Cooperative Caching: Using Remote Client Memory to Improve File System Performance

Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, David A. Patterson

University of California at Berkeley
{dahlin, rywang, tea, patterson}@cs.berkeley.edu

Abstract

Emerging high-speed networks will allow machines to access remote data nearly as quickly as they can access local data. This trend motivates the use of *cooperative caching*: coordinating the file caches of many machines distributed on a LAN to form a more effective overall file cache. In this paper we examine four cooperative caching algorithms using a trace-driven simulation study. These simulations indicate that for the systems studied cooperative caching can halve the number of disk accesses, improving file system read response time by as much as 73%. Based on these simulations we conclude that cooperative caching can significantly improve file system read response time and that relatively simple cooperative caching algorithms are sufficient to realize most of the potential performance gain.

1. Introduction

Cooperative caching seeks to improve network file system performance by coordinating the contents of client caches and allowing requests not satisfied by a client's local in-memory file cache to be satisfied by the cache of another client.

Two technology trends push us to consider cooperative caching. First, processor performance is increasing much more rapidly than disk performance. This divergence makes it increasingly important to reduce the number of disk accesses by the file system. Second, emerging high-speed low-latency switched networks can supply file system blocks across the network much faster than standard Ethernet as indicated in Figure 1. Where fetching data from remote memory over an older network might be only three times faster than getting the data from remote disk, remote memory may now be accessed ten to twenty times as quickly as disk, increasing the payoff for cooperative caching.

This work is supported in part by the Advanced Research Projects Agency (N00600-93-C-2481), the National Science Foundation (CDA 8722788), California MICRO, the AT&T Foundation, Digital Equipment Corporation, Hewlett Packard, Siemens Corporation, Sun Microsystems, and Xerox Corporation. Dahlin was also supported under a National Science Foundation Graduate Research Fellowship. Anderson was also supported by a National Science Foundation Young Investigator Award.

This paper first appeared in the Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI 1994).

Existing file systems use a three-level memory hierarchy, implementing a limited form of "cooperative caching" by locating a shared cache in server memory to supplement the other two memory levels, client memory and server disk. Although we can often reduce disk accesses by increasing the fraction of the system's RAM that resides in the server, four factors make cooperative caching more attractive than physically moving memory from clients to the server. First, cooperative caching can provide better performance: although either approach can improve the global hit rate and thus reduce the system's disk accesses, cooperative caching leaves large memories at the clients and so can also maintain high hit rates in the

	Ethernet		155 Mbit/s ATM	
	Remote Memory	Remote Disk	Remote Memory	Remote Disk
Mem. Copy	250 µs	250 µs	250 µs	250 µs
Net Overhead	400 µs	400 µs	400 µs	400 µs
Data	6250 µs	6250 µs	400 µs	400 µs
Disk	--	14,800 µs	--	14,800 µs
Total	**6,900** µs	**21,700** µs	**1050** µs	**15,850** µs

Figure 1. Time to service a file system local cache miss from remote memory or disk for a slow network, *Ethernet*, and a faster network, *155 MBit/s ATM*. Local memory copy time is the measured time to read 8 KB from the file cache on a DEC AXP 3000/400. Network overhead times indicate round trip small packet latencies based on TCP times reported in [Mart94] for a Hewlett-Packard 9000/735 workstation. Ethernet data transfer figures makes the unrealistically optimistic assumption that data is transferred at the full 10 Mbit/s link speed (in reality, transfer times would likely be at least double those listed above for unswitched Ethernet). The ATM transfer time assumes the full 155 Mbit/s bandwidth is attained (also an optimistic assumption, but one likely to be met in a year or two as processor speeds continue to increase.) The disk transfer time is based on measured physical disk time (excluding queueing) for the fastest of three systems measured under real workloads by Ruemmler and Wilkes [Ruem93].

clients' local caches, saving network latencies compared to going to the server. Second, the server in the cooperative caching system will be less loaded since it can satisfy many requests with small packets to forward requests rather than large data transfers. Third, cooperative caching allows more flexible use of memory: since the memory is still physically located at the clients, it can also be used for client virtual memory as system demands warrant [Nels88]. Finally, cooperative cache systems are more cost effective than building a system with an extremely large server cache. For example, it would be significantly cheaper to add 16 MB of industry-standard SIMM memory to each of one hundred clients than it would be to buy a specialized server machine capable of holding the additional 1.6 GB of memory. We quantify the trade-offs between centralized and distributed memory in more detail at the end of Section 4.

Cooperative caching introduces a fourth level in the network file system's cache hierarchy. Not only can data be found in local memory, in server memory, or on server disk, but it can also be found in another client's memory. Depending on the cooperative caching algorithm used, this new level may be found between a client's local memory and the server memory or between the server memory and the disk. Note that we are examining cooperative caching assuming that clients cache file system data in their local memories but not on their local disks. For the fast networks of the future, it will be much faster for a client to fetch data from another client's memory than to fetch that data from local disk.

In this paper we make the assumption that all clients in the system are equally secure. We believe this to be a fair assumption in most LAN environments where all machines are administered in the same way. Our trust is no stronger than that given to clients in currently popular file systems like NFS; in either case, if the client's operating system is compromised, the client can issue unauthorized file system requests. The increasing availability of process migration among networks of workstations [Nich87, Doug91, Litz92, Zhou93] is likely to speed the trend towards trust within an administrative domain—if a system allows a user's jobs to be migrated among machines, that user's data may be cached in the memories of many machines regardless of cooperative caching.

This study has two goals. Our first goal is to ascertain whether cooperative caching can provide significant benefits under real workloads. Our trace-driven simulation approach contrasts with previous efforts to evaluate cooperative caching using synthetic workloads [Leff93a, Fran92]. Our second goal is to evaluate a range of algorithms to find practical algorithms to implement effective cooperative caching. Previous studies have focused on algorithms requiring global knowledge of client cache contents [Leff93a] or on algorithms that sacrifice performance by not coordinating the contents of client caches [Fran92].

Our primary result is that cooperative caching can improve file read performance by as much as 73% for the configurations and workloads studied. We further conclude that an algorithm called *N-Chance Forwarding* is a practical algorithm that achieves nearly all of the potential performance gains for these workloads.

Cooperative caching is designed to improve cache performance for file system reads. This technique does not address issues such as write performance and large file performance that are also important to end users of the file system. To study these and other issues, we are implementing cooperative caching as a part of the xFS project [Wang93, Dahl94]. Cooperative caching illustrates a primary design philosophy of xFS, the use of the vast aggregate resources of the system's clients to improve performance.

Section 2 describes the four cooperative caching algorithms we examine. Section 3 describes our simulation methodology and Section 4 examines our simulation results. We compare our work to previous efforts to improve file cache performance in Section 5. Finally, Section 6 summarizes our conclusions.

2. Cooperative Caching Algorithms

This paper examines four variations of cooperative caching in detail, covering a range of algorithm design decisions. Cooperative caching creates a new level in the file system storage hierarchy: remote client memory. Different cooperative caching algorithms could manage this new level in many different ways. Figure 2 illustrates four fundamental design questions and the relationship of the four algorithms to these questions. Although the algorithms we examine are by no means an exhaustive set of cooperative caching algorithms, the subset contains representative examples from large portion of the design space and includes a practical algorithm with close to optimal performance.

Note that the algorithms examined do not affect data storage reliability since they do not alter the write-through, write-delay, or write-back policy of the file system. Clients still send modified data to the server when they would have without cooperative caching, and the server commits data to disk as it would in a traditional system.

The rest of this section describes the four algorithms under scrutiny and then briefly discusses two other algorithms.

2.1. Direct Client Cooperation

A very simple cooperative caching approach, *Direct Client Cooperation,* allows an active client to use an idle client's memory as backing store. The active client forwards cache entries that overflow its local cache directly to an idle machine. The active client can then access this private remote cache to satisfy its read requests until the remote machine becomes active and evicts the cooperative cache. The system must define criteria for designating active and idle clients and must provide a mechanism for the former to locate the latter.

Direct Client Cooperation is appealing because of its simplicity—it can be implemented without server modification. As far as the server is concerned, a client utilizing remote memory appears to have a temporarily enlarged local cache. A drawback of this lack of server coordination is that active clients do not benefit from the contents of other active clients' memories. A client's data request must go to disk if the desired block no longer happens to be in the limited server memory even if another client is caching that block. As a result, the performance benefits of Direct Client Cooperation are limited, motivating the next algorithm.

2.2. Greedy Forwarding

Another simple cooperative caching approach, called *Greedy Forwarding*, treats the cache memories of all clients in the system as a global resource that may be accessed to satisfy any client's request, but the algorithm does not attempt to coordinate the contents of these caches. As for traditional file systems, each client manages its local cache greedily, without regard to the contents of the other caches in the system or the potential needs of other clients. If a client does not find a block in its local cache, it asks the server for the data. If the server has the required data in its memory cache, it supplies the data. Otherwise, the server consults a data structure listing the contents of the client caches. If any client is caching the required data, the server forwards the request to that client. The client receiving the forwarded request sends the data directly to the client that made the request. Note that the block is not sent back through the server since that would unnecessarily increase latency and add to the server's workload. If no client is caching the data, the request is satisfied by the server disk as it would have been if there were no cooperative caching.

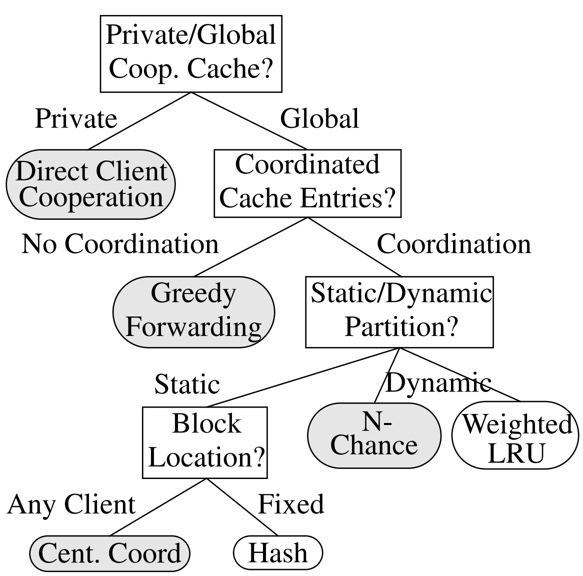

Figure 2. Cooperative caching algorithm design space. Each box represents a design decision while each oval represents an algorithm examined in this study. We focus on the four highlighted algorithms and do not consider the other two in detail due to space constraints.

With Greedy Forwarding the only change to the file system is that the server needs to be able to forward requests and the clients need to be able to handle forwarded requests; this support is also needed by the next two algorithms discussed. This server forwarding can be implemented with the data structures already present in systems implementing write-consistency with call backs [Howa88] or cache disabling [Nels88]. In those systems the server tracks the files being cached by each client so that it can take appropriate action to guarantee consistency when a file is modified. In this study we assume that cooperative caching extends a call back data structure to track the individual file blocks cached by each client to allow forwarding. For systems such as NFS whose servers do not maintain precise information about what clients are caching [Sand85], implementation of this directory may be simplified if its contents are taken as hints; some forwarded requests may be sent to clients no longer caching the desired block. In that case the client would inform the server of the mistake and the server would either forward the request to another client or get the data from disk.

Although the per-block forwarding table is larger than traditional per-file callback lists, the additional server memory overhead is reasonable since each entry allows the server to leverage a block of client cache. For instance, if the forwarding table is implemented as a hash table with each hash entry containing a four byte file identifier, a four byte block offset, a four byte client identifier, a four byte pointer for linked-list collision resolution, and two four byte pointers for a doubly linked LRU list, the server would require 24 bytes for every block of client cache. For a system caching 8 KB file blocks, such a data structure would consume 0.3% as much memory as it indexes. For a system with 64 clients each with 32 MB of cache, the server could track the contents of the 2 GB distributed cache with a 6 MB index.

Greedy Forwarding is also appealing because it preserves fairness—clients manage their local resources for their local good while still deriving benefit from the other clients. On the other hand, this lack of coordination among cache contents may cause unnecessary data duplication, not taking the best advantage of the system's memory to avoid disk accesses [Leff91, Fran92]. The next two algorithms attempt to address this lack of coordination.

2.3. Centrally Coordinated Caching

Centrally Coordinated Caching adds coordination to the Greedy Forwarding algorithm by statically partitioning each client's cache into a locally managed section, managed greedily by that client, and a globally managed section, coordinated by the server as an extension of its central cache. If a client does not find a block in its locally managed cache, it sends the request to the server. If the server has the desired data in memory, it supplies the data. Otherwise the server checks to see if it has stored the block in centrally coordinated client memory. If it locates the data in client memory, it forwards the request to the client storing the data. If all else fails, the server supplies the data from disk.

Centrally Coordinated Caching behaves very much like physically moving memory from the clients to the server. The server manages the globally managed fraction of each client's cache using a global replacement algorithm. When the server evicts a block from its local cache to make room for data fetched from disk, it sends the victim block to replace the least recently used block among all of the blocks in the centrally coordinated distributed cache. When the server forwards a client request to a distributed cache entry, it renews the entry on its LRU list for the global distributed cache. Unless otherwise noted we simulate a policy where the server manages 80% of each client's cache.

The primary advantage of Centrally Coordinated Caching is the high global hit rate that it can achieve through global management of the bulk of its memory resources. The main drawbacks to this approach are that the clients' local hit rates may be reduced since their local caches are effectively made smaller and also that the central coordination may impose significant load on the server.

2.4. N-Chance Forwarding

The final algorithm that we quantitatively evaluate, *N-Chance Forwarding*, dynamically adjusts the fraction of each client's cache managed cooperatively, depending on client activity. The N-Chance algorithm modifies the Greedy Forwarding algorithm to have clients cooperate to preferentially cache *singlets*, blocks stored in only one client cache. Except for singlets, N-Chance Forwarding works like Greedy Forwarding.

N-Chance Forwarding attempts to avoid discarding singlets from client memory. When a client discards a block, it checks to see if that block is the last copy cached by any client. This check may require a message to the server or it may be done by consulting some flags associated with each block as described below. If the block is a singlet, rather than throw the block away, the client sets the block's *recirculation count* to n, forwards the data to a random peer, and then sends the server a message telling it that the block has moved. The peer that receives the data adds the block to its LRU list as if the block had been recently referenced. If a recirculating block reaches the end of the LRU list, its count is decremented and it is forwarded again unless the count is now zero, in which case it is simply discarded. If a client references a singlet, it resets the block's recirculation count and caches the data normally while the client that had been cooperatively caching the singlet discards the block from its cache.

The parameter n indicates how many times a singlet should be allowed to recirculate through different clients' LRU lists without being referenced before finally being discarded. Greedy forwarding is simply the degenerate case of this algorithm with $n = 0$. Unless otherwise noted we use $n = 2$ for our simulations.

This algorithm provides a dynamic trade-off for each client cache's allocation between local data, data being cached because the client referenced it, and global data, singlets being cached for the good of aggregate system performance. Active clients will tend to force any global data sent to them out of their caches quickly as local references displace global data. Idle clients will tend to accumulate global blocks and hold them in memory for long periods of time. An enhancement to this algorithm might be to preferentially forward singlets to idle clients to avoid disturbing active clients. For this study, however, clients forward singlets uniformly randomly to the other clients in the system.

An implementation of this algorithm must prevent a ripple effect where a block forwarded from one client displaces a block to another client and so on. Note that in the common case, the displaced block is not the last copy of data and so no ripple occurs, however we simulate a policy that prevents deep recursion from ever occurring: a client receiving a recirculating block is not allowed to forward a block to make space. When a client receives such a block, it uses a modified replacement algorithm, discarding its oldest duplicated block. If the cache contains no duplicated blocks, the client discards the oldest recirculating block with the fewest recirculations remaining.

Several optimizations to this algorithm reduce the amount of communication with the server. First, on a cache miss, the client combines its messages to the server, updating the server's directory of client cache contents in the same message that requests data to satisfy the miss. This update indicates what block the client has discarded from its cache and where, if anywhere, that block has been forwarded.

The second set of optimizations reduces the number of messages asking the server if a block is the last cached copy when a client is deciding if a block should be recirculated or discarded. First, any block whose recirculation count is set must be a singlet, so no server message is necessary to decide its fate. For non-recirculating blocks, the client must usually send a message to the server, but once it has determined if the block is a singlet the client will discard or forward it, so only one message is needed during a block's lifetime in a cache. In the special case where the client is making space for a singlet that was kicked out of another client's cache, it will not discard blocks that it discovers to be singlets, but it will mark those blocks as singlets (without setting the recirculation count) so that it will not need to ask the server again unless another client references the block. If another client references such a block, the server forwards the request to the singlet, and the client resets the singlet flag.

The main advantage of N-Chance Forwarding is that it provides a simple dynamic trade-off between each client's private cache data and the data being cached for the good of the overall system. Favoring singlets provides better performance than the simple Greedy algorithm since discarding a singlet is potentially more expensive than discarding a duplicated block; later references to the duplicate can still be satisfied from another client's memory [Leff91]. A potential disadvantage of this approach is that a given block may be bounced among multiple caches while living in the "cooperative" portion of the caches, resulting in unnecessary system load.

2.5. Other Algorithms

We considered two other cooperative caching algorithms. Performance measurements for these algorithms are omitted from this report because each performed similarly to one of the other algorithms we examined.

Hash-Distributed Caching differs from Centrally Coordinated Caching in that Hash-Distributed Caching partitions the centrally managed cache based on block identifiers, with each client managing one partition of the cache. The central server sends blocks displaced from its local cache to a client selected by hashing on the block's identifier. On a local miss a client accesses this distributed cache by sending its request directly to the appropriate client. That client supplies the data if it is currently caching that block, or forwards the request to the server if it does not have that block. Our simulations indicate that Hash-Distributed caching provides nearly identical hit rates compared to Centrally Coordinated caching; fixed partitioning of the centrally managed cache does not hurt the hit rate. The main advantage of Hash-Distributed caching over Centrally Coordinated Caching is that Hash-Distributed caching significantly reduces server load since many requests satisfied by the cooperative cache don't go through the server.

We also examined *Weighted LRU*, a dynamic algorithm that attempts to replace the object with the globally lowest value/cost ratio. As with N-Chance, objects that are duplicated in multiple client caches are not very valuable since even if one copy is discarded, the data may be fetched from another client's memory. On the other hand the last cached copy of a block is very valuable since its loss might cause a disk access. The opportunity cost of keeping an object in memory is the cache space it consumes until the next time the block is referenced [Smit81], approximately the time since the last reference. Thus, weighted LRU explicitly balances keeping frequently used duplicates to avoid network accesses against keeping less frequently used singlets to avoid disk accesses. For our traces, however, response time was slightly worse than for the substantially simpler N-Chance Forwarding.

3. Simulation Methodology

We use trace-driven simulation to evaluate the cooperative caching algorithms. Our simulator tracks the state of all caches in the system and monitors the requests and hit rates seen by each client. We assume a cache block size of 8 KB, and we do not allow partial blocks to be allocated even for files smaller than 8 KB. We verified our simulator by using the synthetic workload described in [Leff93a] as input.

We calculate response times by multiplying the local memory, remote client memory, server memory, and disk hit rates by the times it takes to access those memories. Our baseline technology assumptions are the same as for the 155 Mbit/s ATM columns of Figure 1, that an 8 KB block can be fetched from local memory in 250 µs, that a fetch from remote memory takes an additional 400 µs plus 200 µs per network hop, and that an average disk access takes a further 14,800 µs. Figure 3 summarizes access times to different resources for the algorithms. In Section 4.3 we examine the sensitivity of our results to technology changes. Note that we do not include any queueing delays in our response time figures. Since the most attractive algorithms studied do not increase server load and since emerging high performance networks use a switched topology, we would not expect queueing to significantly alter our results.

To maintain data consistency on writes, we assume that data modifications are written through to the central server and that client caches are kept consistent using a write-invalidate protocol [Arch86]. Since we focus on read performance, a delayed write or write back policy would not affect our results.

	Local Mem.	**Remote Client Mem.**	**Server Mem.**	**Server Disk**
Direct	250 µs	1050 µs	1050 µs	15,850 µs
Greedy	250 µs	1250 µs	1050 µs	15,850 µs
Central	250 µs	1250 µs	1050 µs	15,850 µs
N-Chance	250 µs	1250 µs	1050 µs	15,850 µs

Figure 3. Access times for the different levels in the memory hierarchy for different cooperative caching algorithms. The differences among the Remote Client times for the different algorithms depends on the number of network hops to reach the data for the algorithm.

For most of the results in this paper, we use traces five and six from the Sprite workload, described in detail by Baker et al. [Bake91]. The Sprite user community included about 30 full time and 40 part time users of the system. These users included operating systems researchers, computer architecture researchers, VLSI designers, and "others" including administrative staff and graphics researchers. These traces list the activity of 42 client machines and one server over a two day period measured under the Sprite operating system.[1] They contain over 700,000 read and write block accesses, and we use the first 400,000 accesses (a little over a day) to warm the simulated caches. Section 4.4 describes simulation results for an additional workload.

When reporting our results, we compare against a set of baseline cache management assumptions and against an unrealistic best case model. The *base case* assumes that each client has a cache and that the central server also has a cache, but that no cooperative caching is used. The unrealizable *best case* assumes that the cooperative caching algorithm is able to achieve a global hit rate as high as if all client memory were managed as a single global cache, but that the local hit rates are as if all client memory were managed as a private local cache. This best case provides a lower bound for the response time for cooperative caching algorithms that physically distribute client memory equally to each client and that use LRU replacement. We simulate this algorithm by doubling each client's local cache and allowing the clients to manage half of it locally and allowing the server to manage half of it globally as it does for the centrally coordinated case. For the best case we assume that data found in remote client memory is fetched with three network hops (request, forward, and reply) for a total of 1250 μs per remote memory hit.

4. Simulation Results

This section presents the main results from our simulation studies of cooperative caching. Section 4.1 compares the different cooperative caching algorithms to the base case, to each other, and to the unrealizable best case. For clarity, this comparison is made assuming a particular set of parameters for each algorithm, for a given set of technology and memory assumptions, and under a single workload. Section 4.2 examines the individual algorithms more closely, studying different values for the algorithms' parameters. Section 4.3 examines the sensitivity of our results to technology and memory assumptions such as the client cache size, server cache size, and hardware performance. Section 4.4 examines the algorithms under an additional workload. Finally, Section 4.5 summarizes our results, highlights our conclusions, and compares cooperative caching to moving more of the system's memory to the server.

[1] Baker et al.'s traces also included requests to three auxiliary servers. We just used accesses to the main server, 81% of the trace.

4.1. Comparison of Algorithms

This section compares the algorithms' response times, hit rates, server loads, and impact on individual clients. Our initial comparison of the algorithms fixes the client caches to be 16 MB per client and fixes the server cache to be 128 MB for the Sprite workload. For the Direct Cooperation algorithm we made the optimistic assumption that clients do not interfere with each other when they use remote caches; we simulate this assumption by allowing each client to maintain a permanent remote cache of a size equal to its local memory size, effectively doubling the size of each client's cache. For the Central Coordination algorithm, we assume that 80% of each client's cache memory is dedicated to the cooperative cache and that 20% is managed locally. For the N-Chance algorithm, we choose a recirculation count of two; unreferenced data will be passed to two random caches before being purged from memory. We will examine why these are appropriate parameters in Section 4.2.

Figure 4 illustrates the response times for each of the four algorithms being examined and compares these times to the base case on the left and the best case on the right. Direct Cooperation provides only a small speedup of 1.05[2] compared to the base case despite our optimistic assumptions for this algorithm. Greedy Forwarding shows a modest but significant performance gain, with a speedup of 1.22. The two algorithms that coordinate cache contents to reduce redundant cache entries show more impressive gains. Central Coordination provides a speedup of 1.64 and N-Chance Forwarding provides a performance

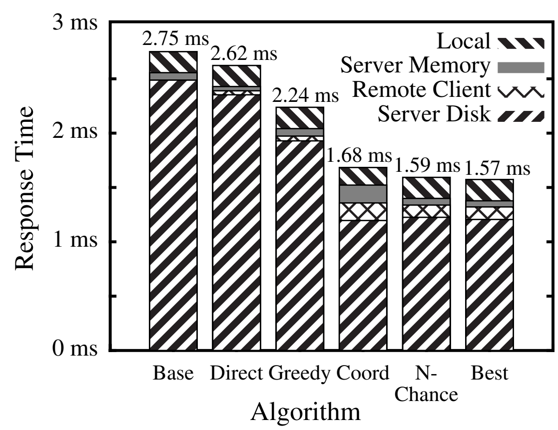

Figure 4. Average block read time. Each bar represents the time to complete an average read for one of the algorithms. The segments of the bars show the fraction of the total read time for data accesses satisfied by *Local* memory, *Server Memory*, *Remote Client* memory, or *Server Disk*.

[2] All speedup and performance improvement figures in this paper use the terminology in [Henn90]. Speedup is defined as the execution time of the slower algorithm divided by the execution time for the faster algorithm. Performance improvement percentages are calculated by subtracting 1.00 from the speedup and then multiplying by 100 to get a percentage.

improvement of 1.73. Both coordinated algorithms are within 10% of the ideal cooperative caching response time.

Two conclusions seem apparent from Figure 4. First, disk accesses are the dominant source of latency for the base case, so efforts like cooperative caching that improve the overall hit rate will be beneficial. Second, the most dramatic improvements in performance come from the coordinated algorithms, where the system makes an effort to reduce the duplication among cache entries to improve the overall hit rate.

Figure 5 provides additional insight into the performance of the algorithms by illustrating the access rates at different levels of the memory hierarchy. The total height of each bar represents the miss rate for each algorithm's local cache. The base, Direct Cooperation, Greedy, and best case algorithms all manage their local caches greedily and so have identical local miss rates of 22%.[3] Central Coordination has a local miss rate of 36%, over 60% higher than the baseline local miss rate. This algorithm makes up for this local deficiency with aggressive coordination of most of the memory in the system, providing combined memory miss rates essentially identical to those achieved in the best case, with just 7.6% of all requests going to disk. This disk access rate is less than half of the 15.7% rate for the base caching scheme. The N-Chance algorithm's emphasis on holding onto the last data copies hurts the local miss rate by a surprisingly small amount; recirculation increases the local miss rate from 22% to 23%. N-Chance also provides a very low overall disk access rate of 7.7%.

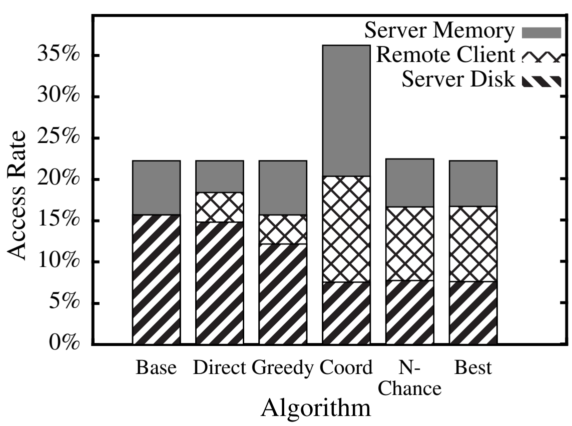

Figure 5. Fraction of requests satisfied at each level of the memory hierarchy for different algorithms. The total height of the bar is the local miss rate for each algorithm. The sum of the *Server Disk* and *Remote Client* segments shows the miss rate for the combined local and server memories. The bottom segment shows the miss rate once all memories are included, i.e. the disk access rate.

[3] The simulated local miss rate is lower than the 40% miss rate measured for the Sprite machines in [Bake91] because we simulate larger caches than the average 7 MB caches observed in that study and because these larger caches service requests to only one server.

A comparison between the static memory partition algorithm, Centralized Coordination, and the dynamic partition algorithm, N-Chance Forwarding, illustrates that both the local and global miss rates must be considered in evaluating these algorithms. Although the static algorithm provides the lower disk access rates, it provides this low miss rate at significant cost to its local cache performance. The N-Chance algorithm coordinates a smaller fraction of the client cache contents, protecting the local cache hit rate but sacrificing some global hits.

Another important metric of comparison is the server load imposed by the algorithms. If a cooperative caching algorithm significantly increases server load, increased queueing delays might reduce the performance gains. Figure 6 illustrates the relative server loads for the algorithms.

Since we are primarily interested in verifying that cooperative caching's increased server coordination doesn't greatly increase server load, we make a number of simplifications in our server load calculations. First, we do not include the load for write-backs, deletes, file attribute requests, or other sources of server load in the load comparison. Including these loads would add equally to the load for each algorithm, reducing the relative differences among them.

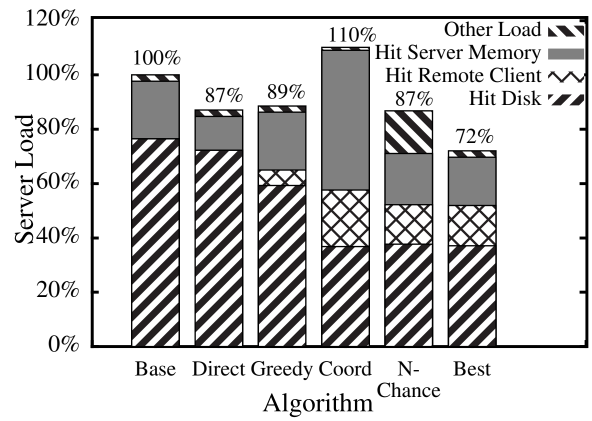

Figure 6. Server loads for the algorithms as a percentage of the baseline no cooperative caching server load. The *Hit Disk* segment includes both the network and disk load for all requests satisfied at the server disk. The *Hit Remote Client* segment shows the server load for receiving and forwarding requests to remote clients. The *Hit Server Memory* segment includes the cost of receiving requests and supplying data from the server's memory. Local hits generate no server load. The *Other Load* segment includes server overhead for invalidating client cache blocks and for answering client queries (e.g. N-Chance asks, "Is this block the last cached copy").

Also, we base the server load calculations on the network messages and disk transfers made by the server for each algorithm. We assume that a network message overhead costs one load unit and that a block data transfer costs two load units. A small network message therefore costs one unit; a network data transfer costs one for overhead plus two for data transfer for a total of three units. We also charge the server two load units for a disk data transfer.

The results of the server load measurements suggest that most of the cooperative caching algorithms will not significantly increase server load and that our response time approximation of ignoring queueing delay should provide valid comparisons with the base case. The Centralized Coordinated algorithm does appear to increase server load somewhat, at least under these simple assumptions. This increase is because the centralized algorithm significantly increases the local miss rate, and all local misses are sent to the server. More detailed measurements would have to be made to determine if the centralized algorithm can be implemented without increasing server queueing delays.

A final comparison among the algorithms examines individual client performance rather than the aggregate average performance. Figure 7 illustrates the relative performance for individual clients under each cooperative caching algorithm compared to that client's performance in the base case. The graph positions data points for the clients so that inactive clients appear on the left of the graph and active clients on the right. Speedups or slowdowns for inactive clients may not be significant both because the clients are spending relatively little time waiting for the file system in either case and because these inactive clients' response times can be significantly affected by adding just a few disk accesses.

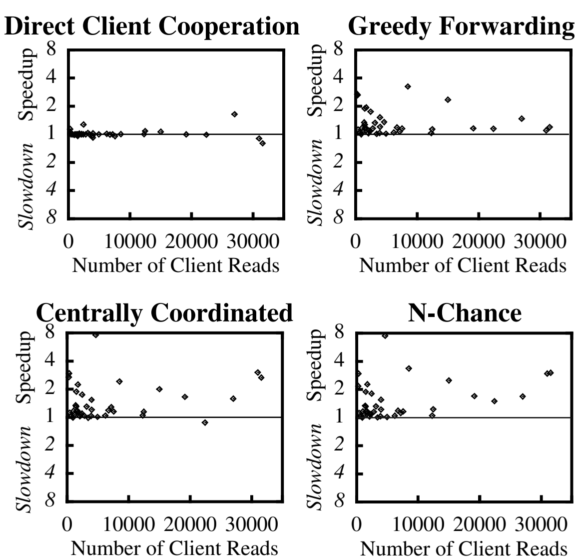

Figure 7. Performance of each individual client. Each point represents the speedup or slowdown seen by one client for a cooperative caching algorithm compared to that client's performance in the base case. Speedups are above the line and slowdowns are below it. A client's slowdown is defined as the inverse of its speedup if its speedup is less than one. The x-axis indicates the number of read requests made by each client; relatively inactive clients appear near the left edge of the graph, and active clients appear on the right.

One important aspect of individual performance is fairness: are any clients significantly worse off because they contribute resources to the community rather than managing their local caches greedily? Fairness is important because even if the average client performance is improved, some clients may refuse to participate in cooperative caching if their performance would be worse.

The data in Figure 7 suggest that fairness is not a widespread problem for this workload. Direct Client Cooperation and Centrally Coordinated Caching each slow a few clients by modest amounts. Greedy Forwarding and N-Chance Forwarding do no harm to any clients in this workload.

Although we would expect the two algorithms with greedy client cache management to always be fair, Direct Client Cooperation causes a few clients to suffer up to 25% worse performance than they had without the additional cooperative cache memory. These clients do not benefit greatly from their cooperative cache memory but have lower server cache hit rates under Direct Client Cooperation than in the base case. The lower server hit rates occur because the accesses to the server cache by all the clients in the system are filtered by their effectively larger local caches, reducing the correlation among client access streams at the server.

Although both the N-Chance and Centrally Coordinated algorithms disturb local greedy caching, their significant improvements in global caching provide a net benefit to almost all clients. N-Chance Forwarding hurts no clients for this workload, and Centrally Coordinated Caching damages the response of one client by 19%. Neither of these algorithms help a client whose working set fits completely into its local cache, but such a client can be hurt by interference with its local cache contents. Since N-Chance Forwarding interferes with local caching less than Centrally Coordinated Caching as was indicated in Figure 5, it is less likely to be unfair to individual clients. Other algorithms that statically partition client memory, such as Hash Distributed Caching or physically moving cache from the clients to the server, would suffer from the same vulnerability as Centrally Coordinated Caching.

4.2. Detailed Algorithm Analysis

This subsection examines the cooperative caching algorithms in more detail and evaluates their sensitivity to algorithm-specific parameters.

4.2.1 Direct Client Cooperation

Although Direct Client Cooperation is appealingly simple, achieving even the modest 5% response time improvement seen above may be difficult. We based the above results on the optimistic assumption that clients could recruit sufficient remote cache memory to double their caches without interfering with each other. In reality the algorithm must meet three challenges to provide even these modest gains.

The first difficulty for Direct Client Cooperation is that clients may not be able to find enough remote memory to significantly affect performance. Figure 8 plots Direct Cooperation response time as a function of the amount of remote memory recruited by each client. If, for instance, clients can only recruit enough memory to increase their cache size by 25% (4 MB), the response time improvement drops to under 1%. Significant speedups of 40% are only achieved if each client is able to recruit about 64 MBs—four times the size of its local cache.

Interference from other clients is likely to further limit Direct Client Cooperation benefits. When a client donating memory becomes active, it will flush any other client's data from its memory. A client trying to take advantage of remote memory sees a series of temporary caches, reducing its hit rate since a new cache will not be warmed with its data. Studies of workstation activity [Nich87, Thei89, Doug91, Mutk91, Arpa94] suggest that although many idle machines are usually available, the length of their idle periods can be relatively short. A possible solution to this problem would be to send the evicted data to a new idle client rather than discarding it, but that would increase the system's complexity.

A final challenge for Direct Client Cooperation is dynamically selecting which clients should donate memory and which should utilize remote memory. This problem appears solvable; if only the most active 10% of clients are able to recruit a cooperative cache they would achieve 85% of the maximum benefits available to Direct Client Cooperation for this trace. On the other hand, the implementation of a recruiting mechanism detracts from the algorithm's simplicity and may require server involvement.

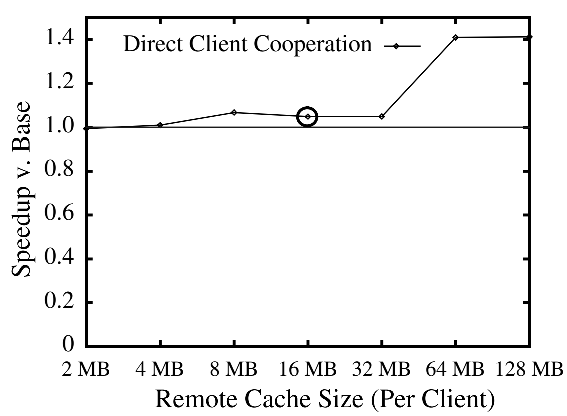

Figure 8. Direct Client Cooperation speedup compared to the base case as a function of each client's remote cache size. The circle indicates the result for the 16 MB per client remote cache assumed for this algorithm in the previous section.

4.2.2 Greedy Forwarding

Although the performance gains for the greedy algorithm are modest, the greedy algorithm may still be attractive because of its simplicity, because it does not increase server load, and because it is fair. In other words, this 22% performance improvement comes essentially for free once the clients and server have been modified to forward requests and the server's callback state is expanded to track individual blocks.

4.2.3 Centrally Coordinated Caching

Centrally Coordinated Caching can provide significant speedups and very high global hit rates. On the other hand, devoting a large fraction of each client's cache to Centrally Coordinated Caching reduces the local hit rate, potentially increasing the server load and reducing overall performance for some of the clients.

The fraction of each client's cache that is treated as a centralized resource determines the effectiveness of the algorithm. Figure 9 plots the overall response time as the centrally coordinated fraction is increased. As the fraction is increased, the global hit rate improves, reducing the time spent fetching data from disk. At the same time, the local hit rate decreases, driving up the time spent fetching from remote caches. These two trends create a response

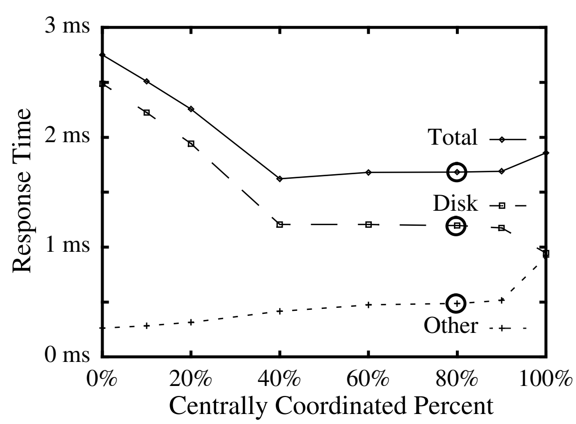

Figure 9. Response time for Centrally Coordinated Caching depends on the percent of the cache that is centrally coordinated. 0% corresponds to the baseline no cooperative caching case. The *Total* time is the sum of the time for requests that are satisfied by the *Disk* and the time for *Other* requests that are satisfied by a local or remote memory. The rest of this study uses a centrally coordinated fraction of 80% for this algorithm, indicated by the circled points.

time plateau when 40% to 90% of each client's local cache is managed as a global resource. Note that these measurements do not take increased server load into account; increasing the centrally managed cache fraction also increases the load on the central server as local caches satisfy fewer requests. This effect may increase queueing delays at the server as the centrally-managed fraction is increased, reducing the overall speedups and pushing the "break-even" point towards smaller centrally managed fractions.

We chose to use 80% as the default centrally managed fraction because that appears to be the more "stable" part of the plateau under different workloads and cache sizes. For instance, the plateau runs from 60% to 90% for the same workload but with 8 MB client caches. A high centrally managed fraction tends to achieve good performance because of the large disparity between disk and network memory access times compared to the gap between network and local memory. If the network were slower, a smaller percentage would be appropriate.

4.2.4 N-Chance Forwarding

N-Chance Forwarding also provides very good overall performance, but it does so by improving overall hit rates without significantly reducing local hit rates. This algorithm also has good server load and fairness characteristics.

Figure 10 plots response time against the recirculation count parameter, n, for this algorithm. The largest improvement comes when n is increased from zero (the Greedy algorithm) to one. Increasing the count from one to two also provides a small improvement while larger values make little difference. Relatively low values for n are effective since data that is recirculated through a random cache often lands in a relatively idle cache and so remains in memory for a significant period of time before being flushed. When the parameter is two, the random forwarding almost always gives a block at least one relatively long period of time in a mostly idle cache. Higher values make little additional difference both because few blocks need a third try to find an idle cache and because the algorithm sometimes discards old cache items without recirculating them all n times to avoid a "ripple" effect among caches.

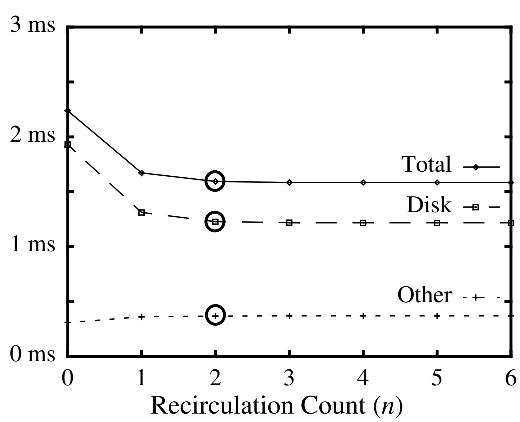

Figure 10. Response time for N-Chance algorithms depends on number of times unreferenced blocks are recirculated through random caches. Zero corresponds to the Greedy algorithm (no recirculation). The *Total* time is the sum of the time for requests that are satisfied by going to *Disk* and *Other* requests that are satisfied by a local or remote memory. The rest of this study uses a recirculation count of two for this algorithm, indicated by the circled points.

4.3. Sensitivity

This subsection explores the sensitivity of the results we present here to assumptions about each client's cache size, the central server's cache size, and the performance of the LAN over which the machines are connected.

Figure 11 plots the performance of the algorithms as a function of the size of each client's local cache. The graph shows that the two coordinated algorithms, Centralized Coordination and N-Chance Forwarding, perform well as long as caches are reasonably large. If caches are too small, however, coordinating the contents of client caches provides little benefit because borrowing any client memory causes a large increase in local misses with little aggregate benefit in reducing disk accesses. The simple Greedy algorithm also performs relatively well over the range of cache sizes.

Figure 12 illustrates the effect of varying the size of the cache at the central server. Increasing the server cache size significantly improves the base no cooperative caching case, while only modestly improving the performance of the cooperative algorithms that already have good global hit rates. For sufficiently large server caches, cooperative caching provides no benefit once the server cache is about as large as the aggregate client caches. Such a large cache, however, would double the system's memory cost compared to using cooperative caching. Note that when the server cache is very large Centrally Coordinated Caching performs poorly because of its degraded local hit rate.

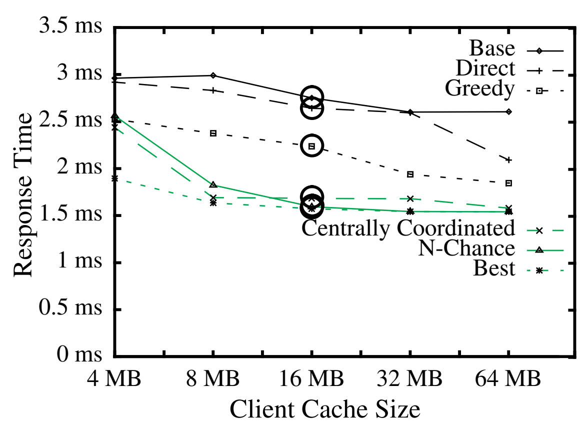

Figure 11. Response time as a function of client cache memory for the algorithms. Other graphs in this study have assumed a client cache size of 16 MB (circled).

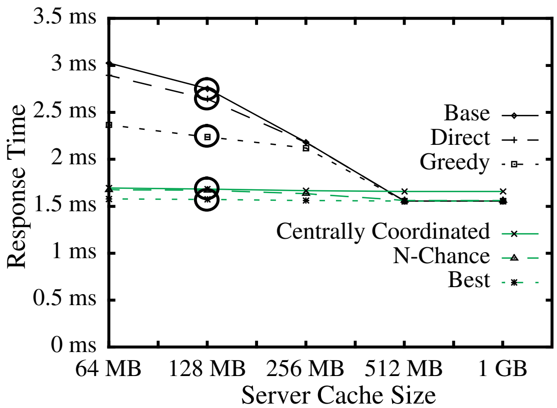

Figure 12. Response time v. total central server cache size. The circled points highlight the results for the default 128 MB server assumption.

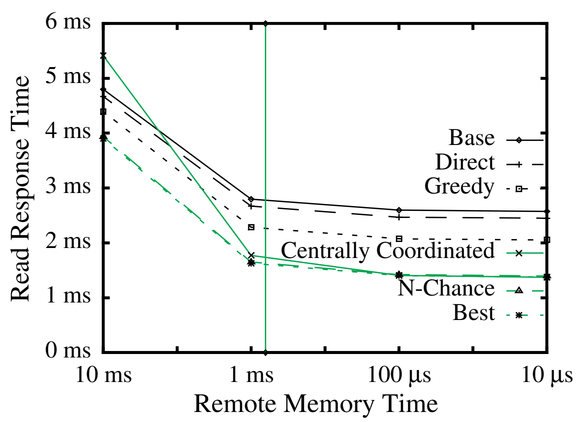

Figure 13. Response time as function of network speed. The X axis is the round trip time to request and receive an 8 KB packet. Disk access time is held constant at 15 ms and the memory access time is held constant at 250 µs. For the rest of this study we have assumed 200 us per hop plus 400 us per block transfer for a total remote fetch time of 800 us (request-reply excluding memory copy time), indicated by the vertical bar. The *N-Chance* and *Best* lines nearly overlap over the entire range of the graph.

The emergence of fast networks means that the time is ripe to begin utilizing cooperative caching in file systems. Although Ethernet-speed networks are too slow to get large benefits from cooperative caching, emerging ATM networks promise to be fast enough to see significant improvements. Figure 13 plots response time as a function of the network time to fetch a remote block. For an Ethernet-speed network, where a remote data access can take nearly 10 ms, the maximum speedup seen for a cooperative caching algorithm is 20%. If network fetch time were reduced to 1 ms, for instance by using a fast ATM network, the peak speedup increases to 70%. This graph shows little benefit from reducing network block fetch time below 100 µs because once the network is that fast,

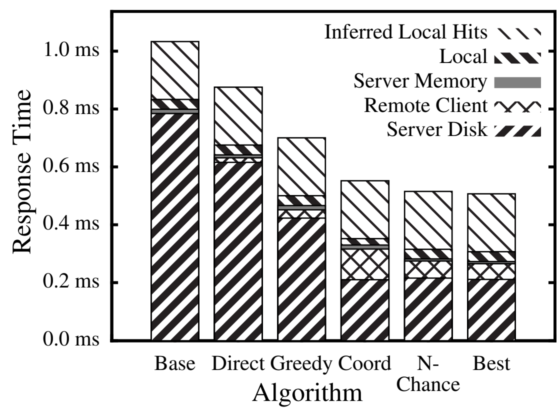

Figure 14. Response time for algorithms under the Auspex workload. The *Inferred Local Hits* segment indicates an estimate of the amount of time spent processing local hits that do not appear in the incomplete Auspex traces assuming that the traced system had an 80% local hit rate.

network times are not a significant source of delay compared to the constant memory and disk times.

Although either coordinated algorithm can provide nearly ideal performance when the network is fast, N-Chance Forwarding appears to be much less sensitive to network speed than Centrally Coordinated Caching. Centrally Coordinated Caching only makes sense in environments where accessing remote data is much closer to accessing local data than going to disk. Otherwise, its reduced local hit rate outweighs the increased global hit rate.

4.4. Berkeley Auspex Workload

The response time results for a second workload, called Berkeley Auspex, appear in Figure 14. The Berkeley Auspex workload traces the NFS file system network requests for 237 clients in the U.C. Berkeley Computer Science Division that are serviced by an Auspex file server. This workload is interesting because it follows the activity of a larger number of clients and includes a longer period of time than any of the Sprite traces. The large number of clients provide an extremely large pool of memory for cooperative caching to exploit. The traces cover a 6 day period and include five million read and write events of which we use the first million to warm the caches.

The trace was taken by snooping on the network; because it does not include local hits, we must adjust the simulation to account for the missing local accesses. We use Smith's Stack Deletion method [Smit77] to approximate the response time results based on this incomplete trace. Smith found that omitting references that hit in a small cache makes little difference in the number of faults seen when simulating a larger cache. The actual miss rate can be accurately approximated by dividing the number of faults seen when simulating the reduced trace by the actual number of references in the full trace.[4] As a further refinement, we utilize the *read attribute* requests present in our trace to more accurately simulate the local client LRU lists. NFS uses read attribute requests to validate cached blocks before referencing them. We can therefore use read attribute requests as a hint that the cached blocks of a file are being referenced even though the block requests do not appear in our trace. The attribute requests still provide only an approximation—an attribute cache hides attribute requests validated in the previous three seconds, and not all read attribute requests really signify that a file's cached blocks are about to be referenced—but they do allow us to infer some of the "missing" block hits.

Although the results for the Auspex workload are only approximate, they support the results seen for the Sprite workloads. The relative ranking of the algorithms under the Auspex workload is the same as it was for the Sprite workload: Centrally Coordinated Caching and N-Chance Forwarding work nearly as well as the best case, and the Greedy algorithm also provides significant speedups. Direct Cooperation provides more modest gains. This result is insensitive to the hit rate assumed. The predicted speedup factors for the Auspex workload does depend on the hit rate assumed but is significant over a wide range of assumed local hit rates.

4.5. Summary

N-Chance Forwarding is a relatively simple algorithm that appears to provide very good performance over a wide range of conditions. Centrally Coordinated Caching and the omitted Hash Distributed Caching can also provide very good performance, but they are more likely to degrade the performance of individual clients and depend heavily on fast network performance to make up for the reduced local hit rates they impose. The Weighted LRU algorithm (results omitted) performs similarly to the N-Chance algorithm, but it is more complicated and may also load the server with requests for information about global state.

The Greedy Forwarding algorithm appears to be the algorithm of choice if simplicity is the primary concern. Although the Direct Cooperation algorithm is also simple, satisfying the demands of cooperative caching without interfering with other client activities may be difficult, particularly since the Direct algorithm would have to locate 32 MB to 64 MB of remote memory per active client to equal the Greedy algorithm's performance.

Finally, consider the alternative to cooperative caching: physically moving more memory to the central server. This approach is very similar to the Centrally Coordinated algorithm and provides similar performance; moving 80% of client memory to the server yields improvements of 66% and 93% over the standard memory distribution for the Sprite and Auspex workloads respectively. These speedups are nearly equal to the speedups for the N-Chance algorithm, but fall short of equalling the N-Chance algorithm because of the reduced local hit rates resulting from smaller local caches. Moving large fractions of the clients' caches to the server has a number of other disadvantages compared to a good cooperative caching algorithm such as N-Chance Forwarding:

- Static allocation of the global/local caches is more likely to provide bad performance for some individual clients as was seen for Centrally Coordinated Caching in Figure 7.
- A system with more cache memory at the server and less at the clients would be very sensitive to network speed as was seen for Centrally Coordinated Caching in Figure 13. As the ratio of network performance to local memory performance is reduced, moving memory to the server becomes less attractive.
- Reducing the size of client local caches and transferring more data from the server can increase server load. The read load for a traditional caching system with the enlarged central cache is 50% higher than for N-Chance Forwarding under the Sprite workload.
- Memory physically moved for use as central server file system cache cannot be used by clients for other activities. Cooperative caching, on the other hand, may allow client cache memory to be released for use as client virtual memory as system demands warrant [Nels88].
- Configuring servers with large amounts of memory may be less cost-effective than spreading the same amount of memory among the clients. For instance, 80% of the 16 MB of cache memory for the 237 clients in the Auspex trace would be 3 GB of memory, demanding an extremely expandable and potentially expensive server.

5. Related Work

This paper evaluates the performance benefits and implementation issues of cooperative caching. Its primary contributions are evaluating realistic management algorithms under real file system workloads and a systematic exploration of implementation options.

Leff et al. [Leff91, Leff93a, Leff93b] investigate remote caching architectures, a form of cooperative caching, using analytic and simulation-based models under a synthetic workload. Two important characteristics of their workload were that the access probabilities for each object by each client were fixed over time and that each client knew what these distributions were. Leff found that if clients base their caching decisions on global knowledge of what other clients are caching, they could achieve nearly

4. Unfortunately, the Auspex trace does not indicate the total number of references. For the results in Figure 14 we assume a "hidden" hit rate of 80% (to approximate the 78% rate simulated for the Sprite trace), giving a maximum speedup of 2.00 for N-Chance Forwarding. If the local hit rate were higher, all of the bars would have a slightly larger constant added and the differences among the algorithms would be smaller (e.g. a 90% local hit rate reduces the N-Chance speedup to 1.67). If the local hit rate were lower, the differences would be magnified (e.g. a 70% local hit rate gives an N-Chance speedup of 2.20).

ideal performance, but that if clients made decisions on a strictly local basis, performance suffered greatly.

This paper differs from the Leff studies in a number of important ways. First, this paper uses actual file system reference traces as a workload, allowing us to quantify the benefits of cooperative caching realizable under real workloads. A second major feature of this study is that we have focused on getting good performance while controlling the amount of central coordination and knowledge required by the clients rather than focusing on optimal replacement algorithms.

Franklin et al. [Fran92] examined cooperative caching in the context of client-server data bases where clients were allowed to forward data to each other to avoid disk accesses. The study used synthetic workloads and focused on techniques to reduce replication between the clients' caches and the server cache. The server did not attempt to coordinate the contents of the clients' caches to reduce replication of data among the clients. Their "Forwarding—Sending Dropped Pages" algorithm is similar to our N-Chance Forwarding algorithm, but they send the last copy of a block to the server cache rather than to another client.

Blaze [Blaz93] proposed allowing file system clients to supply hot data to each other from their local on-disk file caches. The focus of this work was on reducing server load rather than improving responsiveness. He found that the use of client-to-client data transfers allowed *dynamic hierarchical caching* and avoided the store and forward delays experienced by static hierarchical caching systems [Munt92].

The idea of forwarding data from one cache to another has also been used to build scalable shared memory multiprocessors. DASH hardware implements a scheme similar to Greedy Forwarding for dirty cache lines [Leno90]. This policy avoids the latency of writing dirty data back to the server when it is shared. The same optimization could be used for a cooperative caching file system that uses delayed writes. Several "Cache Only Memory Architecture" (COMA) designs have also relied on cache to cache data transfers [Hage92, Rost93].

Other researchers have examined the idea of using remote client memory rather than disk for virtual memory paging. Felten and Zahorjan [Felt91] examined this idea in the context of traditional LANs. Schilit and Duchamp [Schi91] scrutinized using remote memory paging to allow diskless portable computers, and Iftode, Li, and Petersen [Ifto93] explored using memory servers in parallel supercomputers. Comer and Griffioen propose a communications protocol for remote paging in [Come92].

6. Conclusions

The advent of high-speed networks provides the opportunity for clients to work closely together to significantly improve the performance of file systems. We have investigated the technique of cooperative caching and conclude that cooperative caching can reduce read response times by nearly a factor of two for the workloads studied and that a relatively simple algorithm allows clients to efficiently manage their shared cache.

Acknowledgments

We owe special thanks to David Black of OSF for working as the OSDI shepherd for this paper. We would also like to thank Fred Douglis, John Howard, Edward Lee, John Ousterhout, and the anonymous OSDI referees whose comments improved both the content and the presentation of this paper. We are grateful to Mary Baker, John Hartman, Michael Kupfer, Ken Shirriff, and John Ousterhout for making the Sprite traces available. Finally, we thank Matt Blaze for providing the rpcspy tools we used to gather the Auspex traces.

References

[Arch86] James Archibald and Jean-Loup Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4:273–298, November 1986.

[Arpa94] Remzi Arpaci, Amin Vahdat, Thomas Anderson, and David Patterson. Combining Parallel and Sequential Workloads on a Network of Workstations. Technical report, Computer Science Division, University of California at Berkeley, 1994.

[Bake91] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a Distributed File System. In *Proc. of the 13th Symposium on Operating Systems Principles*, pages 198–212, October 1991.

[Blaz93] Matthew Addison Blaze. *Caching in Large-Scale Distributed File Systems*. PhD thesis, Princeton University, January 1993.

[Come92] Douglas Comer and James Griffioen. Efficient Order-Dependent Communication in a Distributed Virtual Memory Environment. In *Symp. on Experiences with Distributed and Multiprocessor Systems III*, pages 249–262, March 1992.

[Dahl94] Michael D. Dahlin, Clifford J. Mather, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. A Quantitative Analysis of Cache Policies for Scalable Network File Systems. In *Proc. of 1994 SIGMETRICS*, pages 150–160, May 1994.

[Doug91] Fred Douglis and John Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software: Practice and Experience*, 21(7), July 1991.

[Felt91] Edward W. Felten and John Zahorjan. Issues in the Implementation of a Remote Memory Paging System. Technical Report 91-03-09, Dept. of Computer Science, University of Washington, March 1991.

[Fran92] Michael J. Franklin, Michael J. Carey, and Miron Livny. Global Memory Management in Client-Server DBMS Architectures. In *Proc. of the International Conference on Very Large Data Bases*, pages 596–609, August 1992.

[Hage92] Erik Hagersten, Anders Landin, and Seif Haridi. DDM–A Cache-Only Memory Architecture. *IEEE Computer*, 25(9):45–54, September 1992.

[Henn90] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.

[Howa88] 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 on Computer Systems*, 6(1):51–81, February 1988.

[Ifto93] Liviu Iftode, Kai Li, and Karin Petersen. Memory Servers for Multicomputers. In *Proc. of COMPCON93*, pages 538–547, 1993.

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

[Leff93a] Avraham Leff, Joel L. Wolf, and Philip S. Yu. Replication Algorithms in a Remote Caching Architecture. *IEEE Trans. on Parallel and Distributed Systems*, 4(11):1185–1204, November 1993.

[Leff93b] Avraham Leff, Philip S. Yu, and Joel L. Wolf. Performance Issues in Object Replication for a Remote Caching Architecture. *Computer Systems Science and Engineering*, 8(1):40–51, January 1993.

[Leno90] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proc. of the 17th International Symposium on Computer Architecture*, pages 148–159, May 1990.

[Litz92] Michael Litzkow and Marvin Solomon. Supporting Checkpointing and Process Migration Outside the UNIX Kernel. In *Proc. of the Winter 1992 USENIX*, pages 283–290, January 1992.

[Mart94] Richard P. Martin. HPAM: An Active Message Layer for a Network of HP Workstations. In *Proc. 1994 Hot Interconnects*, August 1994.

[Munt92] D. Muntz and P. Honeyman. Multi-level Caching in Distributed File Systems or Your cache ain't nuthin' but trash. In *Proc. of the Winter 1992 USENIX*, pages 305–313, January 1992.

[Mutk91] Matthew M. Mutka and Miron Livny. The Available Capacity of a Privately Owned Workstation Environment. *Performance Evaluation*, 12(4):269–84, July 1991.

[Nels88] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[Nich87] David A. Nichols. Using Idle Workstations in a Shared Computing Environment. In *Proc. of the 9th Symposium on Operating Systems Principles*, pages 5–12, October 1987.

[Rost93] E. Rosti, E. Smirni, T. D. Wagner, A. W. Apon, and L.W. Dowdy. The KSR1: Experimentation and Modeling of Poststore. In *Proc. of 1993 ACM SIGMETRICS*, pages 74–85, June 1993.

[Ruem93] Chris Ruemmler and John Wilkes. UNIX Disk Access Patterns. In *Proc. of the Winter 1993 USENIX*, pages 405–420, January 1993.

[Sand85] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and Implementation of the Sun Network Filesystem. In *Proc. of the Summer 1985 USENIX*, pages 119–130, June 1985.

[Schi91] Bill N. Schilit and Dan Duchamp. Adaptive Remote Paging for Mobile Computers. Technical Report CUCS-004-91, Dept. of Computer Science, Columbia University, February 1991.

[Smit77] Alan Jay Smith. Two Methods for the Efficient Analysis of Memory Address Trace Data. *IEEE Transactions on Software Engineering*, SE-3(1):94–101, January 1977.

[Smit81] Alan Jay Smith. Long Term File Migration: Development and Evaluation of Algorithms. *Computer Architecture and Systems*, 24(8):521–532, August 1981.

[Thei89] Marvin M. Theimer and Keith A. Lantz. Finding Idle Machines in a Workstation-Based Distributed System. *IEEE Transactions on Software Engineering*, 15(11):1444–57, November 1989.

[vE92] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proc. of 1992 ASPLOS*, pages 256–266, May 1992.

[Wang93] Randolph Y. Wang and Thomas E. Anderson. xFS: A Wide Area Mass Storage File System. In *Fourth Workshop on Workstation Operating Systems*, pages 71–78, October 1993.

[Zhou93] Songnian Zhou, Jingwen Wang, Xiaohu Zheng, and Pierre Delisle. Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems. *Software - Practice and Experience*, 23(12):1305–1336, December 1993.