age information is distributed among all the clients, making it expensive to determine the best block to replace.

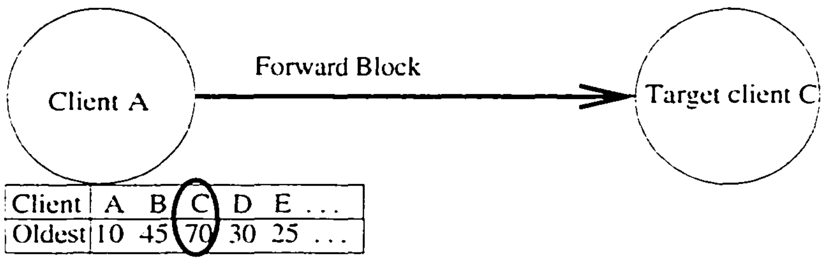
In best-guess replacement, each client maintains an *oldest block list* that contains what the client believes to be the ages of the oldest block on each client. The algorithm is simple: blocks are forwarded to the client that has the oldest block in the oldest block list.

The high accuracy of best-guess replacement comes from exchanging information about the ages of the oldest block on each client. When a block is forwarded from one client to another, both clients exchange the age of their current oldest block, allowing each client to update its oldest block list. This step in the algorithm is illustrated in Figure 4.3.

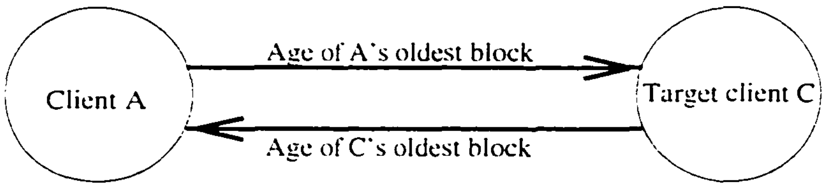
When a client boots up, the client assumes that the remaining clients have free

Client	A	B	C	D	E ...
Oldest	10	45	70	30	25 ...

(a) Oldest Block List



(b) Selecting a target client



(c) Exchanging age information

Figure 4.3: **Best-guess Replacement.** (a) Each client contains replacement hints in the form of the *oldest block list.* The oldest block list is indexed by every client in the distributed file system and each entry contains the probable age of the oldest block in the client. (b) When a client A decides to forward a block. the client looks in its oldest block list and determines that client C has the oldest block in the list of age 70. Client A then forwards its block to the target client C. (c) Concurrently with the replacement. clients A and C exchange the ages of their oldest blocks and update their respective lists.

memory to store global blocks and proceeds with this assumption until the oldest block list is updated during the course of replacements. Accordingly. the probable age of the oldest block for each client in the oldest block list is initialized to INF. where INF is the maximum age of a block. Similarly. a client having free memory in its cache also returns INF as the age of its oldest block during replacement.

4.3.2.2 Properties

This exchange of block ages allows both active and idle clients to maintain accurate oldest block lists. Active clients have accurate lists because they frequently forward blocks. Idle clients will be the targets of the forwards, keeping their lists up-to-date as well. Active clients will also tend to have young blocks, preventing other clients from forwarding blocks to them. In contrast, idle clients will tend to accumulate old blocks and therefore be the target of most forwards.

Best guess replacement adapts to changes in the behavior of a client. An active client that becomes idle will initially not be forwarded blocks, but its oldest block will age relative to the other blocks in the system. Eventually this block will be the oldest on the lists, and therefore used for replacement. On the other hand, an idle client that becomes active will initially have an up-to-date list because of the blocks it was forwarded while idle. This allows it to forward blocks accurately. Other clients may erroneously forward blocks to the newly-active client but once they do, their updated oldest block lists will prevent them from making the same mistake twice.

Best-guess replacement is also not prone to client failures. This is primarily because replacement information is not centrally managed at any one client and each client maintains its own replacement hints. As a result, if a client fails, the remaining clients can stop replacing blocks from the failed client upon detection. Failure detection can be done either actively using a heartbeat protocol or lazily during the process of replacement. To minimize overhead, we have opted for lazy detection (more details are available in Chapter 6.1.1.3).

4.3.2.3 Analysis

The performance of best-guess replacement is measured by the extent of its deviation from Global LRU, the ideal for this algorithm. The extent of this deviation depends both on the workload in the distributed file system as well as the deviation of the optimal replacement algorithm for this workload from Global LRU. An initial estimate can be obtained by analyzing the *bound* which determines the maximum extent of deviation of best-guess replacement from Global LRU.

The performance bound of best-guess replacement indicates that the algorithm approximates Global LRU very well. Let us assume a distributed file system with

N clients, where we are trying to determine the maximum extent by which the replacement of the globally LRU block can be delayed. Consider the scenario in which a client replaces a block from another client in best-guess replacement. The two clients then exchange the ages of their oldest blocks. After this replacement, neither client will replace a block younger than the oldest blocks on these two clients because clients choose the target client with the oldest block in the oldest block list. Note that at this time, the globally LRU block is the oldest block on one of these N clients. Therefore, after all possible replacements and subsequent exchanges of information between N clients, no client can replace a block younger than the oldest block on all the N clients, which is the globally LRU block. Since these are $(N-1)(N-2)/2$ possible replacements between N clients, the replacement of the globally LRU block can be delayed by at most this many replacements. In other words, if Global LRU replaces a particular block in the i^{th} replacement, then best-guess replacement will replace the block at most by the $i+(N-1)(N-2)/2^{th}$ replacement.

While this limit seems high, it represents the worst case because the analysis assumes that every client is replacing blocks from the caches of other clients. In a typical distributed file system, not all clients access the cooperative cache at the same time[Acharya98]. Under these conditions, best-guess replacement does much better. If there are k active clients accessing the cooperative cache at any given time, then the globally LRU block can be assumed to be in the remaining $N-k$ clients (as the caches of the k clients is filled with young local blocks). Following the same logic as above, after all possible exchanges between these k active clients and $N-k$ idle clients, no client can replace a block younger than the globally LRU block, which the oldest block on one of these $N-k$ clients. In such a case, the replacement of the globally LRU block is delayed by at most $(k-1)(N-k-1)/2$ replacements. Furthermore, in both simulations and prototype measurements, k was on an average a small constant compared to N, implying that the delay in the replacement of the globally LRU block was at most by $O(N)$ replacements. Given that the performance of most applications is not very sensitive to small deviations from Global LRU[Voelker97], the existence of a linear bound is very encouraging.

The performance bound of best-guess replacement is not affected by exchanging more age information. The reason is that the strategy does not reduce the probability of a replacement error. For example, assume two clients exchange the block age information of all the clients. However, other than the ages of the oldest blocks of these two clients, the rest of the block age information can not be guaranteed to be accurate. The probability of a replacement error depends on the accuracy of the oldest block's age on any client and is therefore not affected by the rest of the block age information. As a result, it is best to exchange only the age of the oldest blocks of the two clients involved in the replacement.

While the performance bound determines the maximum extent to which best-guess replacement can deviate from Global LRU, the performance is determined by the average delay in the replacement of the globally LRU block. Even though the linear performance bound suggests that best-guess replacement is likely to closely approximate Global LRU, the actual performance is best evaluated using simulations and prototype measurements.

4.3.2.4 Forwarding Storms

Although both theory and practice have shown this simple algorithm to work well, there are potential problems, the most important of which is overloading a client with simultaneous replacements. This phenomenon is known as a *forwarding storm* and happens when several clients believe that the same target client has the oldest block. In such a situation, the clients all forward their blocks to this target client, potentially overloading the client and replacing young blocks.

Fortunately, it is highly unlikely that the clients using the cooperative cache would forward their blocks to the same target. This is because clients that do forward their blocks to the same target will receive different ages for the oldest block on the target, since each forwarded block replaces a different oldest block in the target client. Hence, if the idle memory of the network is uniformly distributed over k clients, the probability of a client choosing a particular target client among these k clients is $1/k$. Thus, the probability of x clients choosing the same target client is $(1/k)^{x-1}$, as each client replacement decision is independent of one another. Similarly, the probability of a client choosing the same target client y successive times is $(1/k)^{y-1}$. Consequently, the probability for x clients choosing the same target client y successive times is $(1/k)^{x+y-2}$. Assuming (arbitrarily) that a forwarding storm involves 5 clients forwarding blocks to 10 target clients over a period of 10 successive replacements on each forwarding client, the probability of the forwarding clients choosing the same target client over this period is 10^{-13}. This indicates that clients using the cooperative cache would be very unlikely to forward the block to the same target client. Furthermore, we did not observe any forwarding storm either in the simulations or in the prototype.

4.3.2.5 Client Loads

Best guess replacement can also be configured to work with client loads in addition to block ages. When a client forwards a block to the cooperative cache on a target client, the two clients involved in a replacement could exchange the age of their oldest blocks and their CPU loads. As a result, a client could use the information to avoid a heavily loaded client even though it might have very old blocks in its memory. However, the exact mechanism with which a block would be valued both with respect to its age and its CPU load is a matter of future study. Though research has shown that client loads may be an issue if idle clients are used as dedicated remote memory servers[Voelker97], these high client loads were not observed in the course of our measurements because idle clients have very low CPU loads in a typical distributed file system[Acharya98].

4.4 Cache Consistency

Cooperative caching, for the most part, poses no restrictions on any cache consistency protocol, as discussed in Chapter 2.6. Cooperative caching algorithms can work with any update policy or consistency mechanism, though the use of certain policies like write-though or a restricted version of delayed write simplifies the management of the cooperative cache. The only aspect of cache consistency which is directly affected by the choice of a cooperative caching algorithm is granularity. where the level of coordination in cooperative caching directly impacts the granularity of cache consistency.

One approach is to use block-based consistency. but this requires frequent communication with a manager to locate an up-to-date copy. making it pointless to use hints for block lookup or replacement. For this reason. the hint-based algorithm uses a file-based consistency mechanism and uses *tokens* to control access to shared files. Clients must acquire either a *read* or *write* token from a manager prior to accessing a file for reading or writing. The manager controls the file tokens. revoking them as necessary to ensure that at most one client has exclusive access to a write token. even though multiple clients may hold read tokens simultaneously. Once a client has a file's token. the client may access the the file's blocks without involving the manager. enabling the use of hints to locate and replace blocks in the client caches.

4.5 Discard Cache

As discussed in Chapter 2.7. the server memory is underused in cooperative caching because the high hit ratio to the client caches reduces the number of server accesses. This raises the question of effectively using the server memory to benefit cooperative caching.

4.5.1 Alternatives

One option is to treat the server memory as part of the cooperative cache. The server is treated like any other client in the distributed file system and clients can forward blocks to the cooperative cache in the server memory. This would increase the effective size of the cooperative cache and therefore the hit rates on the server memory. as the server no longer duplicates the contents of the client caches.

A more attractive option is to use the server memory to complement the use of hints in cooperative caching. When the working set of client applications starts to approach the size of the client memories and blocks move frequently in and out of the client caches. the distribution of idle memory in the distributed file system also changes rapidly. resulting in incorrect replacement hints. As the number of clients actively accessing their caches is high under these circumstances. an incorrect replacement hint will cause a block to be forwarded to such an active client and a valuable local block is consequently replaced from the client caches. The use of the server memory to reduce the effect of these incorrect hints would be beneficial. Furthermore. since the server memory contains young mistakenly replaced blocks. our hypothesis is that this will increase the use of the server memory more than if the server memory were used as part of the cooperative cache. Fortunately. the results in Table 5.7 confirm this hypothesis.

4.5.2 Mechanism

To offset replacement mistakes. the notion of a *discard cache* is introduced. one that is used to hold possible replacement mistakes and thus increase the overall cache hit rate to the server memory. A client chooses to replace a block on a particular target client because the client believes that the target client contains the oldest block in the client caches. The target client considers the replacement to be in error if it does not agree with this assessment.

Two heuristics are used to determine whether the replacement of a block was erroneous. The first heuristic relates to the type of blocks which are sent to the discard cache on the server. One possibility is to send all mistakenly replaced blocks to the server. However. this ignores the fact that mistakenly replacing some types of blocks does not affect performance. For example. performance would not be affected if a duplicate copy of a block were to be mistakenly replaced.

Another possibility sends a mistakenly replaced block to the discard cache only if the block is a singlet. as mistakenly replacing duplicates does not affect performance. However. in a hint-based system. there is no mechanism to count the number of copies of a block in the client caches. The next possibility leverages the use of master copies in the hint-based algorithm and sends only mistakenly replaced master copy blocks to the discard cache. First, the target client checks whether the replaced block is a master copy. If the replaced block is not a master copy, then the replaced block is discarded and the replacement is not considered erroneous.

The second heuristic tries to determine whether the replacement was a mistake. The spectrum of possibilities to determine a mistaken replacement is wide. In general, possible approaches can be classified as either pessimistic or optimistic. The pessimistic policy would tend to assume that the replacement of a master copy is in error unless proved otherwise. In contrast, the optimistic policy would be biased towards declaring the replacement of a master copy to be correct. A pessimistic policy increases the number of erroneous replacements sent to the discard cache but at the same time benefits from a higher number of hits to the server memory. The key in choosing a policy is to ensure that the number of replacement errors does not overload the server, while at the same time increasing the hit rate on the discard cache.

To choose an appropriate policy, we experimented with two heuristics. The first one is optimistic and declares the replacement of a block to be in error if the block is a master copy and is younger than *all* blocks in the oldest block list of the client from which the block is replaced. The second one (pessimistic) is similar to the first but differs in that a replacement of master copy is declared to be in error if the block is younger than *any* of the blocks on the list. In both these heuristics, a mistakenly replaced block is sent to the discard cache; otherwise, the block is discarded. During the course of the simulations and prototype measurements, we found that even with a pessimistic heuristic, the number of replacement errors were low compared to the number of replacements (Table 6.5), while the server hit rate was higher than that in other uses of server memory (Table 5.7). This convinced us to go with the pessimistic heuristic in determining a replacement error, as summarized in Table 4.1.

A more informed decision could possibly be arrived at if the clients had more knowledge about the ages of blocks in the caches of other clients. For example, if the clients involved in replacement exchanged the ages of the z oldest blocks in their respective caches, then the determination of whether the replacement of a block is erroneous could be determined more accurately. However, best-guess replacement limits the amount of information exchanged during a replacement process to minimize overhead. Moreover, as seen during the simulations and the prototype measurements, the number of replacement errors were too few to warrant the extra overhead of higher accuracy in determining replacement errors.

Type of block	**Action**
Non-master copy	Discard
Old master copy	Discard
Young master copy	Send to discard cache

Table 4.1: **Discard Cache Policy.** This table lists how the hint-based replacement policy decides which blocks to send to the discard cache. A master copy is *old* if it is older than all blocks in the oldest block list, otherwise it is considered *young*.

The replacement policy in the discard cache is based on the age of the blocks sent to the discard cache. This means that the discard cache always replaces its oldest block, even though this oldest block may have been forwarded to the discard cache more recently than the remaining blocks in the discard cache. This is to ensure that mistakenly replaced young blocks have a longer lifetime in the discard cache than relatively older blocks, irrespective of the order in which the blocks were forwarded to the discard cache.

4.5.3 Size

The size of the discard cache required to augment effectively the use of hints in cooperative caching is also of concern. If the required size is larger than the average size of server memories, then the server is unlikely to be useful as a discard cache. Fortunately, the required size of the discard cache is relatively small at $O(N)$ blocks, where N is the total number of clients in the distributed file system. This result is derived from the properties of best-guess replacement. If the replacement of the globally LRU block is delayed by $O(N)$ replacements, then a block younger than the globally LRU block will be replaced from the discard cache before the globally LRU block unless the size of the discard cache is at least $O(N)$ blocks. In fact, in both the simulations and the prototype, the server memory size was adequate enough that adding memory did not increase the hit ratio to the discard cache.

4.6 Summary

This chapter presents a new cooperative caching algorithm based on hints. Hints refer to the probable global state of a distributed file system and are less expensive to maintain than facts. As long as the benefit of hints exceeds the penalty of a few inaccurate ones, a hint-based system provides high performance and low overhead.

Every client has lookup hints which refer to the master copy of a block which is that copy directly fetched from the server. The client obtains the location hints for the blocks of a file from the last client that opened the file (with help from a consistency manager). A client uses the hints to obtain blocks directly from the caches of other clients.

Hints are also used to replace blocks from the client caches. A client first decides to forward a block to the cooperative cache only if the block is a master copy, discarding the block otherwise. If the client decides to forward the block, the target client which receives the block is the one which has the oldest block in the forwarding client's oldest block list (a list containing the probable ages of the oldest block on each client). A client maintains the accuracy of this list by exchanging the age of its oldest block with that of the target client during a replacement.

Hint-based cooperative caching uses file-based consistency rather than block-based because it allows clients to access blocks without contacting a consistency manager. The final aspect of the hint-based algorithm is the use of a heuristic to determine if the replacement of a block from the client caches is a mistake, sending such mistakenly replaced blocks to a discard cache on the server memory.

CHAPTER 5
SIMULATION

This chapter evaluates the performance of the hint-based algorithm through trace-driven simulation. The simulation results allow us to gain detailed insight into the relative performance and overhead of the hint-based algorithm compared to existing and ideal algorithms. The simulation environment is described first, followed by the criteria for evaluating the algorithms. Finally, the performance and overhead of all the algorithms are analyzed.

5.1 Simulation Environment

5.1.1 Traces

The algorithms were evaluated using trace-driven simulation based on the traces of applications running on the Sprite network-based operating system[Baker91]. The Sprite operating system ran on a network of about 40 Sun and DEC workstations and provided a UNIX application interface[Ousterhout88]. Files in Sprite were stored in servers and cached by clients, with strong consistency maintained among the cached copies. One important aspect of these traces with respect to cooperative caching is that Sprite encouraged sharing of files.

There were about 50 Sprite users, distributed among several academic research groups, and engaging in multiple office and engineering tasks. The traces record the file accesses of various applications in electronic communication, typesetting, editing, software development, VLSI circuit design and graphics. These applications are still widely used and are thus not specific to Sprite.

These traces cover four two-day *periods*, and record three different types of traces: lookups, file activity and attribute management. The lookup and attribute management records are not directly relevant to cooperative caching activity and thus not used in our study. The file activity records contain file system accesses by applications, such as opening and closing files, and seeking on file descriptors. Actual read and write events were not recorded, but can be inferred from file offsets in other records.

The traces were restricted to the use of the main file server *allspice*. Table 5.1 shows statistics for the trace periods, while Table 5.2 shows the simulation parameters.

Trace Parameter	**Period**			
	1	2	3	4
Block reads	276,628	2,011,915	261,023	343,189
Unique blocks accessed	53,349	13,108	33,063	75,273
Active clients	32	24	38	34

Table 5.1: **Trace Period Statistics.** This table contains statistics for the four trace periods. *Active clients* refers to the number of clients that actually used the cooperative cache during the period.

Most of the simulation parameters are derived from the original study on cooperative caching by Dahlin et al.[Dahlin94], to simplify performance comparisons. The access times were obtained from previously published measurements. Although these measurements were taken around 1994 and are likely to be slow when compared to state-of-the-art equipment, they were obtained from real systems. While it was possible to update the simulation parameters, it would have made it difficult to compare results with N-chance.

Clients	42	Servers	1
Client Cache Size	16 MB	Consistency	strong
Server Cache Size	128 MB	Block Size	8 KB
Local Latency	0.25 ms	Remote Latency	1.25 ms
Disk Latency	15.85 ms	Write policy	write-through
Warm-up Block Accesses	400,000	Message Latency	0.2ms

Table 5.2: **Simulation Parameters.** This table describes the environment used to evaluate the various cooperative caching algorithms. The rows *Local Latency*, *Remote Latency* and *Disk Latency* refer to the average time to access a file block from the local cache, cooperative or server cache, and disk respectively.

The N-chance simulator was obtained from the designers of the algorithm, while we developed the simulators for the hint-based, GMS and the two ideal algorithms. The design of the simulator for the N-chance algorithm assumed write-through caching and strong consistency with write-invalidation. The same assumptions were made in the remaining simulators to make sure that there were no important differences in how the simulators handled events and to allow for fair comparison.

An important assumption was that there was a single manager handling centralized functions such as consistency and block location. This makes it easier to measure the manager load imposed by the different systems, without introducing an algorithm to distribute the load over multiple managers. This assumption was also made by the designers of the N-chance simulator.

The N-chance simulator was modified to incorporate additional functionality used in the *xfs* file system[Anderson95]. In the modified system, a manager preferentially forwards a request to the client caches instead of a server, improving the cooperative cache hit ratio and reducing the number of server accesses.

As the GMS algorithm does not specify a consistency mechanism, the GMS simulator used file-based consistency, which was identical to that used in the hint-based algorithm for reasons of fairness.

5.1.2 Evaluation Criteria

As stressed earlier, the key focus of this dissertation is to evaluate cooperative caching algorithms in terms of both performance and overhead. Consequently, the following two metrics are used to evaluate the cooperative caching algorithms:

- **Average Block Access Time**: This metric is the average time required to access a file block. The access time is determined by the hit ratios to the different layers of the storage hierarchy. Algorithms that make better use of the local and cooperative caches to avoid disk accesses will have lower access times. Access time is only measured for block reads because all algorithms use write-through caches. The average block access time is measured for both ideal and existing algorithms.

- **Overhead**: This metric is the work required to manage the cooperative cache. The overhead is primarily the messages exchanged between the clients and the managers to coordinate cooperative caching. The computational overhead of cooperative cache management is considered negligible: in the simulation environment, cache management takes about 5 μs on an average while the total latency to get a 8 KB block from a remote client cache is around 1.25 ms.

  The overhead is broken down into manager load, messages for block lookup and replacement, and network and client loads. The manager load is expressed as the number of messages sent and received by the manager. This is a reasonable measure of manager load because each message represents work by the manager to coordinate cooperative caching. The overhead measurements are not done for ideal algorithms because they are impractical to implement.

5.2 Simulation Results

This section describes the performance and overhead of the cooperative caching algorithms when simulated using the Sprite traces. These results were first described in an earlier paper[Sarkar96]. However there were two bugs that affected some of the results but fortunately not our conclusions. The first bug was related to the incorrect processing of the file identifier field in the traces by the simulators for the hint-based, GMS, Optimal and Global LRU algorithms. This reduced the number of unique file identifiers, and erroneously increased the local cache hit ratio due to the resultant improved locality. The second bug was related to a version numbering problem in the simulator for the hint-based algorithm which increased the number of disk accesses at the expense of remote cache hits. The bugs marginally ($< 1\%$) affected the average block access times in all the trace periods except for the second. In the second period, the average block access time of the affected algorithms was incorrectly reduced by a factor of three. The bugs also affected the sensitivity analysis, revealing a small divergence (5%) between the block access times of the hint-based and ideal algorithms as seen in Section 5.2.7.

The performance of the algorithms are compared in terms of average block access time while the overhead is discussed using manager load, lookup messages, replacement messages, and network and client loads. The effectiveness of the discard cache is also measured, as is the sensitivity of the block access time to variations in the simulation parameters. All the simulation results presented in this chapter ignore the warm-up block accesses.

5.2.1 Block Access Times

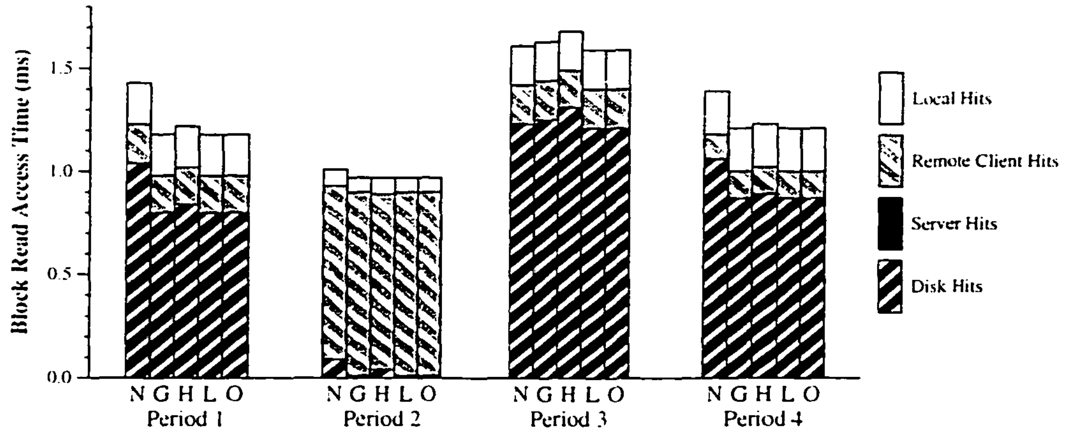


Figure 5.1: **Block Access Time.** This figure shows the average block access times for the N-chance(N), GMS(G), hint-based(H), Global LRU(L) and Optimal(O) algorithms for each period of the Sprite traces. The segments of the bars show the fraction of the block access time contributed by hits to the local cache, remote caches, server cache, and disk.

Figure 5.1 shows the average block access time for all the algorithms, broken down by the time spent in accessing the local cache, remote caches, server cache, and disk. The average block access times for the GMS and hint-based algorithm are very close to the ideal algorithms, and they spend similar amounts of time handling hits to the different levels of the storage hierarchy. The average block access time of the hint-based algorithm is about 3% worse than that of the other algorithms in the third period. This working set requirements of this period are sometimes comparable (as much as 92%) to the aggregate size of the client memories and as a result, the degree of block movement in and out of the client caches in this period is relatively higher than that in the other periods. The high degree of block movement causes some inaccuracies in block location hints in this particular period, as seen in Table 5.4. Table 5.3 provides the distribution of hits to the different layers of the storage hierarchy averaged across all the traces. Overall, the performance of the hint-based algorithm is particularly encouraging, given that hints can be occasionally incorrect.

Algorithm		N-chance	GMS	Hint-based	Global LRU	Optimal
Hit Distr. (%)	**Local**	47.2	45.5	47.3	47.3	47.4
	Remote	49.8	52.5	50.4	50.9	50.9
	Server	0.0	0.0	0.2	0.0	0.0
	Disk	3.0	2.0	2.1	1.8	1.7

Table 5.3: **Distribution of Hits.** This figure shows the distribution of hits (*Hit Distr.*) to the different layers of the storage hierarchy for the N-chance(N), GMS(G), hint-based(H), Global LRU(L) and Optimal(O) algorithms averaged across all the periods of the Sprite traces. The rows *Local*, *Remote*, *Server* and *Disk* refer to the percentage of total accesses to the local cache, remote caches, server cache, and disk.

The N-chance algorithm has a higher number of disk accesses compared to the others in all of the trace periods except for the third. This is caused by N-chance's random replacement policy, coupled with the low degree of sharing in these periods. The probability that a replaced block in a randomly-chosen target client was being accessed by the client is higher than that in an idle client. The low degree of sharing makes it likely that there are no other copies of the randomly replaced block in the client caches. Consequently, a future access to the replaced block would likely result in a disk access. Further evidence of this phenomenon was found by Feeley et al[Feeley95].

5.2.2 Lookup Messages

The overhead imposed by lookup messages in hint-based cooperative caching depends on the accuracy of hints. If a hint is accurate, block lookup takes two messages: one to send a block request to a client or the server, and one for the client or the server to respond with the requested block. If a hint is inaccurate, there is an additional message for each time the block request is forwarded to another client or the server.

Period	1	2	3	4	average
Hint Correctness(%)	99.68	99.98	99.07	99.54	99.94
Absolute Correctness(%)	99.77	99.97	99.44	99.33	99.93
False Negatives(%)	0.008	0.010	0.015	0.007	0.010

Table 5.4: **Block Location Hint Accuracy.** The row *Hint Correctness* refers to the percentage of local cache misses where block location hints correctly determine that the block is in the client caches. The row *Absolute Correctness* represents the percentage of correct block location hints that point to the actual location of the block. The row *False Negatives* represents the percentage of local cache misses when the block is in the client caches but the hints say otherwise. The column *average* refers to the average for the categories of hint accuracy across all periods.

The simulations revealed that the block location hints for the client caches are highly accurate (Table 5.4). For only 0.01% of the local cache misses (averaged across all periods) is the desired block in the client caches but the hints say otherwise. Conversely, when a hint says a block is in the client caches, it is correct for 99.94% of all local cache misses. Of these correct hints, 99.93% point to the actual location of the block, while the remaining result in requests being forwarded. The high hint accuracy and the small number of forwarded requests translate into an average of only 2.001 ± 0.006 messages to perform a block lookup. In comparison, both N-chance and GMS always require 3 messages per block lookup: one to send a block request to the manager, one for the manager to forward the block request to a client, and one for the client to respond with the requested block.

5.2.3 Manager Load

The load imposed on the manager is one measure of the overhead of an algorithm. Figure 5.2 shows the manager load, expressed as the number of messages sent and received by the manager per block access. This metric allows an estimate of the load on the manager given block access patterns other than the ones in the traces.

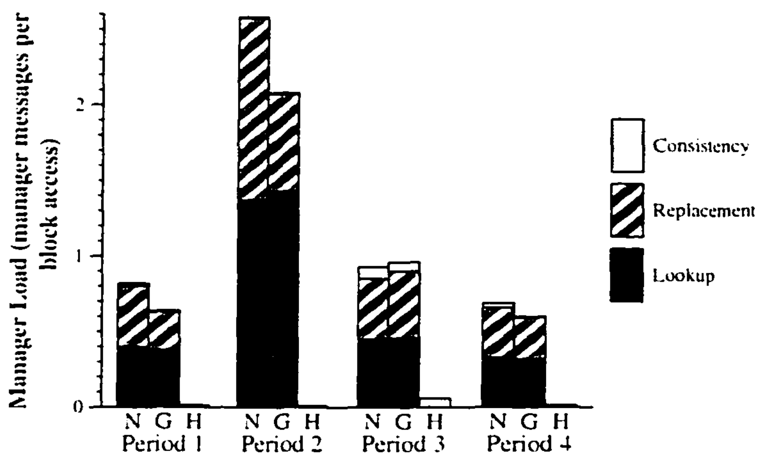


Figure 5.2: **Manager Load.** This shows the average manager load of the N-chance(N). GMS(G) and hint-based(H) algorithms in the Sprite traces. The manager load is defined as the number of messages received and sent by the manager per block access. The load is categorized by its cause: consistency. replacement and lookup.

The manager load is further broken down by the source of the load. *Consistency* messages are those required to keep block location information up-to-date in the client caches and this overhead is incurred by all the three algorithms. The *Lookup* and *Replacement* messages are those sent and received by the managers in the GMS and N-chance algorithms to lookup and replace blocks from the client caches. The hint-based algorithm does not incur any manager load for lookup and replacement as these decisions are taken by clients.

As can be seen, managing the client cache consistency imposes a very small load on the manager. While the choice of a consistency algorithm may affect performance. it does not contribute significantly to manager load. File-based consistency is still important for enabling the use of hints for replacement and lookup.

Period	1	2	3	4
Degree of Sharing	1.98	1.85	4.02	2.08
Invalidation Rate	0.008	0.005	0.035	0.01

Table 5.5: **Consistency Patterns.** The row *Degree of Sharing* refers to the average number of copies of a block in the client caches for each trace period. The row *Invalidation Rate* represents the average number of invalidations per block access in each trace period.

One further observation is that the consistency traffic per block access is highest in the third period. This is due to the higher degree of sharing and increased invalidation rate in this particular period compared to the others(Table 5.5). When a client obtains a write token for a file, the consistency manager revokes the tokens from the clients accessing the file and invalidates the file's blocks from their caches. The higher degree of sharing implies that relatively more clients are accessing the same file block than in other periods. This means that in the event of a write to the file, the consistency manager needs to contact more clients to invalidate the file's blocks than in other periods. Furthermore, since the rate of invalidation is also higher than in other periods, the amount of consistency traffic per block access is also higher.

Replacement and lookup traffic account for nearly all of the manager load for the N-chance and GMS algorithms. The clients must contact the manager each time a block is forwarded or lookup is done, whereas the hint-based algorithm allows clients to perform these functions themselves. The result is that the manager load is much higher for N-chance and GMS.

The replacement traffic is higher in N-chance than GMS in all the periods except for the third because of the poor performance of the N-chance algorithm as documented in Section 5.2.1. The increased number of disk accesses in these periods in N-chance compared to GMS also increases the number of replacements needed to make space for the blocks fetched from the disk, thereby resulting in replacement traffic that is higher than that in GMS.

5.2.4 Replacement Messages

Figure 5.2 showed that about half of the N-chance and GMS manager loads are caused by block replacement messages. Figure 5.3 depicts the number of replacement messages each algorithm requires to handle a local cache miss. This metric allows us to estimate the overhead associated with replacing a block in the client caches for each algorithm. N-chance and GMS have three sources of replacement messages: forwarding the block to another client and notifying the manager, notifying the manager when a block is deleted, and exchanging messages between the clients

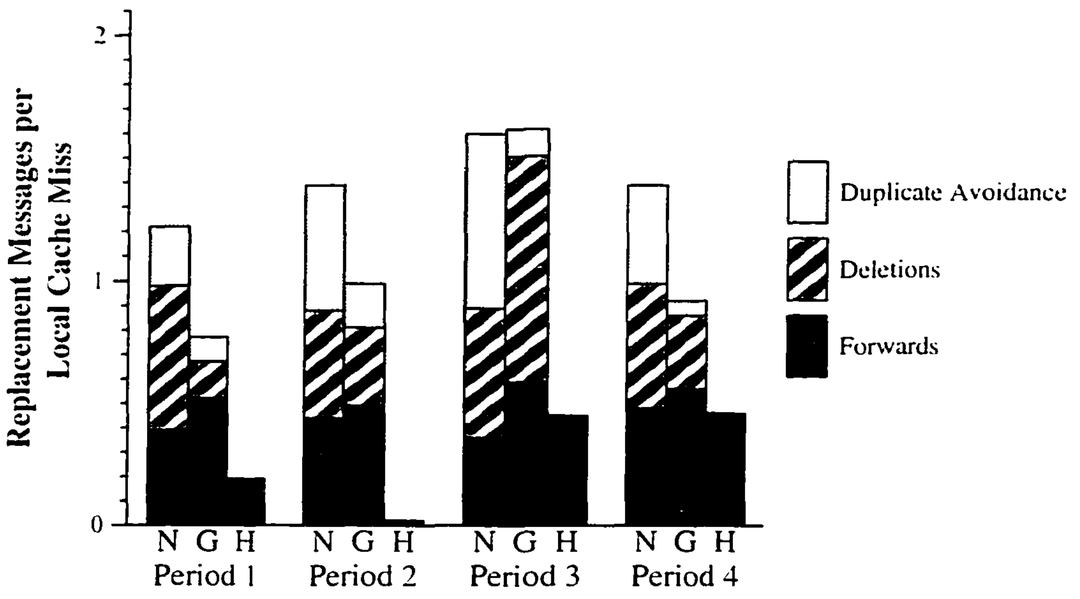


Figure 5.3: **Replacement Traffic.** This figure shows the number of replacement messages required per local cache miss in the N-chance(N), GMS(G) and hint-based algorithms(H). The messages are categorized by those required to forward a block, delete a block, and keep track of the number of copies of a block (duplicate avoidance). For N-chance and GMS, this includes two messages per singlet forwarded (one to forward the block and another to notify the manager), one message per block deleted, as well the messages required to keep track of the number of copies of a block. For the hint-based algorithm, this includes one message per master copy forwarded. Best-guess replacement does not need to exchange messages with a manager for either duplicate avoidance or deletion, and as a result, incurs no manager load for these functions.

and the manager to determine when a block should be discarded versus forwarded. Except for the actual forwarding of the block to another client, all messages involve the manager, increasing its load. For best-guess replacement, the only message required is the one to forward the master copy of a block to another client. Best-guess replacement does not need to exchange messages with a manager for either duplicate avoidance or deletion, and as a result, incurs no manager load for these functions. This dramatically reduces the total number of replacement messages required per local cache miss for the hint-based algorithm.

Further observing the relative replacement traffic between GMS and N-chance, the former algorithm sends a lower number of replacement messages per local cache miss. As discussed in Chapter 3.2.2, this is because GMS relies on the manager to inform a client only when a block becomes a singlet. while the N-chance algorithm relies on a client to contact the manager almost every time (98%) the client replaces a block.

One of the potential drawbacks of the master copy algorithm is that it may unnecessarily forward the master copy of a block to the cooperative cache as described in Chapter 4.3.1. Although Figure 5.3 shows that best-guess replacement outperformed the other algorithms. the fraction of forwards that are unnecessary were measured. An average of only 1.97% are unnecessary across all periods (Table 5.6).

Period	1	2	3	4	average
Unnecessary Forwards	2.28	1.92	3.93	1.58	1.97

Table 5.6: **Unnecessary Forwards.** The row *Unnecessary Forwards* refers to the percentage of forwards that are not necessary in the hint-based algorithm (i.e. forwards of master copy blocks which are deleted before they are down to their last copy). The column *average* refers to the average percentage of forwards that were unnecessary across all periods.

5.2.5 Network and Client Loads

The additional load on the network and clients due to cooperative caching is also important. Figures 5.4 and 5.5 show the average and maximum network load due to cooperative caching. Similarly. Figures 5.6 and 5.7 show the average and maximum load on a client due to cooperative caching. As is evident. the throughput requirement of cooperative caching on the network for all the algorithms is negligible at less than 4% of the available bandwidth of a 10 Mbps Ethernet network. Similarly. the throughput requirement on a client for all the algorithms is also negligible ($< 0.05\%$) when compared to the bandwidths of 20 MBps and greater that are available in standard I/O busses. On further inspection. the loads due to cooperative caching on both the network and clients are higher by a factor of about 20-30 in the existing algorithms than those in the hint-based algorithm (it is particularly evident in the second period). This implies that that the existing algorithms are more likely than the hint-based algorithm to swamp the network and disrupt client activity if

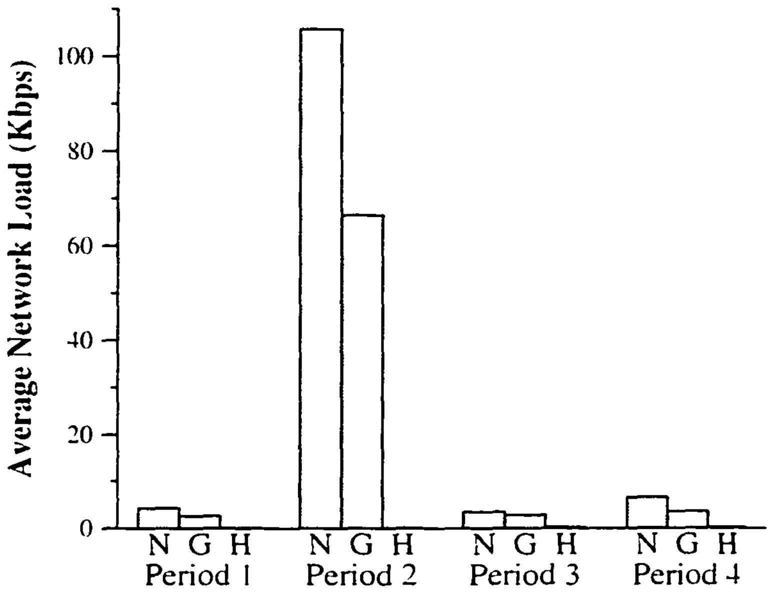


Figure 5.4: **Average Network Load.** This figure shows the average load on the network (Kbps) due to cooperative caching for the trace periods in the N-chance(N), GMS(G) and hint-based algorithms(H).

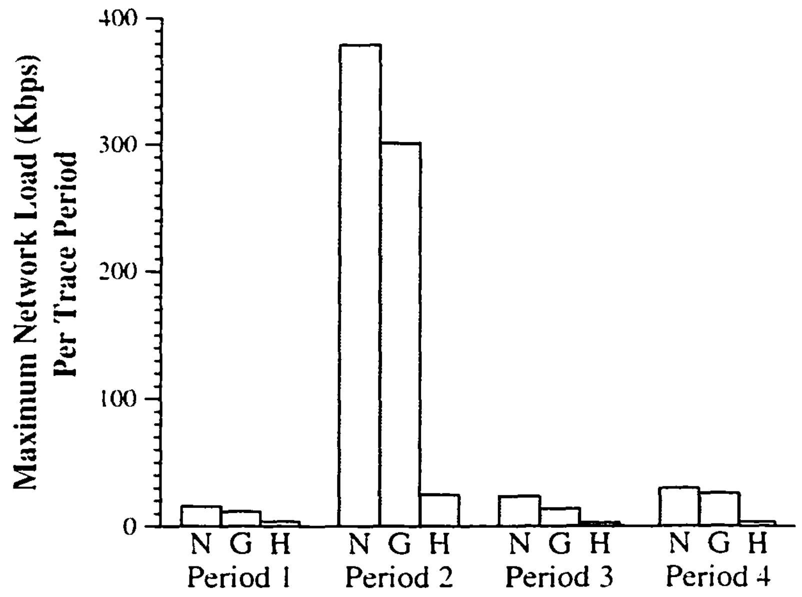


Figure 5.5: **Maximum Network Load.** This figure shows the maximum load on the network (Kbps) due to cooperative caching for each trace period in the N-chance(N), GMS(G) and hint-based algorithms(H).

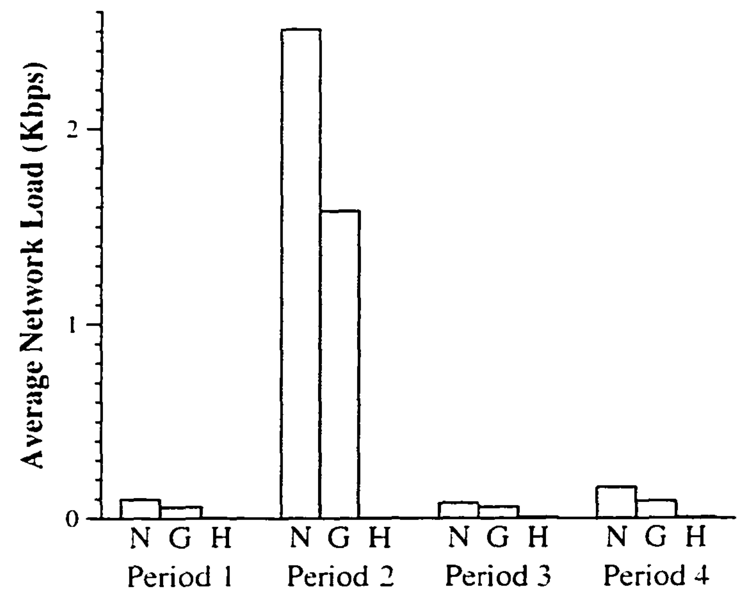


Figure 5.6: **Average Client Load.** This figure shows the average load on a client (Kbps) due to cooperative caching for the trace periods in the N-chance(N), GMS(G) and hint-based algorithms(H).

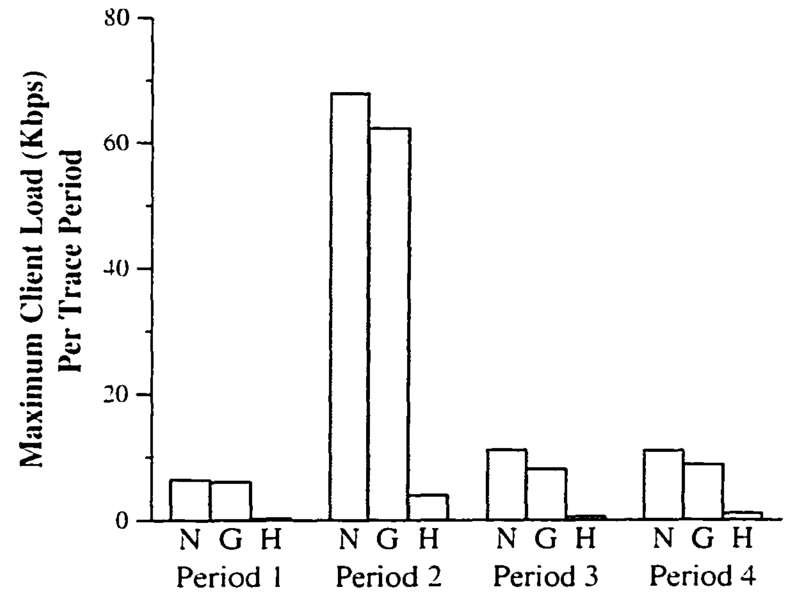


Figure 5.7: **Maximum Client Load.** This figure shows the maximum load on a client (Kbps) due to cooperative caching for each trace period in the N-chance(N), GMS(G) and hint-based algorithms(H).

the bandwidth requirement of applications starts to approach the available network bandwidth.

5.2.6 Discard Cache

Server Mem Use	**Hit Ratio(%)**					**Block Access Time(ms)**				
Period	1	2	3	4	Ave	1	2	3	4	Ave
Disk Cache	0.12	0.47	0.22	0.20	0.45	3.43	3.22	3.89	3.64	3.23
Coop Cache	1.16	1.96	0.98	1.35	1.83	2.86	2.79	3.22	3.12	2.77
Discard Cache	1.55	2.56	1.22	1.76	2.46	2.66	2.58	2.81	2.74	2.57

Table 5.7: **Server Memory Uses.** This table shows how different uses of the server memory affected the hit ratio of the server memory and the average block access time of the hint-based system. Server memory is used as either a traditional disk cache, as part of the cooperative cache, or as a discard cache. The results are shown for each of the trace periods as well as the average across all periods in the column *Ave*.

The server memory represents a valuable resource to the system and at 128 MB it constitutes a large fraction of the system's total memory. The hint-based system uses the server memory as a discard cache to mask mistakes made by the best-guess replacement policy. There are other possible uses for the server memory, including as a traditional server cache and as a portion of the cooperative cache, as discussed in Chapter 4.5.1.

The default 16 MB client cache size used in the simulations makes it difficult to measure the effectiveness of the discard cache. The aggregate client memory size greatly exceeds the working set size of applications and as a result, few accesses go to the server. Thus, to measure the effectiveness of the discard cache, the sizes of the client caches and server cache were reduced to 4 MB and 16 MB respectively, and we reran simulations of the hint-based algorithm with the different uses of the server memory over the same traces. Reducing the cache sizes makes the aggregate client memory size comparable to the working set requirements of client applications. These cache sizes increase the miss ratios on the local and cooperative caches, and therefore the number of server accesses.

The results are shown in Table 5.7 and indicate that when the server memory is used as a server cache, it has a very low hit ratio of 0.45% (averaged over all periods) because most of the blocks in the server memory are duplicated in the client caches. This results in an average block access time of 3.23 ms. If the server memory is instead used as part of the cooperative cache, old blocks with no duplicates are forwarded to the cooperative cache on the server memory and consequently, the hit ratio increases by nearly a factor of 4 to 1.83%, causing the block access time to drop to 2.77 ms. Using the server memory as a discard cache results in forwards of only young mistakenly replaced master copies to the discard cache, which further increases the hit ratio to 2.46% and reduces the block access time by nearly 10% to 2.57 ms.

5.2.7 Sensitivity

The analysis presented in the previous sections was based on a single system configuration, in which the number of clients, client cache size, number of servers, and other parameters were fixed. Although the hint-based algorithm performed well under the chosen configuration, its sensitivity to variations in the environment is also important. This section presents the sensitivity of the block access time and the manager load to two environmental variables: the client cache size and the fraction of the clients that actively use the cooperative cache. These changes allow us to simulate the effect of memory-intensive workloads by changing the aggregate size of the client memories relative to the working set requirements of client applications.

5.2.7.1 Client Cache Size Sensitivity

Figure 5.8 shows the average block access time across all the trace periods as the client cache size is varied from 4 MB to 16 MB. The remaining system parameters are the same as those shown in Table 5.2. A smaller client cache increases the load on cooperative caching in two ways: first, it increases the local cache miss ratios and therefore accesses to the caches of other clients; and second, it reduces the size of the cooperative cache. Even with 4 MB caches the algorithms do a good job of finding and using the available idle memory, producing block access times that are

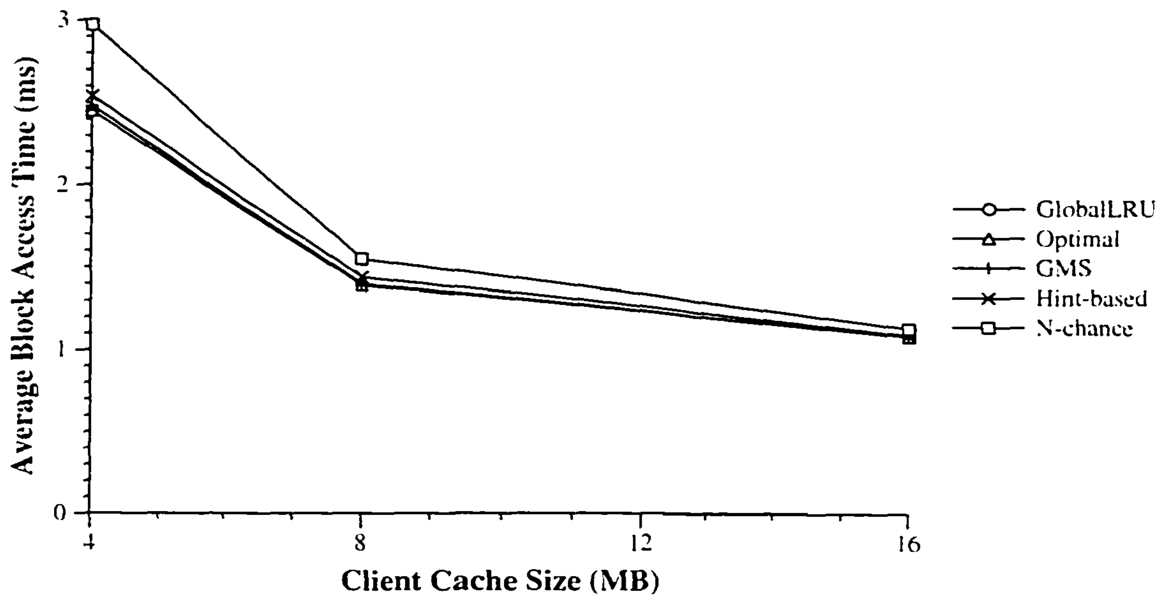


Figure 5.8: **Access Time vs. Cache Size.** This figure shows the average block access time for the algorithms as a function of the client cache size.

close to that of Optimal. The exception is the N-chance algorithm, whose policy of randomly forwarding blocks hurts performance when the working set size of client applications starts to approach the aggregate size of the client memories.

The average block access time of the hint-based algorithm is about 5% worse than that of the ideal algorithms when the client cache size is reduced to 4 MB. This deviation is expected to grow as the client cache size is reduced further, because block location hints tend to become increasingly inaccurate as the effective size of the cooperative cache is reduced compared to the working set requirements of client applications.

If we consider the manager load for the existing and hint-based algorithms, a different picture emerges. As Figure 5.9 indicates, the manager load for the existing fact-based algorithms almost quadruples when the cache size is reduced from 16 MB to 4 MB. The decrease in cache size increases accesses to remote client caches and the server, causing increased manager load in N-chance and GMS. In contrast, the manager load in the hint-based algorithm is lower by as much as 30 times than that in the existing algorithms because most cooperative cache decisions do not

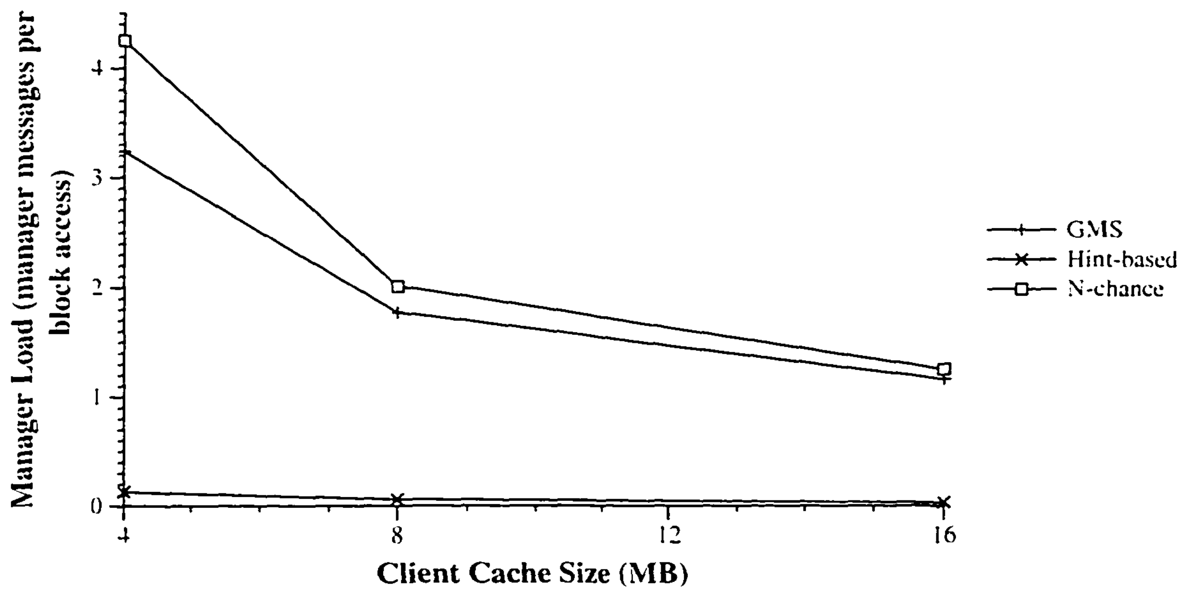


Figure 5.9: **Manager Load vs. Cache Size.** This figure shows the manager load (manager messages per block access) for the algorithms as a function of the client cache size.

involve the manager. Thus, while the average block access time of the hint-based algorithm diverges marginally (5%) from that of the ideal algorithms as cache sizes are reduced, the manager load is substantially lower (30 times) than those of the existing fact-based algorithms.

5.2.7.2 Active Client Sensitivity

The sensitivity of the block access time to the fraction of clients that are using the cooperative cache is also important. Increasing the fraction of clients that use the cooperative cache increases the demand for memory, and also decreases the effective cooperative cache size compared to the working set requirements of client applications. This combined effect increases the importance of managing the cooperative cache efficiently. Figure 5.10 shows the average block access times of the algorithms for the second period as the fraction of clients that used the cooperative cache was increased from 50% to 75% (doing this for the remaining periods without altering workload was difficult). As is evident, the block access time of the N-chance algorithm declines at a faster rate than that of the remaining algorithms as the fraction

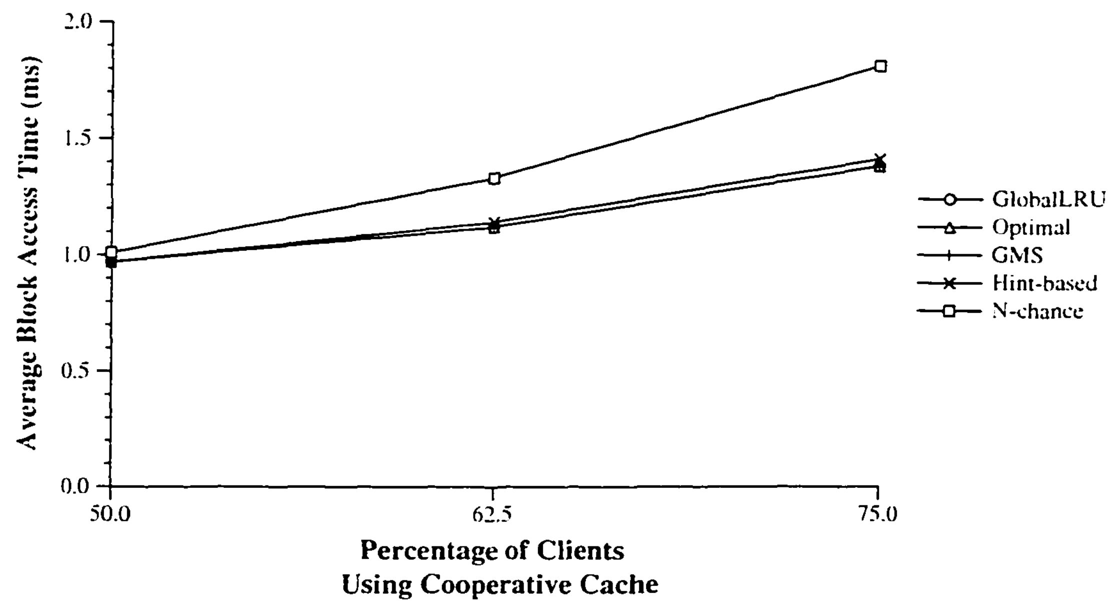


Figure 5.10: **Access Time vs. Active Clients.** This figure shows the average block access time for the algorithms as a function of the fraction of clients that used the cooperative cache during the period. The fraction of clients using the cooperative cache was varied by removing idle client trace records from the trace. Due to the difficulty in doing this without affecting the workload behavior, only the second period was used.

of clients using the cooperative cache increases. Again, this is due to the random forwarding of blocks to other clients in N-chance. The remaining algorithms all have block access times close to that of Optimal, while the hint-based algorithm shows the same marginal divergence from the ideal algorithms as in the previous sensitivity experiment. Similarly, the manager load for the hint-based algorithm is 30-50 times lower than that of the existing algorithms (Figure 5.11) because the increased traffic to the cooperative cache also increases the manager load in N-chance and GMS.

5.3 Summary

This chapter presents trace-driven simulation results which compare the performance and overhead of the proposed hint-based algorithm with those of the existing and ideal algorithms. The traces were obtained from applications running on a cluster of about 40 workstations and taken over four two-day periods.

The algorithms were measured using two separate criteria. The first was the

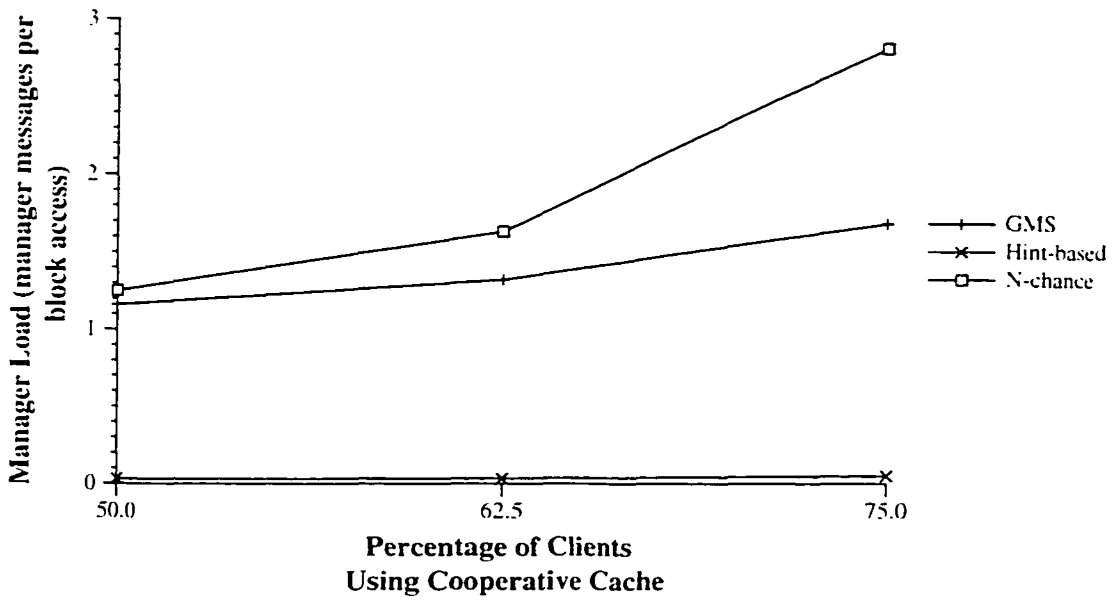


Figure 5.11: **Manager Load vs. Active Clients.** This figure shows the manager load (manager messages per block access) for the algorithms as a function of the fraction of clients that used the cooperative cache during the period. The fraction of clients using the cooperative cache was varied by removing idle client trace records from the trace. Due to the difficulty in doing this without affecting the workload behavior, only the second period was used.

---

average block access time, which measures how successful an algorithm is in increasing hits to the local and cooperative caches. The second criteria was overhead which measures the work done by the manager in terms of messages in managing the cooperative cache.

The important results from the simulation are summarized below:

- The block access times of the hint-based algorithm match those of the existing and ideal algorithms over all the four periods of the traces.
- Over all the periods, when a hint says that a block is present in the cooperative cache, it is correct for 99.94% of all local cache misses.
- The manager load in the hint-based algorithm was substantially lower (30 times) than that in the existing algorithms.

- The evaluation of the discard cache as a use for the server memory reveals that the hit ratio is the highest at 2.46% among all uses of server memory.

CHAPTER 6
IMPLEMENTATION

This chapter evaluates the hint-based algorithm by measuring a prototype's performance. The prototype allows us to test the algorithm with a workload generated by real users and observe the potential benefits. A prototype also enables us to validate the results obtained from the simulations presented in the previous chapter. Finally, the prototype gives insight into the complexity of implementing a hint-based algorithm.

The chapter presents the details of the prototype implementation, and then discusses the differences between the prototype and the hint-based algorithm. This is followed by a description of the measurements obtained from the prototype over the period of one week. The measurements include the client activity profile of workstations in the cluster, the average block access time as compared to NFS, the overhead in the cluster as well as the performance of the various design decisions in the hint-based algorithm.

6.1 Prototype

The prototype was implemented on a cluster of machines running Linux v2.0.23 [Beck96] and NFS v2[Sandberg85]. The Linux kernel was modified to support hint-based cooperative caching. The extensions to the Linux kernel added about 10 KB to a 390 KB kernel. The kernel extensions are discussed with respect to block lookup, replacement, cache consistency and discard caching. As mentioned above, clients and servers communicated using NFS, while inter-client communication used the SunRPC protocol[Sun88].

6.1.1 Block Lookup

The location mechanism in hint-based algorithm does not use features specific to any operating system and thus there was no need for modifications tailored to the Linux kernel.

The composition of a block location hint is shown in Table 6.1. The block location hints are indexed by the *Block Identifier*. The entry *Cache Location* points to the location of a block present in a client's cache. If the block is not present in the cache, the entry *Client* contains the identity of the client possibly having the block. Clients are identified by name and a machine uses the DNS naming service to resolve the client name to the correct network address. The entry *Source* contains the identity of the server or the client from which a block was obtained, thus enabling the client caching the block to determine if the block is a master copy. The entry *Boot Identifier* contains the boot identifier of the client referred to in the entry *Client*, and is used to determine whether the client was rebooted since the hint was obtained, as described in Section 6.1.1.2.

Block Identifier	*Cache Location*	*Client*	*Source*	*Boot Identifier*

Table 6.1: **Block Location Hints.** The table describes the format of the block location hints. Further explanation is available in Section 6.1.1.

Each client maintains a cache of these block location hints. These hints are obtained from other clients when a client opens a file, as described in Section 4.2.1. Block location hints are deleted if the client is informed that the block referred to by the hint has been deleted or invalidated. However, the hint cache grows in size because the rate at which new blocks are referenced by clients is faster than the rate at which existing blocks are invalidated or deleted. Consequently, the hint cache is pruned using LRU when the size of the cache exceeds a given threshold. This threshold is an operating system parameter that can be changed at runtime.

The remaining implementation issues were uniquely identifying a block in the client caches, and dealing with occasional client reboots and component failures.

6.1.1.1 Block Identification

In a cooperative-caching file system, clients provide blocks to other clients from their caches. As servers in a distributed file system might have overlapping block address spaces, it is important that a request to a client for a block must uniquely identify the block in the distributed file system. Complicating this fact is that a Linux NFS client indexes a block in its file cache using the memory address of the inode of the block's file. As the memory address of the inode of a file is not unique across clients, there needs to be a different indexing mechanism to locate a block in the client caches.

The block identifier used in the location hints relies on the triplet *(Server, File Identifier, Offset)* to uniquely identify a block in the client caches. As in clients, the server is identified by its name and the client uses the DNS service to resolve the name to the correct network address. This triplet is sufficient as a block from a particular server is uniquely identified in NFS by the identifier of the file to which the block belongs and the offset of the block within the file.

The sequence of steps for a cooperative cache lookup in Linux is shown in Figure 6.1. Whenever a Linux application needs a block, the kernel calls *readpage* to check if the block is in the cache. If the block is not present in the cache and belongs to a NFS-mounted filesystem, the *readpage* routine calls *nfs_readpage* . This routine then checks the hint cache to determine where the block is located in the cooperative cache, and sends a request for the block to the appropriate client. The client receives this request for a cooperative cache block using a *nfsd* daemon thread. This thread then checks with the hint cache to get the location of the block in the client's cache. If the receiving client has the block in its file cache, the block is sent to the requesting client. If the receiving client does not have the block, the hint cache may contain the identity of another client possibly having the block. In this case, the receiving client forwards the request for the block to this new client. Otherwise, the receiving client forwards the request for the block to the server.

There is no limit to the number of times a request can be forwarded from one

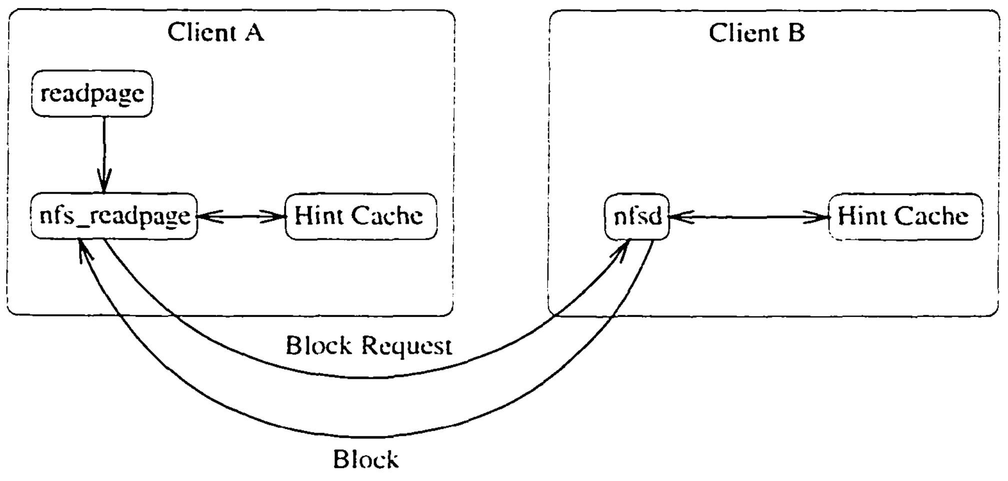


Figure 6.1: **Block Lookup in Linux.** The figure shows the sequence of kernel routines called in Linux to lookup a block in the cooperative cache.

client to another. However, it is conceivable that the forwarded requests could add to the network load when there is a lot of block movement in and out of the client caches. One possibility is to limit the forwarding of requests between clients whenever the network load exceeds a threshold, directing forwarded requests to the server from then on. However, this needs to be investigated further and is a matter of future study.

6.1.1.2 Client Reboots

Clients are occasionally rebooted, clearing their caches. This creates incorrect hints that refer to the now-empty cache, and lowers the accuracy of hints in other clients. It is important to incorporate the effect of rebooting into the design of a cooperative-caching system so that hint accuracy is not adversely affected.

The incorrect hints are largely eliminated by incorporating a boot identifier in each client which is incremented whenever a client is rebooted. A block location hint is tagged with the boot identifier of the client to which the hint refers. When a client reboots, its boot identifier changes. Other clients learn this new boot identifier when they request block location hints from the rebooted client. To eliminate an incorrect block location hint, a client checks if the boot identifier of the client referred to by the hint is the most current one and discards the hint if the test fails. As the propagation of block identifiers is piggybacked on existing traffic. this mechanism imposes minimal overhead on block lookup.

6.1.1.3 Client and Manager Failure

The lookup algorithm uses a lazy approach to handle client failure. A client detects that another client has failed when messages sent to the client are not acknowledged within a specified timeout period. If a client detects that another client has failed. then the client assumes that the failed client is unavailable and ignores all location and replacement hints that refer to the failed client. Once the failed client comes back up. it contacts the manager which provides the client with a list of clients in the distributed file system. The client then gradually contacts the clients on the list as directed by its location and replacement hints. The contacted clients then resume communication with this previously-failed client. The disadvantage of this lazy policy is that a client may remain unaware that a failed client is back up due to lack of contact. A heartbeat protocol would remedy this disadvantage but the overhead would be higher.

The lookup algorithm is also the only part of the hint-based cooperative caching algorithm that deals with a manager. Clients deal with manager failure similarly to the way they deal with client failure. A client detects that a manager has failed when requests for hints are not acknowledged within a specified timeout period. When a client detects that a manager has failed. the client stops contacting the manager. On being restarted, the manager contacts all clients in the list of clients in the distributed file system. The contacted clients then resume communication with the manager.

6.1.2 Replacement

There were two minor differences between the implementation and the hint-based algorithm. The first was that the discussion of the replacement algorithm in Chapter 4.3 assumed LRU as the replacement policy for a client's local file cache. In contrast, Linux uses Global Clock (described in Section 2.5.1) rather than LRU to choose blocks for replacement. While LRU discards the block which has been referenced the least recently from a cache, Global Clock chooses one of a set of candidate blocks which have not been referenced recently and thus does not follow an exact LRU order in replacing blocks from a cache. Hence, the results obtained from the implementation may not be identical to that from the simulation. The effect of not using an exact LRU order in Global Clock is hard to predict without detailed workload information. However both theory and practice suggest that the extent of deviation of Global Clock from LRU is minimal[Easton79, Voelker97].

The second difference between the simulation and the implementation is the use of a threshold to trigger replacement in Linux. In the simulation, a client triggers the replacement policy when its file cache is full and the client needs to make space for a new block. In contrast, Linux triggers the *kswapd* daemon when the number of free blocks in its file cache falls below a threshold *free_pages_low*; the daemon then replaces blocks from its cache until the number of free blocks exceeds the threshold *free_pages_high*.. This difference does not affect the order in which blocks are replaced from a client's cache, but does affect the time of replacement. As a result, this difference should not have a significant effect on performance.

The replacement algorithm in hint-based cooperative caching is independent of manager failures and deals with client failures as described in Section 6.1.1.3.

6.1.3 Consistency

As cooperative caching can work with any consistency mechanism, integrating the hint-based cooperative-caching algorithm with NFS was not a problem. NFS adopts a client-driven consistency where the clients keep their caches up-to-date by periodically checking with the NFS server. While this means that the client caches can be temporarily inconsistent, the implication for cooperative caching is that clients can sometimes get stale blocks from other clients, as discussed in Chapter 2.6.1.. Consequently, the number of accesses to stale copies of a block will be higher than that in the simulation which uses strong consistency.

The granularity of consistency in NFS is also suited for hint-based cooperative caching. NFS keeps consistency in terms of files, which is ideal for the use of hints as clients do not need to frequently communicate with a manager to ensure consistency. On the other hand, it would be difficult to incorporate the N-chance algorithm into NFS as N-chance requires block-based consistency.

6.1.4 Discard Cache

The discard cache could not be located in the server memory because of the constraints imposed by the testbed. The server ran a proprietary operating system and there was no license to modify the source code to incorporate the discard cache. In addition, the server provided uninterruptible file service for instructional purposes, as a result of which there was no possibility of kernel development in the server. Similarly, there was no possibility of using the server as part of the cooperative cache.

However, the utility of the discard cache was extrapolated by designating an idle machine to serve as the host for an independent discard cache. The number of forwards to the discard cache and the hit rate on the discard cache was monitored to measure its effectiveness. An independent discard cache incurs extra overhead as clients may have to check both the discard cache and the server in sequence to locate a mistakenly replaced master copy, while a discard cache in the server memory would require only one check with the server to locate the block. The extra overhead caused by an independent discard cache depends on the number of unsuccessful accesses to the discard cache, but fortunately the results documented in Section 6.4.6 indicate that this traffic was negligible.

6.1.5 Miscellaneous

Finally, as the server ran a proprietary operating system and did not provide the necessary statistics, it was not possible to measure server cache hits. As a result, server accesses were not segregated into cache and disk accesses. Another implication of this was the inability to compare a traditional server cache with the discard cache.

6.2 Experimental Setup

The measurement of the prototype was done on a cluster of 8 Pentium client workstations over the period of one week. Each workstation was a 200 MHz Pentium Pro running Linux and NFS, connected by 100 Mbps switched Ethernet. The workstations were located on the desktops of faculty and students. The workstations *rosewood, pelican, blackoak, delphin, carta* and *omega* were used by students and had 64 MB of memory. The workstations *roadrunner* and *cicada* were used by faculty and had 128 MB of memory.

The server was a Network Appliance F520 machine with 128 MB of memory and 4 MB of NVRAM[Hitz94]. All the data files and most of the Linux binaries were stored in the server. The remaining Linux binaries were stored in the local disk of each workstation because they were required during bootup time. Typical applications run on the cluster ranged from word processing, editing, operating system development and compiler benchmarking[Mosberger96, Proebsting97]. There was a single manager running on *rosewood* which coordinated the cooperative cache.

Local Memory Latency	0.1 ms
Remote Memory Latency	0.5 ms
Server Access Latency	12 ms
Forward Latency	0.5 ms
Block Size	4 KB

Table 6.2: **Experimental Setup Parameters.** This table lists the average access times and block size in the experimental setup. The average times to fetch a 4 KB block in the local cache, the cache of another client and the server are shown in the rows *Local Memory Latency, Remote Memory Latency* and *Server Access Latency.* The average roundtrip time for forwarding a block using kernel-level SunRPC is shown in the row *Forward Latency.*

The block size and average access times are shown in Table 6.2. The average times to fetch a 4 KB block in the local cache, the cache of another client and the server are shown in the rows *Local Memory Latency, Remote Memory Latency* and *Server Access Latency.* These latency values were the average to fetch 10,000 blocks from each layer of the storage hierarchy. The server access latency was measured repeatedly to even out differences at different times of the day and week.

Measurements of individual accesses could not be obtained as the Linux kernel did not support a microsecond timer. The average roundtrip time for forwarding a block using kernel-level SunRPC was also measured by taking the average for 10.000 messages and is shown in the row *Forward Latency*.

6.3 Methodology

Measurements were collected on every workstation at 15 minute intervals. The measurements included the hits to the layers of the storage hierarchy. forwards to the cooperative cache. forwarded block location requests. deletes. hint requests. hint accuracy among others.

The criteria used to evaluate the hint-based cooperative caching file system is similar to that used in the simulation. First. the benefit of hint-based cooperative caching was measured by estimating the average block access time with and without cooperative caching. To measure whether hint-based cooperative caching disrupted the network and client activity. the overhead of maintaining hints in the distributed file system was also monitored. Finally. each of the various design decisions taken in hint-based cooperative caching was individually evaluated. The utility of the discard cache was extrapolated by designating an idle machine to serve as the host for an independent discard cache.

6.4 Prototype Measurements

6.4.1 Client Activity Profiles

To understand better the performance of the file system. it is important to get an idea of the activity profile of every client. The average composition of each client's cache is shown in Figure 6.2. There are two observations to be made. First. the ratio between the number of local and global blocks depended on the level of activity of clients. Highly active clients such as *carta* had a larger fraction of local blocks and a smaller fraction of global blocks. By the same token. mostly idle clients such as *cicada* had a correspondingly larger fraction of global blocks and a smaller fraction

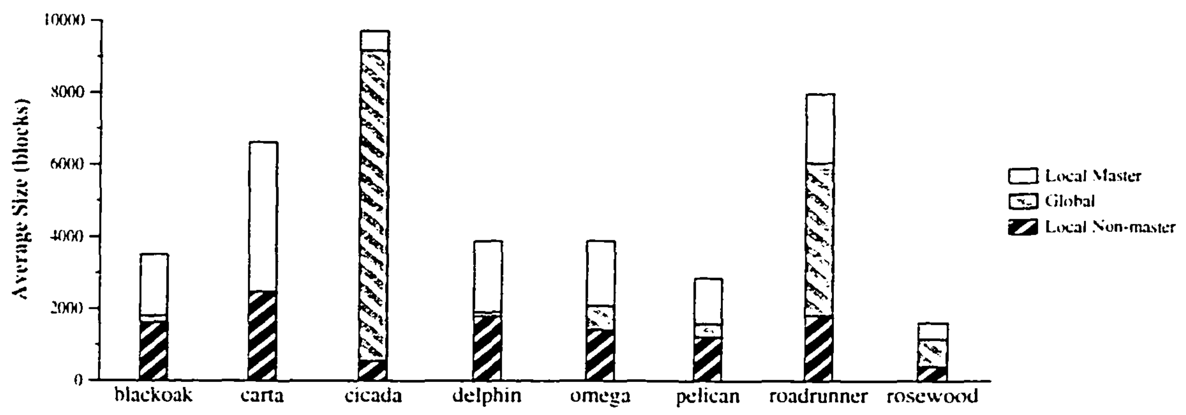


Figure 6.2: **Client Cache Composition.** This figure shows the composition of the caches in the clients. The legends *Local Master*, *Global* and *Local Non-master* indicate the average number of local master copy blocks, global blocks, and local non-master copy blocks in each client.

of local blocks.

Second, the average percentage of local non-master copy blocks (local blocks fetched from other clients instead of the server) in the client caches was around 27.5%. This percentage is comparable to the average percentage of local master copy blocks (local blocks fetched directly from the server) which is 33.6%. This indicates that a substantial portion of the local blocks were obtained from other clients instead of the server. This implies that clients were able to use the cooperative cache effectively as a result of sharing or using idle memory in the distributed file system.

6.4.2 Benefit of Hint-based Cooperative Caching

To measure the benefit of hint-based cooperative caching, the average block access time was estimated with and without cooperative caching. To compensate for the lack of support for a microsecond timer in the Linux kernel, the average block access time was calculated from the distribution of reads to the various layers of the storage hierarchy and the average times needed to access each layer. The average block access time with cooperative caching includes the time due to extra remote client accesses caused by forwarded requests.

Measuring the average block access time without cooperative caching is complicated because it must be estimated from the prototype measurements. The straightforward way to estimate the average block access time without cooperative caching is to replace the remote cache accesses with server accesses. However, this ignores the fact that the cooperative cache can reduce the size of the local cache and in turn, the local cache hit ratio. Since it is difficult to measure the extent to which the local cache hit ratio is affected by the cooperative cache, the worst case assumption was taken that each forward to a client replaced a local block and therefore decreased the number of local cache hits by one, causing an extra server access. Thus if the number of local blocks in a client decreased over a time period, each forward to the client in that time period was assumed to be responsible for replacing a local block and therefore reduced the number of local cache hits by one, causing an extra server access in that time period.

To illustrate the above, assume the following variables for a particular time period i at a client in the distributed file system:

l_{i-1}	$=$	number of local blocks at the end of time period $i-1$
l_i	$=$	number of local blocks at the end of time period i
m_i	$=$	number of local blocks fetched during time period i
e_i	$=$	expected number of local blocks at the end of time period i (without taking replacements into account)
f_i	$=$	the number of blocks forwarded to the client during time period i

Then if replacements are not taken into account, the expected number of local blocks at the end of time period i is the sum of the number of local blocks fetched during time period i and the number of local blocks present at the end of time period $i-1$:

$$e_i = l_{i-1} + m_i$$

Then if x_i is the reduction in the number of local blocks in time period i:

$$x_i = \begin{cases} e_i - l_i & \text{if } e_i > l_i \\ 0 & \text{otherwise} \end{cases}$$

Then, the reduction in the number of local blocks due to replacements caused by forwarded global blocks in time period i is given by d_i:

$d_i = min(x_i, f_i)$

This reduction in the number of local blocks due to cooperative caching (i.e. forwarded global blocks) in time period i is used to compute the number of local cache hits and server hits in time period i if no cooperative caching is used. If l_i, c_i, s_i represent the number of local cache hits, cooperative cache hits and server hits in time period i when cooperative caching is used, then the number of local cache hits and server hits in time period i without cooperative caching is given by l'_i and s'_i:

$l'_i = l_i + d_i$

$s'_i = s_i + c_i - d_i$

Using the above extrapolation, Table 6.3 shows that the the average block access time for all machines with cooperative caching was 1.01 ms, which was 85% faster than the average block access time without cooperative caching. This estimation used worst case assumptions; on the other hand if an optimistic assumption was made that cooperative caching did not affect the local cache hit ratios, then the average block access time with cooperative caching would be 90% faster than the average block access time without cooperative caching. In reality, the percentage difference between the average block access times with and without cooperative caching would be somewhere in between the optimistic and pessimistic analysis figures. Nevertheless, this difference was mainly due to the fact that almost half of all local cache misses hit in the caches of other clients, reducing server accesses by the same amount. The result corroborates previous results[Dahlin94] and shows that cooperative caching can reduce the average block access time in NFS by nearly a factor of two.

6.4.3 Overhead

The effect of hint-based cooperative caching on the network and client activity is a concern. The effect is primarily due to the overhead messages to manage the cooperative cache which include hint requests to managers and clients, the extra accesses due to incorrect block location hints, and the forwards to the cooperative cache. Another concern is the overhead imposed on a client due to servicing requests

Machine	Cooperative Caching				
	Local Hits [Hit Ratio(%)]	Remote Hits [Hit Ratio(%)]	Server Hits	Access Time (ms)	Improv- -ement (%)
blackoak	416183[78]	59250[50]	58933	1.46	78.09
carta	1474668[87]	116199[52]	106141	0.87	86.90
cicada	58094[97]	1076[52]	991	0.30	61.72
delphin	715714[85]	61692[50]	61270	1.00	85.49
omega	339191[87]	22517[43]	29210	1.01	86.77
pelican	60808[67]	12873[43]	16826	2.37	71.23
roadrunner	187806[87]	9194[32]	19676	1.20	33.46
rosewood	146402[94]	5737[63]	3313	0.37	89.40
total	3398866[85]	288538[49]	296360	1.01	85.32

Machine	No Cooperative Caching			
	Local Hits [Hit Ratio(%)]	Remote Hits [Hit Ratio(%)]	Server Hits	Access Time (ms)
blackoak	417278[78]	-	117088	2.71
carta	1474693[87]	-	22315	1.66
cicada	58186[97]	-	1975	0.49
delphin	716956[86]	-	121720	1.83
omega	340500[87]	-	50418	1.63
pelican	64472[71]	-	26035	3.52
roadrunner	189387[87]	-	27289	1.60
rosewood	147689[95]	-	7813	0.70
total	3407065[86]	-	576699	1.82

Table 6.3: **Block Read Access Time.** This table shows the estimated block access time with and without cooperative caching. The columns *Local Hits*, *Remote Hits* and *Server Hits* indicate the number of references to the local cache, the caches of other clients and the server cache in the cases with and without cooperative caching. The column *Access Time* contains the average block access time with and without cooperative caching. The column *Improvement* refers to the percentage improvement obtained in the average block access time if cooperative caching is added to NFS. The row *total* refers to the cumulative behavior of all clients.

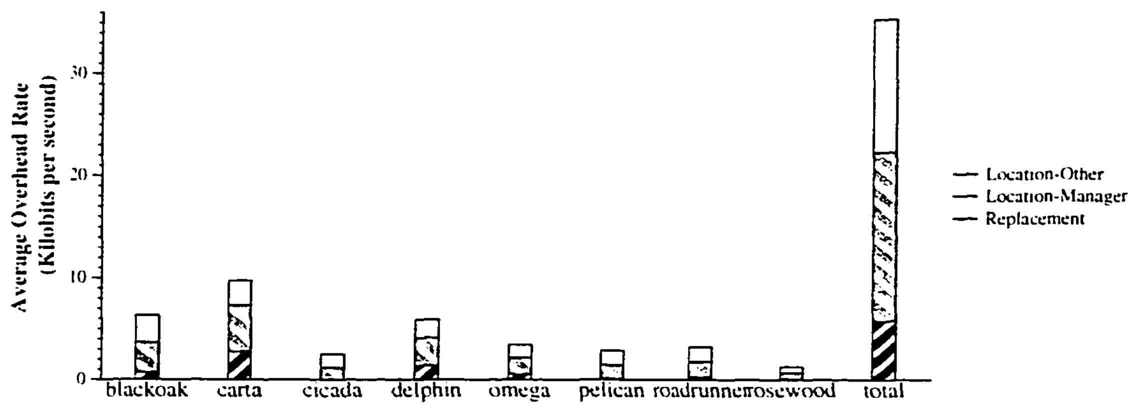


Figure 6.3: **Average Rate of Overhead Messages.** This figure shows the average rate of overhead messages processed by all the clients. The legends *Location* and *Replacement* refer to the overhead due to block lookup and replacement traffic. The sub-category of location overhead that involves communication with the manager is given by the legend *Location-Manager*. The remainder of the location overhead is shown by the legend *Location-Other*. The column *total* refers to the cumulative average overhead of all clients.

for cache blocks.

Figure 6.3 shows the average rate of overhead messages processed by the clients. Note that there is no overhead due to consistency because the manager does not enforce strong consistency and allows files to become temporarily inconsistent according to NFS conventions. Figure 6.4 documents the maximum rate of overhead messages processed by the clients during one week.

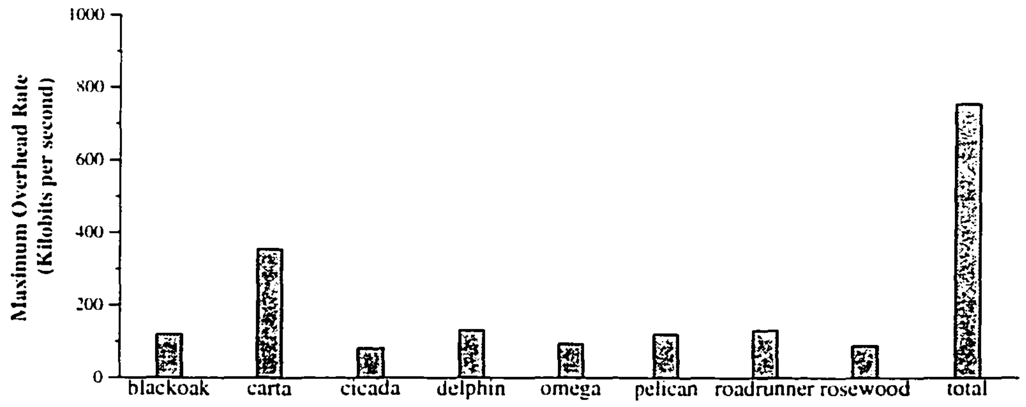


Figure 6.4: **Maximum Rate of Overhead Messages.** This figure shows the maximum rate of overhead messages processed by all the clients during one week. The column *total* refers to the cumulative maximum overhead of all clients.

The conclusion is that the rate of overhead messages in any client was not significant. The average overhead message size was 2,218 bytes, and the cumulative average and maximum throughput requirement was 35 and 755 Kbps. Even on a 10 Mbps shared Ethernet network, the maximum throughput requirement represents a small percentage (7.5%) of the available network bandwidth.

Further measurements revealed the possibility of reducing the number of location messages if the functionality of the manager is incorporated into the NFS server. The number of hint requests to clients (518,696) was half the number of messages sent to the manager (1,003,742). This is because a client contacts the manager on every file open even if the client was the last to the open the file and had the most current location hints related to the file. The number of messages could be reduced if the functionality of the manager is incorporated into the NFS server. Whenever a NFS client contacts the server to open a file, there would be no need for an extra message to the manager.

Finally, the overhead rate of servicing block requests in the clients was measured (Figure 6.5). The measurements indicated that block service requests to clients imposed of an average throughput requirement of 2.25 ± 0.5 Kbps on the clients, negligible ($< 0.01\%$) compared to the bandwidths of 20 MBps and greater found in standard I/O busses. Even the maximum throughput requirement recorded in a client at 738 Kbps is a mere 0.5% of the bandwidths of such I/O busses. The disruptions observed in GMS[Voelker97] were not seen because clients with idle memory were largely inactive (Figure 6.2). In contrast, GMS clients with idle memory acted as remote memory servers to applications running outside the GMS cluster and thereby incurred heavier load.

6.4.4 Block Location Hints

To evaluate the accuracy of block location hints, we tried to ascertain whether hint-based cooperative caching was able to locate blocks present in the caches of other clients (Table 6.4). As can be seen, on an average, hints correctly determined that a block was in the client caches in about 98% of the local cache misses. Of these

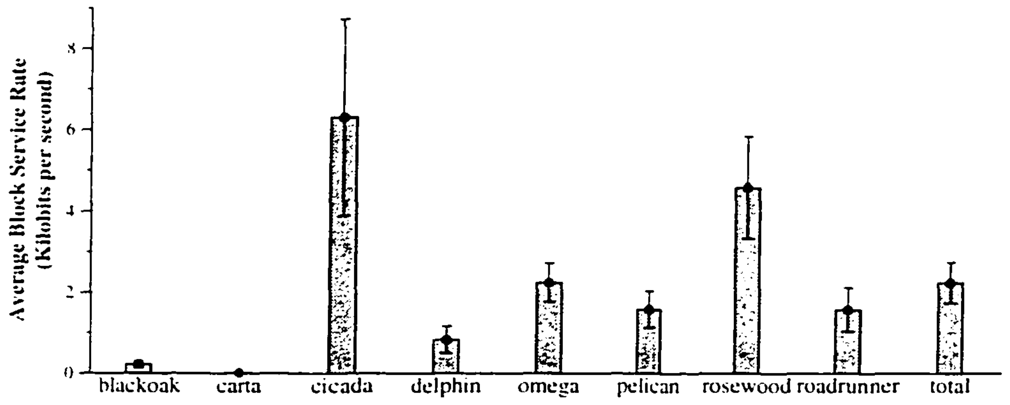


Figure 6.5: **Average Rate of Block Service.** This figure shows the average rate of block service requests (Kbps) processed by the clients during one week. The error bars show the standard deviation of the distribution of block service requests to each client. The column *total* refers to the average overhead of all clients.

correct hints, 97.72% pointed to the actual location of the block in the client caches. Also, in only 0.23% of the local cache misses was the block in the cooperative cache but the hints said otherwise. This result is almost identical to that obtained from the simulation.

The results also showed that tagging block location hints with client boot identifiers solved the problem of reboots. To test this hypothesis, experiments were done without using client boot identifiers. These results showed that with a similar rate of rebooting, the block location hint accuracy dropped to about 80%.

As a final measure, the average number of messages required for block lookup was found to be 2.002 ± 0.00135, demonstrating that lookup requests were rarely forwarded.

6.4.5 Best-guess Replacement

To investigate the performance of best-guess replacement, the measurements focused on whether or not best-guess replacement was removing the least valuable blocks from the client caches.

One way to do this is to compare the age of blocks replaced from the client caches to the ages of blocks present in the client caches. However, as Linux does

Workstation	Hint Correctness (%)	Absolute Correctness (%)	False Negatives (%)
blackoak	98.71	97.86	0.25
carta	98.57	97.98	0.22
cicada	96.87	95.51	0.15
delphin	97.76	96.18	0.37
omega	97.48	95.79	0.55
pelican	97.44	93.35	0.36
roadrunner	98.62	97.01	0.26
rosewood	96.48	92.79	0.39
total	98.23	87.72	0.23

Table 6.4: **Accuracy of Block Location Hints.** The row *Hint Correctness* refers to the percentage of local cache misses where block location hints correctly determine that the block is in the client caches. The column *Absolute Correctness* represents the percentage of correct block location hints that point to the actual location of the block. The column *False Negatives* represents the percentage of local cache misses when the block is in the client caches but the hints say otherwise. The row *total* refers to the combined hint accuracy of all clients.

not maintain the age of blocks in its file cache, this was not possible.

The accuracy of best-guess replacement was evaluated by monitoring the activity of clients which were the targets of high rates of forwards from other clients. The activity of a client was estimated based on the number of forwards from that client, as replacing blocks from the client caches is a definite sign of activity. The *activity forwards* of a client are the number of forwards from a client during the time when the client itself is the target of high rates of forwards from other clients. If the client has a relatively high number of activity forwards, it means that the client was busy when it was the target of forwards from other clients. In contrast, a relatively low number of activity forwards indicates that the client was idle when it was the target of forwards from other clients. To represent this better, the *forward ratio* of a client is defined as the ratio of the activity forwards of the client to the total number of forwards from the client. A small forward ratio indicates that the client was idle when there was a high rate of forwards to that client, implying high accuracy in best-guess replacement. On the other hand, if best-guess replacement

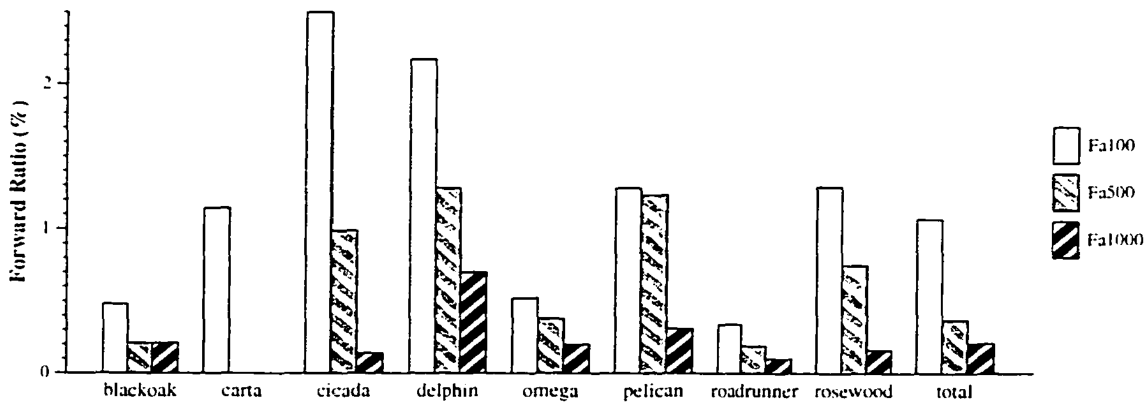


Figure 6.6: **Accuracy of Best-Guess Replacement.** This figure demonstrates the accuracy of best-guess replacement. The *forward ratio* of a client is the ratio of the number of forwards during the time when the client is the target of high rates of forwards to the total number of forwards from the client. The legends *Fa100*, *Fa500* and *Fa1000* represent the forward ratio of each client when the client was the target of forwards at rates greater than 100, 500 and 1000 per hour respectively. The column *total* refers to the combined forward ratios of all clients.

was making mistakes in replacing the least valuable blocks from the client caches, then the forward ratio would be high as active clients would erroneously become the target of forwards.

Figure 6.6 shows the forward ratio of each client during the experiment. Since the definition of high rates of forwards to a client is subjective, a high rate is defined as one greater than the following spectrum of forward rates to a client: 100, 500 and 1000 per hour. While these rates of forwards to a client are low ($<$ 1 forward/second) and therefore unlikely to cause disruption in the client, these rates are chosen conservatively to reveal whether best-guess replacement was making even the slightest mistakes in forwarding blocks to active clients. A higher threshold for the forward rate to a client makes it less likely the client was actively forwarding blocks during the period of high rate of forwards and consequently, the forward ratio decreases as the threshold rate increases.

Overall, the forward ratio of all clients was low at 0.2-1.1% during the periods of high rates of forwards. This means that clients were rarely busy when they were the target of high rates of forwards and best-guess replacement was doing a good job of selecting the target client. In fact, the maximum observed forward ratio was only 2.5%, assuming a forward rate of 100 per hour or greater.

6.4.6 Discard Cache

Using the client cache sizes in Figure 6.2, the average sizes of the client caches were summed up to determine the average working set size of all the applications running in the cluster. While this analysis optimistically assumed that the entirety of the client caches was being accessed, the analysis gave an upper bound on the average working set size. The aggregate size of the client memories (640 MB) was found to exceed the average working set size of all the applications(150 MB). This led to the hypothesis that the discard cache was unlikely to be useful in such circumstances. To test this hypothesis, the number of replacements, replacement errors (replacements of master copy blocks in a client which are younger than any block in the oldest block list of the client) and hits to the discard cache were measured.

Workstation	Replace- -ments	Replace- -ment Errors	Error Ratio (%)	Discard Hits	Discard Hit Ratio (%)
blackoak	5560	33	0.59	6	0.01
carta	534	8	1.48	0	0.00
cicada	155403	868	0.56	0	0.00
delphin	20554	101	0.49	0	0.00
omega	55203	268	0.48	5	0.01
pelican	39133	902	2.25	11	0.08
roadrunner	112896	847	0.74	6	0.06
rosewood	38937	407	1.03	7	0.12
total	428220	3434	0.80	35	0.01

Table 6.5: **Utility of Discard Cache.** The rows *Replacements, Replacement Errors, Error Ratio, Discard Hit* and *Discard Hit Ratio* refer to the number of replacements, the number of blocks forwarded to the discard cache as a result of replacement errors, the percentage ratio of replacement errors to the total number of replacements, the number of hits on the discard cache and the percentage ratio of the number of discard cache hits to the number of remote cache hits respectively on each client. The column ***total*** refers to the combined discard cache use by all clients.

As seen in Table 6.5, the number of replacement errors was less than 1% of the total number of replacements, further indicating that best-guess replacement does a good job of choosing the target clients for replacement. Moreover, the hit rate on the discard cache was very low at 0.01% implying that there was no performance benefit by adding a discard cache to the experimental setup. However, as the simulation results have shown, the discard cache can become useful if the working set size of client applications starts to approach the aggregate size of the client memories.

6.5 Summary

This section describes a prototype implementation of the hint-based cooperative caching system on a cluster of workstations running Linux and NFS. The prototype allows the evaluation of the algorithm on a real user workload and the validation of the simulation results.

As the hint-based algorithm did not rely on features specific to any particular operating system, implementing the algorithm on the cluster posed no problems, except for some minor implementation issues:

- The chief issue in block lookup was to design an indexing mechanism to allow Linux clients to uniquely identify blocks in the distributed file system.
- Linux uses Global Clock for replacing blocks from the local cache while the simulations assumed LRU, resulting in performance deviations.

The implementation was based on a cluster of 8 Pentium Pro workstations connected by 100 Mbps switched Ethernet and served by a Network Appliance F520 machine. The important results obtained from the prototype measurements were as follows:

- The average block access time with cooperative caching was almost half that of NFS.
- The average and maximum overhead rate in the hint-based cooperative caching system was negligible ($< 1\%$) compared to the available bandwidth of even a 10 Mbps Ethernet network.

CHAPTER 7
RELATED WORK

This chapter presents work related to cooperative caching in distributed file systems. The first section discusses efforts to harness remote memory in virtual memory systems. databases and multiprocessors. The final section focuses on the use of hints in the design of computer systems with examples from the field of operating systems. computer architecture. networks and programming languages.

7.1 Remote Memory Use

7.1.1 Virtual Memory

Research on remote memory use grew out of the desire to use memory as a backing store for virtual memory systems. Traditional virtual memory systems use disks as a backing store. As disk performance lags that of memory by at least an order of magnitude. Comer and Griffioen suggested the use of dedicated remote memory servers in distributed systems[Comer90]. Each remote memory server provides clients with additional memory to store client data. However. unlike cooperative caching. the data stored in a remote memory server cannot be shared across applications in different clients. In addition. the mechanism does not use any idle memory in the clients themselves. Therefore. in contrast to cooperative caching. active clients cannot take advantage of the memory in idle clients.

Felten and Zahorjan removed the limitation of dedicated remote memory servers and allowed idle clients to make their memory available for use by the remaining clients[Felten91]. Whenever a client becomes idle. the client informs a manager that the client is ready to provide remote memory service to other clients. Clients contact the manager to locate a remote memory server and proceed to use the server as a backing store for virtual memory. While the system allows effective use of idle client memory. this system does not allow data stored in the memory of an idle client to be shared across applications in different clients. Cooperative caching improves on this by allowing applications in different clients to benefit from sharing of data.

Schilit and Duchamp introduced remote memory paging to mobile computing [Schilit91]. Mobile computers have limited storage space. making remote memory paging an attractive alternative. As mobile clients operate in a weakly-connected environment. the authors designed a remote paging system that adapts to changes in connectivity and allows clients to locate efficiently remote memory servers after a disconnection.

7.1.2 Databases

Franklin et al. examined traditional client-server database architectures and found that their performance was limited by the inability to use effectively the idle memory in client workstations. Franklin introduced a global approach to memory management by extending the architecture of the Exodus system using three important concepts[Carey86. Franklin92]. First. in a manner similar to block lookup in cooperative caching, a database client obtains pages directly from another database client instead of contacting the server. Second. a database client sends a locally replaced page to the server if the database client is informed by the server that the page is a singlet. This mechanism uses the server memory to hold pages that are replaced from the caches of database clients. Finally. to prevent duplicate caching of a page by both database clients and the server. a page in the server cache is marked for deletion if the page is also cached by database clients.

Many of the principles of this extended client-server database architecture influenced the designers of N-chance. Franklin's architecture emphasizes the utility of duplicate avoidance in a replacement algorithm and focuses on the effective use of the server memory. However. the replacement algorithm in this system is not global as the algorithm does not attempt to estimate the value of a page with respect to the pages in the caches of all database clients. In addition. the algorithm does not attempt to reduce the number of messages to the database server, a primary aim in cooperative caching.

7.1.3 Multiprocessors

Cooperative caching is also related to caching strategies found in multiprocessors. The caching strategy in a multiprocessor depends on its type, and cooperative caching relates to each caching strategy in a different way.

Multiprocessors are principally of two types: *shared-memory* or *distributed memory*. A shared memory multiprocessor provides hardware support to address the entirety of system memory from a single processor. This simplifies parallel programming as every processor accesses memory using the same load/store mechanism seen in uniprocessors. Shared memory architectures are of three types: uniform memory access(UMA), non-uniform memory access(NUMA) and cache-only memory access(COMA).

UMA machines consist of a collection of processors, each with a local cache, and a main memory connected by a shared memory bus[SGI96]. The memory hierarchy in a UMA machine is very similar to the storage hierarchy in a distributed file system. However, as the shared memory bus is a bottleneck for a UMA machine, it is important to reduce the amount of message traffic in such a machine. Consequently, cooperative caching strategies are not suitable for UMA machines as managing the cooperative cache adds message overhead. While there is hardware support for locating a block in the caches of other processors, cooperative caching replacement policies are usually not implemented in hardware because the benefit obtained from global replacement does not warrant the cost of a hardware implementation of the replacement policy.

NUMA machines avoid the bottleneck of UMA machines by removing the restriction of a single memory bus. NUMA machines consist of multiple memories and processors connected by multiple busses and an interconnection network[Lenoski90, Agarwal91, Cray93]. The chief difference between a distributed file system and a NUMA machine is that one processor is more likely to be separated from another by multiple network hops in a NUMA machine than in a distributed file system. As a result, there is a much larger variation in access times to the memories of different processors in a NUMA machine than there is in a distributed file system. Non-uniform access times complicate cooperative caching replacement policies because of the need to factor in the access time in determining the value of a block. As a result, cooperative caching replacement is not suitable for NUMA machines because of the cost of implementing a complicated replacement policy in hardware.

COMA machines are similar to NUMA machines but operate on cache lines rather than memory pages or file blocks[KSR92]. This fine-grained memory access requires the memory management policy to be implemented in hardware, making a global replacement policy unsuitable.

In contrast to shared memory architectures, distributed memory architectures do not provide support for direct access to the memory in remote processors. Distributed memory architectures consists of several nodes connected by an interconnect, where each node has its own processor, cache and local memory[Intel91, Felten96]. A processor accesses the memory in another processor using an explicit send/receive message, as in a cooperative-caching distributed file system. Caching strategies for distributed memory systems usually provide support for looking up pages in the memory of remote processors. However, there has been no focus on global replacement policies for distributed memory architectures because typical applications are more compute-intensive than memory-intensive.

One important difference between multiprocessors and distributed file systems is that message costs are greater in distributed file systems than in multiprocessors and have a greater impact on performance. Thus there must be a concerted effort to reduce the number and size of messages required. This is a focus of distributed shared memory research where researchers try to emulate the behavior of a distributed memory multiprocessor on a cluster of workstations[Carter91]. However, distributed shared memory researchers have not focused on a global replacement primarily because typical applications focus more on parallelism than memory use.

7.2 Hints in Computer System Design

Lampson noted that hints are an effective way to speed up the performance of computer systems[Lampson83]. Lampson describes a hint to be the saved result of some computation whose purpose is to make a system run faster. Echoing the principles stated in Chapter 4, Lampson states that hints need to be correct almost all the time for an improvement in performance and points out that checking hints against facts must be adequate and optimizable.

7.2.1 Operating Systems

The use of hints in operating system design is mostly found in the area of file organization, caching and prefetching. The Alto and Pilot operating systems use hints to lookup file blocks on a disk[Lampson79, Redell80]. In the Alto operating system, each file block on a disk contains a hint as to the location of the next file block on the disk. In the Pilot operating system, a special data structure implements a direct map between a file and the address of its first block on the disk, assuming that consecutive file blocks occupy contiguous disk space. These hints are used to speed up file accesses to the disk. However in this particular case, the cost of an incorrect hint is enormous because both operating systems reconstruct their hints by scanning the entire disk.

Caching and prefetching techniques use hints to improve file system and database performance. For example, LRU treats the time of last use of a cached object as a hint to determine the value of the object. If the object has not been used a long time, LRU diminishes the value of a block.

Hints are aggressively used in prefetching blocks in a file system or a database. As the future accesses to a file system or a database can never be known with absolute certainty, system designers use historical information about previous accesses as hints to predict future accesses. For example, in the sequential read-ahead technique, an operating system aggressively prefetches blocks whenever it detects a pattern of sequential access[McKusick84]. A past pattern of sequential access is taken as a hint for future sequential accesses and the hint is discarded whenever accesses becomes non-sequential. In this case, a correct hint hides the latency of a file system access. However, an incorrect hint hurts the cache hit ratio and incurs the full latency to access a block. As a result, read-ahead techniques tend to be conservative in determining a sequential access pattern.

7.2.2 Distributed Shared Memory

The use of hints to perform block lookup is similar to the techniques used to perform page lookup in distributed shared memory systems that support parallel computation. Li and Hudak describe several strategies for managing distributed shared pages in a cluster of workstations[Li89]. The location of a page changes due to program execution as reading and writing processes on different workstations try to access the page. The authors try out several strategies for managing page location information which include centralized management, fixed distributed management, and dynamic distributed management. In the centralized management scheme, one workstation keeps track of the location of all the shared memory pages. In the fixed distributed management scheme, page management is distributed over multiple workstations but the assignment of pages to a workstation manager is fixed. The dynamic distributed management removes the restriction of fixed assignment and allows any workstation to manage the location of a page. Workstations use hints to keep track of the probable location of a page and use these hints to access shared pages. When a workstation needs a page, the workstation sends a request to the probable location of the page. If the workstation receiving the request does not have the page, the request is forwarded to the workstation which is believed to be the probable location of the page. If the page location hints are accurate, the message overhead of a dynamic distributed management scheme will be less than the other schemes as they require an extra message to a manager to locate a page. Indeed, experimental results show that the location hints are quite accurate and the actual number of forwarded requests is very small.

However, there are important differences between this work and the hint-based cooperative caching algorithm presented in the dissertation. Unlike the hint-based algorithm. all workstations keep track of probable location information for all pages. so that a request eventually reaches the correct location of a page. In the hint-based algorithm. the number of file blocks in the server is too large for a client to keep track of. Consequently. a client keeps track of only the location of those blocks which are being accessed by the client. The hint-based algorithm also differs in that blocks can be forwarded to the cooperative cache. necessitating a global replacement policy. In contrast. the distributed shared memory work relies solely on the local replacement policy in each individual workstation.

7.2.3 Victim Cache

There are some similarities between the discard cache and the victim cache proposed by Jouppi[Jouppi90]. A victim cache is a small (2-4 lines) fully-associative miss cache which is used to store the *victims* from a larger direct-mapped level-one processor cache. where a victim is the entry that is removed from the level-one cache to make room for other entries. As a result. two cache lines that conflict in the processor cache can both be cached in the victim cache. While fully-associative caches are expensive in terms of logic to build. the size of this very small supplemental cache makes it feasible to implement on-chip between the direct-mapped level-one and level-two caches. The benefit of a victim cache therefore depends on the relative cost of fetching a cache line from the victim cache as opposed to a level-two cache. Consequently. the performance of a victim cache is more beneficial if the level-two cache is off-chip. However. with technology trends pointing to more on-chip level-two caches. the utility of a victim cache may decline in the future.

Both the victim cache and the discard cache augment the use of hints in computer systems. In essence. the victim cache catches replacement mistakes made by the direct-map cache allocation in the level-one processor cache. Similarly. the discard cache catches replacement mistakes due to incorrect age hints about the blocks in the client caches.

7.2.4 Networking

Hints are also prevalent in wide-area computer networks because of the difficulty in capturing the entire state of a system. Network hints are also unique in that a correctness check for a hint is practically impossible. In Internet routing, each node broadcasts its local routing table and the quality of its links to its neighbors [Schwartz80, McQuillan80]. These broadcast messages can be viewed as hints as the messages are best-effort and do not guarantee a consistent view of local routing traffic. Hints are also found in Internet protocols like TCP which use transmission delays and sequence numbers to infer flow-control and congestion properties about a TCP connection[Postel81].

7.2.5 Programming Languages

Dynamically binding programming languages sometimes use hints to infer the specific method to be invoked based on the type of the arguments passed to a polymorphic method. For example, Smalltalk uses a cache of methods indexed by the type of the first argument to a method[Deutsch82]. This cache is treated as a hint and is used by Smalltalk to directly invoke a method based on the type of its first argument. Experiments have shown that this type of hint is highly accurate.

7.3 Summary

Cooperative caching for file systems developed from research involving remote memory usage. The idea of remote memory servers in distributed systems was first introduced by Comer and Griffioen. Felten and Zahorjan proposed the use of idle machines as remote memory servers. Franklin et al. introduced the concept of remote client servers to extend the traditional client-server database architecture. Cooperative caching is also related to multiprocessor caching in that processor nodes can access blocks in the local memory or cache in another processor node.

The use of hints is system design is widely prevalent. Operating system designers use hints to speed up performance in critical areas such as file systems, caching and prefetching. Hints can be found even in computer architecture: a small associative cache known as the victim cache catches replacement mistakes from the level-one processor cache. Hints can also be seen in computer networks and dynamic-binding programming languages.

CHAPTER 8
CONCLUSIONS

Researchers have devised a new technique to improve the performance of distributed file systems by introducing the *cooperative cache*, a new layer in the storage hierarchy between the client caches and the server. The technique of *cooperative caching* not only allows a client to access the caches of other clients on a local cache miss, but also enables clients to store valuable blocks in the caches of other clients for later reference.

The goal of cooperative caching is to ensure both high performance and low overhead. Existing fact-based algorithms achieve the first goal of high performance by using managers but incur high overhead in the process. To achieve both the goals of high performance and low overhead, we introduce the use of *hints*. Hints are attractive because they are less expensive to maintain than the exact state of the system. However, inaccurate hints increase overhead and degrade performance, implying the need to make hints as accurate as possible.

This dissertation describes a cooperative caching system that uses hints to locate and replace blocks from the client caches:

- Block location hints allow a client to access blocks in the caches of other clients without involving a manager. To make hints accurate, the cooperative caching system relies on a block's *master* copy, one which is obtained from the server. Every client caches block location hints which refer to the probable location of the master copy of a block in the client caches.

- For replacement, clients maintain an *oldest block list* containing the probable ages of the oldest block on every client. A client uses its oldest block list to replace the oldest block in the client caches and exchanges age information on every replacement to keep the oldest block list accurate. The replacement algorithm does not involve any interaction with a manager.

- To use better the server memory in cooperative caching, replacement mistakes are sent to a *discard cache* in the server memory.

Two experimental methodologies are used to evaluate the use of hints in managing the cooperative cache. First, trace-driven simulations compare the hint-based algorithm with existing and ideal algorithms. The simulations revealed that hints are highly successful in locating and replacing blocks from the client caches:

- The block access time of the hint-based algorithm matches those of the existing algorithm over all four periods of the Sprite traces.
- Block location hints are highly accurate: when a location hint indicates that a block is present in the cooperative cache, the hint is correct 99.94% of the time.
- The manager load in the hint-based algorithm is lower by as much as 30 times when compared to that in the existing algorithms.
- The evaluation of the discard cache as a use for the server memory reveals that the hit ratio to the server memory as a discard cache is the highest at 2.46% compared to that of 1.84% and 0.46% when the server memory is used as part of the cooperative cache and as a traditional server cache respectively.
- The block access time of the hint-based algorithm diverges by about 5% from that of the ideal algorithms when the client cache sizes are reduced by a factor of four, but the hint-based algorithm's manager load is 30-50 times lower than that of the existing algorithms.

A subsequent prototype implementation over a cluster of Linux workstations not only showed that hint-based cooperative caching was able to provide high performance and low overhead over a real user workload, but also validated the results from the simulations:

- The average block access time with cooperative caching was almost half that of NFS, providing considerable performance gains to applications.
- The average and maximum overhead rate in the hint-based cooperative caching was found negligible ($< 7.5\%$) compared to the available network bandwidth of a 10 Mbps Ethernet network.
- The accuracy of block location hints in the prototype mirrored those from the simulation at 98%.
- The accuracy of best-guess replacement was also found to be very high, as only 0.2-1.1% of the forwards were found to be directed towards active clients.

8.1 Future Research

While the hint-based algorithm was effective with the workloads used in this dissertation, the cooperative caching system must be tested further in different environments. Of particular interest are memory-intensive benchmarks such as large databases which test how efficiently a cooperative caching system uses the idle memory in the distributed file system. While the hint-based algorithm was evaluated in this mode by varying simulation parameters, the implementation must also be evaluated using real workloads whose working sets are comparable to the aggregate size of the client memories in the distributed file system.

There are also other directions for improving the hint-based algorithm. The hint-based algorithm assumes LRU for replacing blocks from the client caches. However, LRU is less than ideal as a replacement policy for a significant number of benchmarks. As a result, future research should concentrate on how best-guess replacement could be configured to take into account other parameters in replacing a block from the client caches.

Another important research area is to construct a cost-benefit model for cooperative caching. A distributed file system might reach a state where the benefit of cooperative caching is outweighed by the overhead in managing the cooperative cache. One example of such a state is when the working set of applications greatly exceeds the aggregate size of the client memories, as seen in Chapter 5.2.7. While a distributed cost-benefit model is hard to formulate or implement, the active clients accessing idle memory could each use a local model to independently decide when to stop using the cooperative cache.

Even though there is scope for further research on cooperative caching, the dissertation effectively shows how hints are used to achieve high performance and low overhead in cooperative caching.

REFERENCES

[Acharya98] Anurag Acharya and Sanjeev Setia. The utility of exploiting idle memory for data intensive computations. Technical Report TRCS98-02. University of California at Santa Barbara. February 1998.

[Agarwal91] Anant Agarwal. David Chaiken. Godfrey D'Souza. Kirk Johnson. David Kranz. John Kubiatowicz. Kiyoshi Kurihara. Beng-Hong Lim. Gino Maa. Dan Nussbaum. Mike Parkin. and Donald Yeung. The MIT Alewife machine: A large-scale distributed-memory processor. Technical Report MIT/LCS Memo TM-454. MIT. 1991.

[Anderson95] Thomas Anderson. Michael Dahlin. Jeanna Neefe. David Patterson. Drew Roselli. and Randolph Wang. Serverless Network File Systems. In *Proceedings of the 15th Symposium on Operating System Principles*. pages 109–126. December 1995.

[ANSI87] ANSI. *Fiber-Distributed Data Interface (FDDI) Token Ring Media Access Control.* ANSI X3.139-1987. November 1987.

[Bach87] Maurice Bach. Mark Luppi. Anna Melamed. and Kang Yueh. A Remote File-Cache for RFS. In *Proceedings of the USENIX Summer 1987 Conference*. pages 275–280. June 1987.

[Baker91] Mary Baker. John Hartman. Michael Kupfer. Ken Shirriff. and John Ousterhout. Measurements of a Distributed File System. In *Proceedings of the 13th Symposium on Operating System Principles*. pages 198–212. October 1991.

[Beck96] Michael Beck. Harold Bohme. Mizko Dziadzka. Ulrich Kunitz. and Robert Magnus. *Linux Kernel Internals*. Addison-Wesley Publishing Company. 1996.

[Belady66] Les Belady. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Systems Journal*. 5(2):78–101. 1966.

[Cao95] Pei Cao. Edward Felten. Anna Karlin. and Kai Li. A Study of Integrated Prefetching and Caching Strategies. In *Proceedings* *of the 1995 SIGMETRICS Conference*. pages 188–197. May 1995.

[Carey86] Michael Carey. David Dewitt. Daniel Frank. Goetz Graefe. Madhu Muralikrishna. Joel Richardson. and Eugene Shekita. The architecture of the EXODUS extensible DBMS. In *Proceedings of the 12th VLDB Conference*. pages 52–65. 1986.

[Carter91] John Carter. John Bennett. and Willy Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th Symposium on Operating System Principles*. pages 152–164. October 1991.

[Comer90] Douglas Comer and James Griffioen. A New Design for Distributed Systems: The Remote Memory Model. In *Proceedings of the Summer 1990 Usenix Conference*. pages 127–135. June 1990.

[Cray93] Cray. The Cray T3D Hardware Reference Manual. Technical report. Cray Research Inc.. 1993.

[Dahlin94] Michael Dahlin. Randolph Wang. Thomas Anderson. and David Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proceedings of the 1st Symposium on Operating System Design and Implementation*. pages 267–280. November 1994.

[Dahlin95] Michael Dahlin. *Serverless Network File Systems*. Ph.D. Thesis - University of California at Berkeley. 1995.

[Deutsch82] Peter Deutsch. Personal Communication to Butler Lampson. In *Xerox PARC*. February 1982.

[Easton79] Malcom Easton and Peter Franaszek. Using Bit Scanning in Replacement Decisions. *IEEE Transactions on Computing*. 28(2):133–141. February 1979.

[Feeley95] Michael Feeley. William Morgan. Frederic Pighin. Anna Karlin. and Henry Levy. Implementing Global Memory Management in a Workstation Cluster. In *Proceedings of the 15th Symposium on Operating System Principles*. pages 201–212. December 1995.

[Feeley96] Michael Feeley. *Global Memory Management for Workstation Networks*. Ph.D. Thesis - University of Washington. 1996.

[Felten91] Edward Felten and John Zahorjan. Issues in the Implementation of a Remote Memory Paging System. Technical Report 91-03-09. University of Washington. March 1991.

[Felten96] Edward Felten. Richard Alpert. Angelos Bilas. Matthias Blumrich. Douglas Clark. Stefanos Damianakis. Cezary Dubnick. Liviu Iftode. and Kai Li. Early experience with message passing in the SHRIMP multicomputer. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*. May 1996.

[Forum93] The ATM Forum. *The ATM Forum User-Network Interface Specification*. Prentice-Hall International. 1993.

[Franklin92] Michael Franklin. Michael Carey. and Miron Livny. Global Memory Management in a Client-Server DBMS Architectures. In *Proceedings of the 18th VLDB Conference*. pages 596-609. August 1992.

[Griffioen94] James Griffioen and Randy Appleton. Reducing File System Latency using a Predictive Approach. In *Proceedings of the 1994 Summer USENIX Conference*. 1994.

[Henessey96] John Henessey and David Patterson. *Computer Architecture: A Quantative Approach*. Morgan Kaufmann Publishers. 1996.

[Hitz94] Dave Hitz. James Lau. and Michael Malcom. File System Design for an NFS File Server Appliance. In *Proceedings of the Winter USENIX Techical Conference*. pages 235–246. December 1994.

[Howard88] John Howard. Michael Kazar. Sherri Menees. David Nichols. Mahadevan Satyanarayanan. Robert Sidebotham. and Michael West. Scale and Performance in a Distributed File System. *ACM Transactions of Computer Systems*. 6(1):51-81. February 1988.

[Intel91] Intel. Paragon XP/S Product Review. Technical report. Intel Supercomputer Systems Division. 1991.

[Jamrozik96] Herve Jamrozik. Michael Feeley. Geoffrey Voelker. James Evans II. Anna Karlin. Henry Levy. and Mary Vernon. Reducing Network Latency Using Subpages in a Global Memory Environment. In *Proceedings of the Seventh ACM Conference on*

Architectural Support for Programming Languages and Operating System. October 1996.

[Jouppi90] Norman Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[Kotz91] David Kotz and Carla Ellis. Practical Prefetching Techniques for Parallel File Systems. In *Proceedings of the 1st International Conference on Parallel and Distributed Information Systems*, pages 182–189, December 1991.

[KSR92] KSR. Kendall Square Research Technical Summary. Technical report, Kendall Square Research Inc., 1992.

[Lampson79] Butler Lampson and Robert Sproull. An Open Operating System for a Single-user Machine. *Operating System Review*, 13(5):98–105, December 1979.

[Lampson83] Butler Lampson. Hints for Computer System Design. *Operating System Review*, 17(5):33–46, October 1983.

[Lazowska86] Edward Lazowska, John Zahorjan, David Cheriton, and Willy Zwaenepoel. File Access Performance of Diskless Workstations. *ACM Transactions on Computing Systems*, 4(3):238–268, August 1986.

[Leach83] Paul Leach, Paul Levine, Bill Douros, James Hamilton, Debra Nelson, and Bernard Stumpf. The Architecture of an Integrated Local Area Network. *IEEE Journal of Selected Areas in Communication*, 1(5):237–252, November 1983.

[Leff91] Avraham Leff, Philip Yu, and Joel Wolf. Policies for Efficient Memory Utilization in a Remote Caching Architecture. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, pages 198–207, December 1991.

[Lenoski90] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The Directory-based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.

[Li89] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions of Computer Systems*. 7(4):321–359. November 1989.

[McKusick84] Marshall McKusick. William Joy. Samuel Leffler. and Robert Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*. 2(3):181–197. August 1984.

[McQuillan80] John McQuillan. Ira Richer. and Eric Rosen. The New Routing Algorithm for the Arpanet. *IEEE Transactions on Communication*. 28(5):711–719. May 1980.

[Metcalfe76] Robert Metcalfe and David Boggs. Ethernet: Distributed Packet Switching for Local Area Networks. *Communications of the ACM*. 19(7). July 1976.

[Mosberger96] David Mosberger and Larry Peterson. Making Paths Explicit in the Scout Operating System. In *Proceedings of the 2nd Symposium on Operating System Design and Implementation*. pages 153–168. November 1996.

[Nelson93] Michael Nelson. Brent Welch. and John Ousterhout. Caching in the Sprite Network File System. *ACM Transactions of Computer Systems*. 11(2):228–239. February 1993.

[Ousterhout85] John Ousterhout. Herve DaCosta. David Harrison. John Kunze. Michael Kupfer. and James Thompson. A Trace-driven Analysis of the 4.2 BSD UNIX File System. In *Proceedings of the 10th Symposium on Operating System Principles*. pages 15–24. December 1985.

[Ousterhout88] John Ousterhout. Andrew Cherenson. Fred Douglis. Michael Nelson. and Brent Welch. The Sprite Network Operating System. *IEEE Computers*. 21(2):23–36. February 1988.

[Patterson95] Hugo Patterson. Garth Gibson. Eka Ginting. Daniel Stodolsky. and Jim Zelenka. A Study of Integrated Prefetching and Caching Strategies. In *Proceedings of the 15th Symposium on Operating System Principles*. pages 79–95. December 1995.

[Peterson96] Larry Peterson and Bruce Davie. *Computer Networks: A Systems Approach*. Morgan Kaufmann Publishers. 1996.

[Popek85] Gerald Popek and Bruce Walker. *The LOCUS Distributed System Architecture*. The MIT Press. 1985.

[Postel81] John Postel. Transmission Control Protocol. Technical Report RFC793. Information Sciences Institute. September 1981.

[Proebsting97] Todd Proebsting. Gregg Townsend. Patrick Bridges. John Hartman. Tim Newsham. and Scott Watterson. Toba: Java for Applications: A Way Ahead of Time (WAT) Compiler. In *Proceedings of the 3rd USENIX Conference on Object-Oriented Technologies and Systems*. June 1997.

[Redell80] David Redell. Pilot: An operating system for a personal computer. *Communications of the ACM*. 23(2):81–91. February 1980.

[Riedel96] Eric Riedel and Garth Gibson. Understanding Customer Dissatisfaction with Underutilized Distributed File Servers. In *Proceedings of the 5th NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies*. September 1996.

[Sandberg85] Russel Sandberg. David Goldberg. Steve Kleiman. Daniel Walsh. and Bob Lyon. Design and Implementation of the Sun Network File System. In *Proceedings of the Summer 1985 Usenix Conference*. pages 119–130. June 1985.

[Sarkar96] Prasenjit Sarkar and John Hartman. Efficient Cooperative Caching using Hints. In *Proceedings of the 2nd Symposium on Operating System Design and Implementation*. pages 35–46. November 1996.

[Satyanarayanan85] Mahadevan Satyanarayanan. James Howard. David Nichols. Robert Sidebotham. Alfred Spector. and Michael West. The ITC Distributed File System: Principles and Design. In *Proceedings of the 10th Symposium on Operating System Principles*. pages 35–50. December 1985.

[Schilit91] Bill Schilit and Daniel Duchamp. Adaptive Remote Paging for Mobile Computers. Technical Report CUCS-004-91. Columbia University. February 1991.

[Schroeder85] Michael Schroeder. David Gifford. and Roger Needham. A Caching File System for a Programmer's Workstation. In *Proceedings of the 10th Symposium on Operating System Principles*. pages 25–34. December 1985.

[Schwartz80] Mischa Schwartz and Thomas Stern. Routing Techniques used in Computer Communication Networks. *IEEE Transactions on Communication*. 28(4):539-552. April 1980.

[SGI96] SGI. Power Challenge: Technical Report. Technical report. Silicon Graphics Inc.. 1996.

[Silberschatz95] Abraham Silberschatz and Peter Galvin. *Operating System Concepts*. Addison Wesley Publishing Company. January 1995.

[Smith82] Alan Smith. Cache Memories. *ACM Computing Surveys*. 14(3):473-530. September 1982.

[Sun88] Sun. RPC: Remote Procedure Call Protocol Specification Version 2. Technical Report RFC1057. Internet Network Working Group. June 1988.

[Tannenbaum96] Andrew Tannenbaum. *Modern Operating Systems*. Prentice-Hall International. 1996.

[Voelker97] Geoffrey Voelker. Herve Jamrozik. Mary Vernon. Henry Levy. and Edward Lazowska. Managing Server Load in Global Memory Systems. In *Proceedings of the 1997 ACM SIGMETRICS conference*. pages 127-138. June 1997.

[Voelker98] Geoffrey Voelker. Eric Anderson. Tracy Kimbrel. Michael Feeley. Jeff Chase. Anna Karlin. and Henry Levy. Implementing Cooperative Prefetching and Caching in a Global Memory System. In *Proceedings of the 1998 ACM SIGMETRICS conference*. June 1998.

IMAGE EVALUATION TEST TARGET (QA-3)