The following paper was originally published in the
Proceedings of the USENIX 2nd Symposium on
Operating Systems Design and Implementation
Seattle, Washington, October 1996

Efficient Cooperative Caching using Hints

Prasenjit Sarkar and John Hartman
University of Arizona

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

Efficient Cooperative Caching using Hints

Prasenjit Sarkar and John Hartman
{psarkar,jhh}@cs.arizona.edu
Department of Computer Science
University of Arizona
Tucson, AZ 85721

Abstract

We present a very low-overhead decentralized algorithm for cooperative caching that provides performance comparable to that of existing centralized algorithms. Unlike existing algorithms that rely on centralized control of cache functions, our algorithm uses *hints* (i.e. inexact information) to allow clients to perform these functions in a decentralized fashion. This paper shows that a hint-based system performs as well as a more tightly coordinated system while requiring less overhead. Simulations show that the block access times of our system are as good as those of the existing tightly-coordinated algorithms, while reducing manager load by more than a factor of 15, block lookup traffic by nearly a factor of two-thirds, and replacement traffic by more than a factor of 5.

1 Introduction

Caching is a common technique for improving the performance of distributed file systems [Howard88, Nelson93, Sandberg85]. Client caches filter application I/O requests to avoid network and server traffic, while server caches filter client cache misses to reduce disk accesses. A drawback of this organization is that the server cache must be large enough to filter most client cache misses, otherwise costly disk accesses will dominate system performance.

A solution is to add another level to the storage hierarchy, one that allows a client to access blocks cached by other clients. This technique is known as *cooperative caching*[Dahlin94] and it reduces the load on the server by allowing some local client cache misses to be handled by other clients.

The *cooperative cache* differs from the other levels of the storage hierarchy in that it is distributed across the clients and it therefore shares the same physical memory as the local caches of the clients. A local client cache is controlled by the client, and a server cache is controlled by the server, but it is not clear who should control the cooperative cache. For the cooperative cache to be effective, the clients must somehow coordinate their actions.

The major contribution of this paper is to show that a cooperative caching system that relies on local hints to manage the cooperative cache performs as well as a more tightly-coordinated fact-based system. Our motivation is simple: hints are less expensive to maintain than facts and as long as hints are highly accurate, they will improve system performance. Hence, instead of maintaining and accessing global state, each client in a hint-based system gathers its own information about the cooperative cache. Thus the key challenge in designing a hint-based system is to ensure that the hints are highly accurate with minimal overhead.

In this paper, we describe a cooperative caching system that uses hints instead of facts whenever possible. We then use trace-driven simulation to show that the block access times of the proposed hint-based algorithm are as good as those of the existing tightly-coordinated algorithms, while reducing manager load by more than a factor of 15, lookup traffic by nearly a factor of two-thirds, and replacement traffic by more than a factor of 5.

The remainder of the paper is organized as follows: Section 2 introduces the issues in cooperative caching. Section 3 briefly discusses previous cooperative caching algorithms. Section 4 describes the proposed hint-based algorithm. Section 5 describes our methodology for evaluating the algorithms, and Section 6 evaluates the performance of all the algorithms and analyzes the tradeoffs in design. Related work is discussed in Section 7 and we conclude in Section 8.

2 Cooperative Caching

A cooperative caching algorithm involves three logical entities: *clients*, *servers*, and *managers*. Clients access the blocks stored on the servers, and the managers control the cooperative cache. The control provided by the managers may include locating blocks in the cache and deciding which blocks to cache, but it varies among cooperative caching algorithms and serves as the major distinction

between them.

Although it is enticing to think of cooperative caching as simply another layer in the storage hierarchy, management of the cooperative cache can potentially involve every machine in the system since the cache is distributed across all the clients. The following sections look at the coordination aspects of cooperative caching and discuss design approaches.

2.1 Block Lookup

Handling a local cache miss by a client requires retrieving the missing block from the lower levels of the storage hierarchy. This procedure is called *block lookup* and in cooperative caching, it involves retrieving blocks from the cooperative cache. However, the cooperative cache is distributed over the caches of all the clients. Thus, retrieving a block from the cooperative cache would require information about the location of blocks in the client caches.

There are two possible strategies of maintaining block location information. One strategy is based on managers which keep track of which blocks are present in the client caches. A client does a block lookup by sending a request to the manager responsible for the block. 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.

An alternative strategy is to let the clients maintain block location hints and do the block lookup themselves, avoiding the cost of contacting managers.

2.2 Replacement Policy

The role of the cooperative cache replacement policy is to determine the order in which blocks are replaced. The cooperative cache replacement policy is activated when a client decides to replace a block from its local cache. This block can be either *discarded*, or *forwarded* to the cooperative cache on a target client.

A replacement policy can use two factors in deciding whether or not to forward a block. First, a block is discarded if the replacement algorithm decides that the block is less valuable than any block in the cooperative cache. Otherwise, the block is forwarded to the target client which then replaces a block in its cache.

The second factor in deciding whether or not a block should be forwarded to the cooperative cache is *duplicate avoidance*. Since the cooperative cache is a resource used by all of the clients, the potential exists for uncoordinated client actions to result in several copies of the same block in the cooperative cache. These duplicate copies pollute the cooperative cache and reduce its hit rate. Thus a block should not be forwarded to the cooperative cache if it is going to become a duplicate copy. In particular, if several clients have a copy of a block in their local caches, only one of the copies should be forwarded to the cooperative cache, and only if the cooperative cache does not already contain a copy.

If the client decides to forward the block, the choice of the target client becomes important in determining the effectiveness of the replacement policy. Thus, the target client should be chosen such that the forwarded block replaces a block less valuable than itself. The replaced block in the target client may be in the client's local cache or in the cooperative cache.

2.3 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 server disk traffic and improving the performance of the file system[Baker91].

The benefits of a server cache are less apparent with cooperative caching because the vast majority of local cache misses are serviced by the cooperative cache. This raises the issue of what to do with the server memory. One possibility is to use it as a part of the cooperative cache. Alternatively, the hint-based algorithm described in this paper instead uses the server memory as a *discard cache* that offsets the impact of incorrect hints.

2.4 Cache Consistency

Cache consistency is important to cooperative caching because the choice of consistency protocol affects the feasibility of using hints to perform block lookup and replacement. Managers typically enforce consistency using *tokens* which control access to a file or block. Managers use tokens in two ways: a token *grant* allows a client to access a file or block, while a token *revocation* causes a client to invalidates its copy of the file or block. Token revocation removes stale copies of a file or block from the client caches, leaving only the up-to-date version.

A *block-based* protocol maintains consistency on each individual block. The problem with a block-based protocol is that the manager must be contacted on every local cache miss to ensure consistency. As a result, there is little point in using hints to avoid contacting the manager for block lookup as described in Section 2.1.

A *file-based* protocol is similar to the block-based except that it maintains consistency on entire files rather than blocks, allowing clients to potentially 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].

3 Previous Algorithms

The original paper on cooperative caching by Dahlin et al.[Dahlin94] described a variety of different schemes for implementing cooperative caching, and settled on one called *N-chance* as providing the best performance with the lowest overhead. The authors provided a partial refinement of the algorithm in the description of the *xfs* file system[Anderson95]. 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. This section discusses these two algorithms. A summary of the previous algorithms, as well as the hint-based algorithm we propose in the next section, is shown in Table 1.

3.1 N-chance

The N-chance algorithm dynamically partitions the cache of each client between blocks needed by the local client and the cooperative cache. Managers are responsible for maintaining consistency and block location information.

The replacement policy in N-chance uses a combination of local LRU information and duplicate avoidance to decide the best block to replace. Clients always replace the oldest block on their LRU lists. Whether the client forwards or discards the oldest block depends on the number of copies of the block. Blocks with more than one copy are discarded, while blocks with only one copy, or *singlets*, are forwarded to another client at random. If a client does not know whether or not one of its blocks is a singlet, it simply asks the manager.

3.2 GMS

GMS is more general than N-chance in that it is a distributed shared-memory system, for which cooperative caching is only one possible use. 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 provide a consistency mechanism.

The replacement policy used by GMS uses duplicate avoidance to determine whether a block should be discarded or forwarded to the cooperative cache. A local block is forwarded to the cooperative cache if no other copies of the block exist (or in N-chance terminology, the block is a singlet). GMS differs from N-chance in that each copy of a block is always tagged as to whether or not it is a singlet. The manager keeps track of the number of copies of each block and notifies the appropriate client when a block becomes a singlet.

GMS also differs from N-chance in that a centralized algorithm implements the choice of the target client in the replacement policy. The algorithm collects age information of blocks in the client caches from all clients at the end of dynamically controlled time intervals called *epochs*. This manager then distributes the locations of the oldest blocks in the system back to all the clients, which is then used to determine the target client in the replacement policy. The algorithm is so designed that it reasonably approximates the global LRU policy over all the client caches.

4 A Hint-based Algorithm

The previous cooperative caching algorithms rely in part on exact information, or *facts*, to manage the cache. Although facts allow these algorithms to make optimal decisions, they increase the latency of block accesses and the load on the managers. Our goal in designing a cooperative caching algorithm is to remove the reliance on centralized control of the cooperative cache. Clients should be able to access and replace blocks in the cooperative cache without involving a manager.

Reducing the dependence of clients on managers is achieved through the use of *hints*, information that only approximates the global state of the system. The decisions made by a hint-based system may not be optimal, but managing hints is less expensive than managing facts. Hints do not need to be consistent throughout the system, eliminating the need for centralized coordination of the information flow. As long as the overhead eliminated by not using facts more than offsets the effect of making mistakes, the gamble of using hints will pay off.

The remainder of this section describes the components of a hint-based algorithm. Section 4.1 describes the hint-based block lookup algorithm. Section 4.2 describes how the replacement policy decides whether or not to forward a block to the cooperative cache. Section 4.3 discusses how the replacement policy chooses the target client for forwarding a block. Section 4.4 discusses the use of the server cache to mask replacement mistakes. Finally, Section 4.5 describes the effect of the cache consistency protocol on the use of hints.

4.1 Block Lookup

When a client suffers a local cache miss a lookup must be performed on the cooperative cache to determine if and where the block is cached. The manager performs this lookup in the previous algorithms, both increasing the

Algorithm	N-chance	GMS	Hint-based
Cache Consistency	Block-based	None	File-based
Block Location	Manager-based	Manager-based	Hint-based
Replacement Policy	Random Client	Manager-based LRU	Best-guess LRU
Duplicate Avoidance	Non-singlets deleted	Non-singlets deleted	Master Copy
Server Caching	Traditional	Traditional	Discard

Table 1: **N-chance and GMS.** This table lists the key features of the N-chance, GMS, and hint-based algorithms.

block access time and incurring load on the manager and network.

An alternative approach is to let the client itself perform the lookup, using its own hints about the locations of blocks within the cooperative cache. These hints allow the client to access the cooperative cache directly, avoiding the need to contact the manager on every local cache miss.

Based on the above, we can identify two principal functions for a hint-based lookup algorithm:

- *Hint Maintenance*: The hints must be maintained so that they are reasonably accurate, otherwise the overhead of looking for blocks using incorrect hints will be prohibitive.
- *Lookup Mechanism*: Hints are used to locate a block in the cooperative cache, but the system must be able to eventually locate a copy of the block should the hints prove wrong.

Each of these functions is discussed in detail in the following sections.

4.1.1 Hint Maintenance

To make sure that hints are reasonably accurate, our strategy is to change hints only when necessary. In other words, correct hints are left untouched and incorrect hints are changed when correct information is made available.

To keep hints as correct as possible, we introduce the concept of a *master copy* of a block. The first copy of a block to be cached by any client is called the master copy. The master copy of a block is distinct from the block's other copies because the master copy is obtained from the server.

Based on this concept of the master copy, we enumerate two simple rules for hint maintenance:

1. When a client obtains a token for a file from a manager, it is also given a set of hints that contain the probable location of the master copy for each block in the file. The manager obtains the set of hints for the file from the last client to acquire a token for the file, because the last client is likely to have the most accurate hints.
2. When a client forwards a master copy of a block to a second client, both the clients now update their hints to show that the probable location of the master copy of the block is the second client.

The hints only contain the probable locations of the master copy and hence, we ignore the changes to the locations of the other copies of the block. This simplifies the task of keeping the hints accurate.

4.1.2 Lookup Mechanism

Given hints about the probable location of the master copy of a block, the lookup mechanism must ensure that a block lookup is successful, regardless of whether the hints are right or wrong. Fortunately, as all block writes go through to the server, it always has a valid copy of a block and can satisfy requests for the block should the hints prove false. This simplifies the lookup mechanism which is outlined below:

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 consults its hint information for the block and proceeds to Step 2.

The general idea is that each client keeps track of the probable location of the master copy of each block, and uses this information to lookup blocks in the cooperative cache.

Simulations show this algorithm works well in a distributed UNIX environment, as described in Section 6. However, the algorithm will not be effective in the following scenario. If several clients share a working set of blocks larger than the cooperative cache, the locations of the master copies will change rapidly as blocks move in and out of the client caches. This will cause the probable master copy locations to be inaccurate, leading to excessive forwarding of requests.

4.2 Forwarding

When a block is ejected from the local cache of a client, the cooperative cache replacement policy decides whether or not the block should be forwarded to the cooperative cache. As discussed earlier, one of the motivations of the replacement policy is to ensure that only one copy of a block is stored in the cooperative cache. If not, the cooperative cache will contain unnecessary duplicate copies of the same block.

The previous algorithms rely on the manager to determine whether or not a block should be forwarded to the cooperative cache. A block is forwarded if it is the only copy of the block stored in either the local caches or the cooperative cache. Maintaining this invariant is expensive, however, requiring an N-chance client to contact the manager whenever it wishes to discard a block that is not known to be a singlet, and the GMS manager to contact a client whenever a block becomes a singlet.

To avoid these overheads, we propose a forwarding mechanism in which 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, while all other copies are discarded. Since only master copies are forwarded, and each block has only one master copy, there can be at most one copy of a block in the cooperative cache.

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 as in GMS or N-chance, the master copy algorithm forwards the first or master copy. In some cases, this may lead to unnecessary forwardings. A block which is deleted before it is down to its last copy should not have been forwarded to the cooperative cache. The existing algorithms avoid this, while the master copy algorithm will potentially forward the block. Fortunately, our measurements show that few of the master copy forwardings are unnecessary, as described in Section 6.3.

4.3 Best-Guess Replacement

Once the replacement policy has decided to forward a block to the cooperative cache, it must decide the target client of this forwarding. Forwarding a block to this target client will replace a block that is either in the client's local cache or in the cooperative cache. Note that this replaced block can be a master copy, providing a means for removing master copies from the cooperative cache.

Previous algorithms either choose the client at random, or rely on information from the manager to select the target. An alternative based on hints, however, can provide highly accurate replacements without requiring a centralized manager. We refer to this 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 current globally LRU block.

In best-guess replacement, each client maintains an *oldest block list* that contains what the client believes to be the oldest block on each client along with its age. This list is sorted by age. A block is 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 status of 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. The exchange of block age information during replacement allows both *active* clients (clients that are accessing the cooperative cache) and *idle* clients (clients that are not) to maintain accurate oldest block lists. Active clients have accurate lists because they frequently forward blocks. Idle clients will be the targets of the forwardings, 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 forwarding requests.

Changes in the behavior of a client may cause the oldest block lists to become temporarily inaccurate. 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 in the oldest block lists on other clients and be 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 accurately forward blocks. 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.

Although trace-driven simulation has shown this simple algorithm to work well, there are several potential problems, including the effect of replacing a block that is not the globally LRU block and also the problem of overloading a client with simultaneous replacements.

First, since global state information is not maintained, it is possible for a client to replace a block that is not the globally LRU block. However, if the replaced block is *close* to the globally LRU block, the performance impact of not choosing the globally LRU block is minimal. In addition, the next section discusses a mechanism for

masking any deviations from the globally LRU block.

Second, if several clients believe that the same client has the oldest block, they will all forward their blocks to that client, potentially overloading it. Fortunately, it can be shown that 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. Over time, the clients' oldest block lists will contain different block age information even if they start out identical, reducing the probability of always choosing the same forwarding targets.

4.4 Discard Cache

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

Table 2: **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, else it is considered *young*. The oldest block list is the per-client list that contains what the client believes to be the oldest block on each client along with its age.

One drawback of best-guess replacement is that erroneous replacements will occur. A block may be forwarded to a client that does not have the oldest block; indeed, a block may be forwarded to a client whose oldest block is actually younger than the forwarded block.

To offset these mistakes we introduce the notion of a *discard cache*, one that is used to hold possible replacement mistakes and thus increase the overall cache hit rate of the system. The simple algorithm used to determine whether a block is mistakenly replaced and should be sent to the discard cache is shown in Table 2. As is evident, non-master copies are always discarded because only master copies are accessed in the cooperative cache.

Replacements are considered to be in error when the target client of a replacement decides that the block is too young to be replaced. A client chooses to replace a block on a particular target client because it believes that client contains the oldest block. The target client considers the replacement to be in error if it does not agree with this assessment. The target determines this by comparing the replaced block's age with the ages of the blocks on its oldest block list. If the block is younger than any of the blocks on the list, then the replacement is deemed an error and the block is forwarded to the discard cache. Otherwise, the block is discarded.

The blocks in the discard cache are replaced in global LRU order. Thus the discard cache serves as a buffer to hold potential replacement mistakes. This extends the lifetimes of the blocks and reduces the number of erroneous replacements that result in an expensive disk access.

4.5 Cache Consistency

The use of hints for block lookup raises the issue of maintaining cache consistency. One solution is to use block-based consistency, but this would require contacting the manager on every local cache miss to locate an up-to-date copy, making it pointless to use hints for block lookup or replacement. For this reason, we propose the use of a file-based consistency mechanism. Clients must acquire a token from the manager prior to accessing a file. The manager controls the file tokens, revoking them as necessary to ensure consistency. The token includes version numbers for all the file's blocks, allowing copies of the blocks to be validated individually. Once a client has the file's token it may access the the file's blocks without involving the manager, enabling the use of hints to locate and replace blocks in the cooperative cache.

5 Methodology

This section describes the methodology used to compare the performance of the N-chance, GMS, hint-based, and ideal algorithms in detail and analyze the design tradeoffs. We describe the simulation environment, the criteria for evaluating the algorithms, and the ideal algorithms against which the other algorithms are compared.

5.1 Simulation Environment

The algorithms were evaluated using trace-driven simulation. The traces of the Sprite distributed file system [Baker91] were used to drive the simulator. These traces cover four two-day *periods*, and record file system accesses by user programs, 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 record application-level behavior and thus are not Sprite-specific. We restricted our use of the traces to the main file server *allspice*. Table 3 shows statistics for each of the trace periods, while Table 4 shows the simulation parameters.

Most of the simulation parameters are derived from the original study on cooperative caching by Dahlin et al.[6], simplifying performance comparisons. The various access times were obtained from previously published measurements. Although these measurements are now a few years old, and thus likely to be slow when compared to

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 3: **Trace Period Statistics.** This table contains statistics the four trace periods. *Active clients* refers to the number of clients that actually used the cooperative at any point during the period.

Clients	42
Servers	1
Managers	1
Client Cache Size	16 MB
Server Cache Size	128 MB
Block Size	8 KB
Local Memory Access Time	0.25 ms
Remote Memory Access Time	1.25 ms
Disk Access Time	15.85 ms
Write policy	write-through
Warm-up Block Accesses	400,000
Forwarding Cache Entries	100
Network	ATM
Message Latency	0.2 ms

Table 4: **Simulation Parameters.** This table describes the environment used to evaluate the various cooperative caching algorithms.

state-of-the-art equipment, they were obtained from real systems. Furthermore, we did not believe that the benefit of updating the simulation parameters warranted the increased difficulty in comparing results.

We also assume that there is a single manager handling centralized functions such as cache 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.

The N-chance and GMS simulators used in these simulations were derived from the simulators created by the systems' designers. 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 cooperative cache instead of a server, improving the cooperative cache hit rate and reducing the load on the servers.

The GMS simulator was modified to add a file-based consistency mechanism. The original GMS system did not contain a consistency mechanism, making it difficult to use for cooperative caching. The consistency mechanism we implemented is identical to the one we use in our hint-based algorithm, as described in Section 4.5.

The simulators were validated through both extensive debugging and analysis of their behavior. For all simulators, we analyzed in detail their processing of a small sample of the traces. Also, all simulators produce expected results for the traces if cooperative caching was disabled. The GMS simulator was further validated by running the traces while varying the length of the epochs. As the epochs were lengthened, the average age of the replaced blocks decreased as expected. Finally, the hint-based simulator was validated both by inspection of its processing of a sample of the traces, and by comparing its results with those of the N-chance simulator.

5.2 Evaluation Criteria

We use the following two metrics to evaluate the performance of the cooperative caching algorithms:

- **Average Block Access Time**: This metric measures the average time required to access a block. The access time is determined by the hit rates in the different layers of the storage hierarchy. Algorithms that make better use of the local and cooperative caches to avoid disk accesses will have low access times. Access time is only measured for block reads because all algorithms use write-through caches.
- **Overhead**: This metric measures the work required to implement cooperative caching. This overhead is broken down into load on the managers, and network messages required to satisfy a local cache miss and replace a block in the cooperative cache.

5.3 Ideal Algorithms

For comparison purposes, the performance of two ideal cooperative caching algorithms is included. These algorithms provide an upper bound on cooperative cache performance, and thus provide an absolute yardstick against which other algorithms may be measured. The ideal algorithms differ in the replacement policy they use. The Optimal algorithm replaces blocks in a fashion that maximizes the hit rate on the cooperative cache, but cannot be implemented because it requires predicting the future. The Global LRU algorithm approximates the Optimal algorithm by replacing the oldest block in the system, but cannot be efficiently implemented because of the overhead required to find and replace the oldest block.

5.3.1 Optimal

The Optimal replacement always replaces the block 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[Belady66] and therefore has the lowest block access time.

The Optimal replacement algorithm for a cooperative cache differs from that in a virtual memory system in that block location must be considered when determining which block to replace. When a client stores a block to the cooperative cache, there may be several candidate blocks to replace which are never accessed in the future. To reduce the number of block transfers, the Optimal algorithm always replaces blocks stored locally at that client, if possible.

5.3.2 Global LRU

The Global LRU algorithm approximates the Optimal algorithm by collecting information about the LRU block of every client to determine the globally LRU block and then replacing it. In a real implementation, this algorithm would not only be expensive but also inaccurate. The inaccuracy stems from the fact that the LRU block of a client may change while another client is collecting information about the LRU blocks of all clients.

The Global LRU replacement policy is based on the two goals of determining the exact globally LRU block and minimizing the number of copies of a block. When a client needs to forward a block to the cooperative cache, it first checks to see if the block is a duplicate. If so, it is discarded, otherwise it is forwarded to the client storing the globally LRU block and the globally LRU block is discarded.

As a result of its replacement policy, the Global LRU algorithm tries to maximize the local and remote cache hit rates but does not try to minimize the number of block transfers. The algorithm always discards the globally LRU block, usually requiring a block transfer. As a result, an algorithm that approximates Global LRU and attempts to minimize block transfers can have a lower replacement overhead than the Global LRU algorithm.

6 Performance

This section describes the performance of the cooperative caching algorithms when simulated using the Sprite traces. They are compared in terms of average block access time, manager load, lookup messages, and replacement messages. The effectiveness of the discard cache is also measured, as is the sensitivity of the block access time to variations in the simulation parameters.

6.1 Block Access Time

Figure 1 shows the average block read access time for all the algorithms, broken down by the time spent in accessing the local cache, remote client caches, server cache, and server disk. Write accesses are not included in the figure because all algorithms use write-through caches.

The read access times for the GMS and hint-based algorithm are very close to the ideal algorithms, and they spent similar amounts of time handling hits in the different levels of the storage hierarchy. The performance of the N-chance algorithm is similar to that of the optimal algorithms in all but the second and fourth periods. During these periods, the N-chance algorithm has a higher number of disk accesses and remote client hits, leading to a longer access time. This is caused by more block accesses in these periods than in the other periods. This increase affects N-chance more than the other algorithms because of its random replacement policy. The higher number of blocks accessed increases the probability that a block replaced at random was being used by the client that cached it.

The time breakdown shows that hits in the server cache are nearly non-existent for all algorithms except the hint-based. This indicates that the hint-based algorithm occasionally makes mistakes in replacing a block from the cooperative cache, but the discard cache in the server corrects some of them.

The block access times for the GMS and hint-based algorithms are very close to that of the ideal algorithms, leaving little room for improvement. The performance of the hint-based algorithm is particularly encouraging, given that hints can be occasionally incorrect.

The block location hints for the cooperative cache are highly accurate. For only 0.01% of the local cache misses (averaged across all periods) is the desired block in the cooperative cache but the hints say it is not. In these cases, the client will erroneously retrieve the block from the server. Conversely, when a hint says a block is in the cooperative cache, it is correct for 99.95% of all local cache misses. Of these correct hints, 97.56% 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 forwardings translate into an average of only 2.03 messages to perform a block lookup. In comparison, both N-chance and GMS require 3 messages per lookup.

6.2 Manager Load

The load imposed on the manager is one measure of the overhead and scalability of an algorithm. The less work a centralized manager must do on the client's behalf, the more scalable is the system. Figure 2 shows the manager

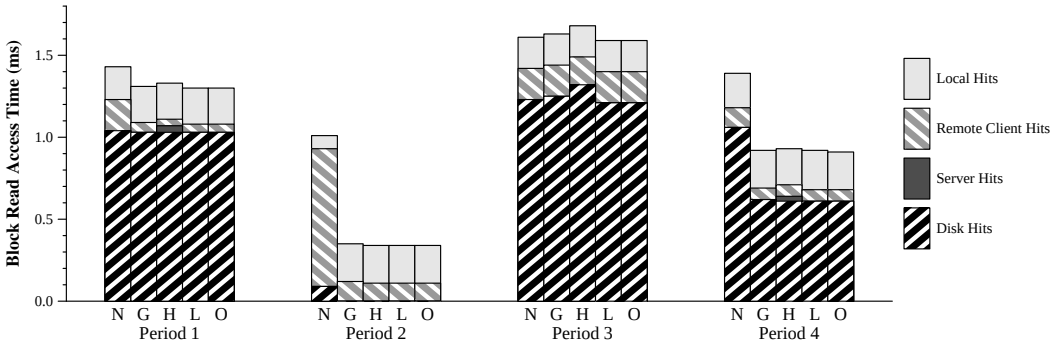

Figure 1: **Block Access Time.** This figure shows the average block read access times for the N-chance(N), GMS(G), hint-based(H), Global LRU(L) and Optimal(O) algorithms for each period of the Sprite traces. The segments of the bars show the fraction of the block read access time contributed by hits to the local cache, cooperative cache, server cache, and server disk.

load categorized by the load's source, and expressed as the number of messages sent and received by the manager. This is a valid measure of the manager load because each message sent and received by the manager represents work performed by the manager.

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

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 and each time a lookup is done, whereas the hint-based algorithm allows the clients to perform these functions themselves. The result is that the manager load is much higher for N-chance and GMS. N-chance has a relatively high manager load during the second and fourth periods because it has a low local cache hit rate in those periods, as described in the previous section. This increases the number of blocks that must be accessed from the cooperative cache or the server, and increases the manager load accordingly.

6.3 Best-Guess Replacement

Figure 2 showed that at about half of the N-chance and GMS manager load is due to managing block replacement. In this section we analyze the overheads associated with block replacement.

Figure 3 depicts the number of replacement messages each algorithm requires to handle a local cache miss. The N-chance and GMS algorithms have three sources of replacement messages: forwarding the block to another client and notifying the manager; notifying the manager

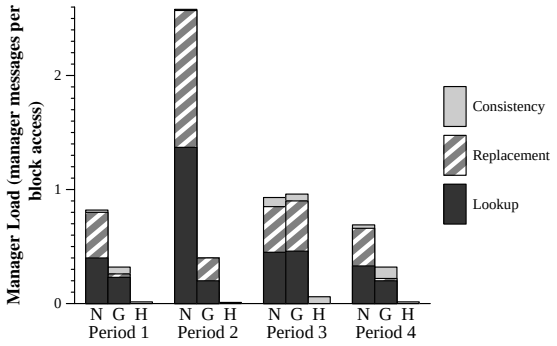

Figure 2: **Manager Load.** This shows the average manager load of the N-chance(N), GMS(G) and hint-based(H) algorithms in the Sprite traces. The manager load is defined as the number of messages received and sent by the manager per block access. The load is categorized by its cause: consistency, replacement and lookup.

when a block is deleted; and exchanging messages between the clients and the manager to determine when a block should be discarded as opposed to 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. This dramatically reduces the total number of replacement messages required per local cache miss.

As mentioned before, one of the potential drawbacks of the master copy algorithm is that it may unnecessarily forward master copies. Although Figure 3 shows that best-guess replacement outperforms the other algorithms despite this drawback, we also measured the fraction of forwardings that were unnecessary. An average of only 2.11% were unnecessary across all periods, indicating that they have a negligible effect on performance.

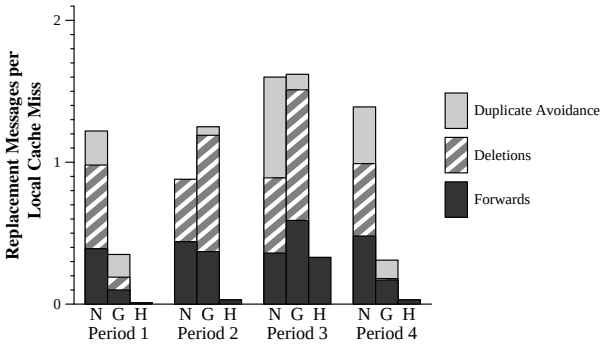

Figure 3: **Replacement Traffic.** This figure shows the number of replacement messages required per local cache miss in the N-chance(N), GMS(G) and hint-based algorithms(H). The messages are categorized by those required to forward a block, delete a block, and keep track of the number of copies of a 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 singlets. For the hint-based algorithm, this includes one message per master copy forwarded.

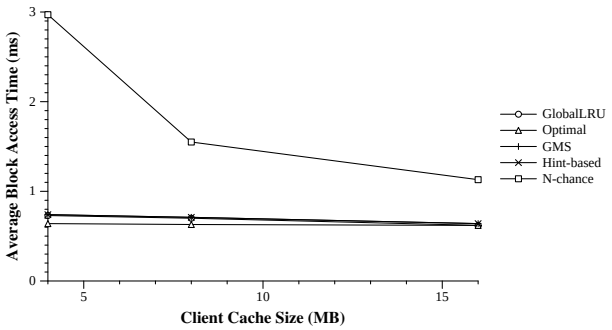

Figure 4: **Access Time vs. Cache Size.** This figure shows the average block read access time for the algorithms as a function of the client cache size.

6.4 Discard Cache

Server Memory	Hit Ratio	Block Access Time(ms)
Disk Cache	0.35%	1.01
Cooperative Cache	1.65%	0.81
Discard Cache	2.16%	0.74

Table 5: **Server Memory Uses.** This table shows how various uses of the server memory affect the 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 averaged across all periods of the trace.

The server memory represents a valuable resource to the system and at 128 MB it constitutes a large fraction of the memory of the system. The hint-based system we propose uses the server memory as a discard cache to mask some of the mistakes made by the best-guess replacement policy. There are other possible uses for the server memory, however, including as a traditional disk cache and as a portion of the cooperative cache. Unfortunately, the default 16 MB client cache size used in the simulations makes it difficult to measure the effectiveness of the discard cache. The cooperative cache is so large and effective that few accesses go to the server. Thus, to measure the effectiveness of the discard cache, we reduced the size of the client caches and server cache to 4 MB and 16 MB, respectively, and ran simulations of the hint-based system with the server memory in the different uses mentioned. Reducing the cache sizes increases the miss rates on the local and cooperative caches, and therefore the load on the server memory.

The results are shown in Table 5 and indicate that when the server memory is used as a traditional disk cache, it has a very low hit rate of 0.35% because most of the blocks it stores are duplicated in the local and cooperative caches. This results in a block access time of 1.01 ms. If the server memory is instead used as part of the cooperative cache, the hit rate increases by nearly a factor of 5, causing the block access time to drop to 0.81 ms. Using the memory as a discard cache, however, further increases the hit rate to 2.16% and drops the block access time to 0.74 ms. By masking some of the replacement mistakes, the discard cache provides a 9% improvement in the block access time over using the memory as part of the cooperative cache.

6.5 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 of concern. This section presents the sensitivity of the block access time to two environmental variables: the client cache size and the fraction of the clients that actively use the cooperative cache.

First, Figure 4 shows the block access time 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 4. A smaller client cache increases the load on the cooperative cache in two ways: first, it increases the local cache miss rates and therefore accesses to the cooperative cache; and second, it reduces the available size of the cooperative cache. As the figure shows, the block access times for most algorithms does not significantly increase with a

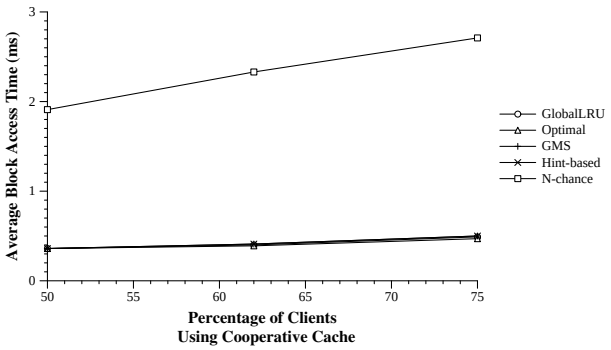

Figure 5: **Access Time vs. Active Clients.** This figure shows the average block read access time for the algorithms as a function of the fraction of clients that are using the cooperative cache during the period. The fraction of clients using the cooperative cache was varied by removing idle client client trace records from the trace. Due to the difficulty in doing this without affecting the workload behavior, only the second period was used.

4 MB client cache, although they do slightly diverge from optimal. Even with caches this small the algorithms do a good job of finding and using the available idle memory, producing access times that are close to optimal. The exception is the N-chance algorithm, whose policy of randomly forwarding blocks hurts it when cooperative cache is scarce.

The sensitivity of the block access time to the fraction of clients that are using the cooperative cache is also of interest. Increasing the fraction of clients that use the cooperative cache increases the demand on the cache, and also decreases the cooperative cache size. This combined effect increases the importance of managing the cooperative cache efficiently.

As Figure 5 shows, only the performance of the N-chance algorithm declines as the fraction 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 perform close to the optimal.

7 Related Work

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 in [Comer90]. Felten and Zahorjan proposed the use of idle machines as remote memory servers in [Felten91]. Franklin et al. in [Franklin92] introduced the concept of remote client servers to extend the traditional client-server database architecture. Leff et al. in [Leff91] showed that memory must be dynamically partitioned between local and remote client needs to maximize hit rates.

Our 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[Li89], including a dynamic distributed manager algorithm in which nodes send page requests to the probable owner of the page. If the target node does not have the page, it forwards the request to the node it believes to be the probable owner. Unlike the hint-based algorithm we propose, all nodes keep track of probable owner information for all pages, so that the request eventually reaches the correct owner. Their results show that the probable owner information is quite accurate and the actual number of forwardings is very small. Our hint-based algorithm also differs in that blocks can be forwarded to the cooperative cache, necessitating a distributed replacement policy. The work of Li and Hudak relies on the virtual memory systems of the individual machines to swap pages to disk, rather than forwarding pages to other nodes.

Cooperative caching is also related to multiprocessor caching in shared memory machines[Lenoski90]. However, 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 one focus of research on distributed shared memory[Carter91].

The discard cache is similar in purpose to the victim cache proposed by Jouppi[Jouppi90]. A victim cache is a small fully-associative miss cache that is placed between a direct-mapped processor cache and the main memory system. The victim cache is loaded with the victim of a cache miss rather than the missed cache line itself. As a result, cache lines that conflict in the processor cache can both be cached in the victim cache, increasing performance. In essence, the victim cache catches replacement mistakes made by the processor cache because it is direct-mapped. The discard cache, in contrast, catches mistakes made because of incomplete information about block ages.

8 Conclusions

Cooperative caching is a technique that allows clients to access blocks stored in the memory of other clients. This enables some of the local cache misses to be handled by other clients, offloading the server and improving the performance of the system. However, cooperative caching requires some level of coordination between the clients to maximize the overall system performance. Previous cooperative caching algorithms achieved this coordination by maintaining global information about the system state. This paper shows that allowing clients to make local de-

cisions based on hints performs as well as the previous algorithms, while requiring less overhead. The hint-based algorithm's block access times are as good as those of the previous and ideal algorithms, while reducing manager load by more than a factor of 15, block lookup traffic by nearly a factor of two-thirds, and replacement traffic by more than a factor of 5.

9 Acknowledgments

We 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. Wanda Chiu, Matti Hiltunen, Mudita Jain, Dave Lowenthal, Anup Kuzhiyil and Todd Proebsting all provided much-needed comments on early drafts of this paper, as did the anonymous reviewers. Our paper shepherd, Peter Chen, also deserves thanks for his efforts to improve the paper. This work has been funded in part by the Advanced Research Projects Agency of the Department of Defense under contracts DABT63-94-C-0049 and DABT63-95-C-0075, and a grant from Intel Corporation.

References

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

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

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

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

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

[Dahlin94] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. 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.

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

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

[Franklin92] Michael J. Franklin, Michael J. 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.

[Howard88] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and Performance in a Distributed File System. *ACM Transactions of Computer Systems*, 6(1):51–81, February 1988.

[Jouppi90] Norman P. 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.

[Leff91] Avraham Leff, Philip S. Yu, and Joel L. 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] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. 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.

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

[Sandberg85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network File System. In *Proceedings of the Summer 1985 Usenix Conference*, pages 119–130, June 1985.