Cooperative Caching Middleware for Cluster-Based Servers*

Francisco Matias Cuenca-Acuna and Thu D. Nguyen
{mcuenca, tdnguyen}@cs.rutgers.edu

Department of Computer Science, Rutgers University
110 Frelinghuysen Rd, Piscataway, NJ 08854

Abstract

We consider the use of cooperative caching to manage the memories of cluster-based servers. Over the last several years, a number of researchers have proposed content-aware servers that implement locality-conscious request distribution to address this memory management problem [2, 18, 4, 5, 8]. During this development, it has become conventional wisdom that cooperative caching cannot match the performance of these servers [18]. Unfortunately, while content-aware servers provide very high performance, their request distribution algorithms are typically bound to specific applications. The advantage of building distributed servers on top of a block-based cooperative caching layer is the generality of such a layer; it can be used as a building block for diverse services, ranging from file systems to web servers.

In this paper, we reexamine the question of whether a server built on top of a generic block-based cooperative caching algorithm can perform competitively with content-aware servers. Specifically, we compare the performance of a cooperative caching-based web server against L2S, a highly optimized locality- and load-conscious server. Our results show that by modifying the replacement policy of traditional cooperative caching algorithms, we can achieve much of the performance provided by locality-conscious servers. Our modification increases network communication to reduce disk accesses, a reasonable trade-off considering the current trend of relative performance between LANs and disks.

1. Introduction

We consider the use of cooperative caching to manage the memories of cluster-based servers. Over the last several years, the number of Internet users has increased rapidly, necessitating the construction of *giant-scale* Internet servers [7]. To achieve the necessary scale and performance, service providers have no choice but to use large multiprocessor systems or clusters. While clusters promise better scalability, availability, and performance vs. cost [7], their distributed memory architecture often makes building scalable services more difficult. In particular, if the memories of individual nodes are used as independent caches of disk content, servers perform well only when their working sets fit into the memory of a single node, limiting system scalability [18, 5].

To address this problem, a number of researchers have proposed content-aware servers that implement locality- and load-conscious request distribution [2, 18, 4, 5, 8]; that is, these servers use information about the content being requested and the load at each node in the cluster to choose which node should serve a particular request. This allows the server to explicitly manage node memories as an aggregate whole rather than independent caches. In this paper, we propose a different approach: the use of a generic (but potentially configurable) cooperative caching middleware layer to manage the memories of cluster-based servers.

The advantage of using a generic middleware layer (or library) lies in its generality: it should be usable as a building block for diverse distributed services, reducing the effort necessary to design and implement cluster-based servers. On the other hand, the disadvantage is that its generality may hurt performance. For example, our middleware layer implements a block-based cooperative caching protocol to maximize generality. Handling blocks may be inefficient, however, for servers that always use entire files such as web servers.

While we believe that it is worthwhile to trade some performance for ease of design and implementation, the question remains: how much (if any) performance would we have to sacrifice? To explore this question, we compare the performance of a server that uses cooperative caching to that of a content-aware server. In particular, we simu-

*This work was performed using equipment purchased under NSF grant CISE-9986046.

late a web server built on top of a block-based cooperative caching layer to L2S, a highly optimized distributed server that implements both locality- and load-conscious request distribution [5].

Our specific contributions include: (1) working out the details necessary to adapt previous client-based cooperative caching algorithms to derive an algorithm for a distributed cluster-based server, (2) modifying the traditional LRU-based global replacement algorithm to improve the performance of cooperative caching in the context of cluster-based servers, and (3) comparing the performance of cooperative caching with that of a content-aware server for serving static web content.

Our simulation results for a set of 4 web traces show that servers employing traditional cooperative caching algorithms will likely perform significantly worse than locality-conscious servers. However, a small modification to the replacement algorithm to keep the last copy of a block in memory whenever possible leads to dramatic increase in throughput. In fact, our results show that a web server employing our modified cooperative caching algorithm can achieve over 80% of L2S's throughput in almost all cases and over 92% of L2S's throughput in most cases. Considered in the light of the fact that currently, the simulation of L2S includes two implementation advantages that may account for this performance difference, including the use of TCP hand-off [8] and the assumption that the entire web content is replicated on the disk of each node in the cluster, our results show that cooperative caching has the potential to fully match the performance of locality- and load-conscious request distribution. These results strongly support further exploration of the use of a middleware cooperative caching layer as a building block to ease the task of designing and implementing scalable cluster-based services.

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 describes our cooperative caching layer in more detail. Section 4 describes our simulation system. Section 5 presents simulation results and discusses our modification to traditional cooperative caching algorithms. Section 6 concludes the paper and discusses future work.

2. Related Work

Cooperative caching has been used to improve client access latency and reduce server load for some time [14, 11, 19]. The basic algorithm of our cooperative caching layer derives from this body of work. Our work differs from these efforts in that we concentrate on the aggregate performance of a distributed server whereas they mostly concentrate on improving the performance of individual client nodes and reducing server load. Further, system parameters for a network of clients differ significantly from those of a cluster-based server. These differences lead to different trade-offs in the caching algorithm.

Our work is also related to efforts to design and build cooperative caches for distributed file systems [6, 1, 10]. However, these efforts did not consider whether cooperative caching is competitive with content-aware request distribution. Further, we are interested in building a cooperative caching middleware layer that can be used by diverse distributed services, not just file systems.

In the context of Gigantic Internet servers [7], Fox et al. suggest using dedicated nodes for caching data in order to help I/O-bound or CPU-bound nodes [12]. In their work, they implement such a middleware layer and provide an API for programmers to use their caching services. Our work could be used in a similar manner. However, we are more interested in a layer that could be used as a library module as well as an independent middleware service of its own.

3. Overview of the Cooperative Caching Middleware (CCM) Layer

CCM is a block-based cooperative caching layer; it accepts requests for specific file blocks and replies with the appropriate data. Block requests can arrive at any node in the cluster but each file is stored on only one disk in the cluster. The node where a file is stored on disk is called its *home*. Currently, we assume a read-only request stream (and so have not implemented a write protocol) since we are studying CCM in the context of a web server serving only static content.

There are two types of blocks in CCM: *master* blocks and *non-master* blocks. When a file block is first read from disk to be cached in memory, it is designated a master copy. Copies of a master copy are called non-master copies. When a request for a block b in file f arrives at a node n, if n has a copy of b in its memory, then it services the request right away. Otherwise, n uses a system of *hints* to locate the master copy of b, b_{mc}. If b_{mc} is currently in the memory of some node m, then n requests a non-master copy of b from m. On receiving m's reply, n keeps the copy of b in its memory and services the request. If n cannot locate b_{mc} in cluster memory, then n requests a copy of b from f's *home* node. If the home node is able to locate b_{mc} in cluster memory, then it forwards a request to the caching node on n's behalf. If it also cannot locate b_{mc}, then it reads b into memory and sends it to n. This copy is now designated the master copy of b in memory. The home does not keep a copy of b; only n does.

In its most basic form, CCM employs an approximate global LRU replacement scheme. As shall be seen in Section 5, this replacement algorithm must be modified to achieve good performance. Each node tracks the age of

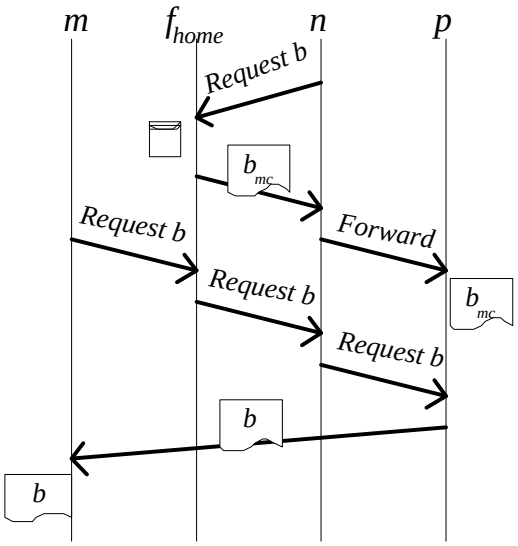


Figure 1. *Time lines for four cluster nodes, $m, f_{home}, n, and p$. Time progresses from top to bottom. b is a block in file f. f_{home} is f's home. The chain of events is as follows: (1) n requests a copy of b from f_{home}. (2) f_{home} reads it from disk and sends b_{mc} to n. (3) n evicts b_{mc} and forwards it to p. (4) m wants a copy of b, sending the request to f_{home} since it does not have a location hint for b. The request then follows the forwarding path of b_{mc} until it reaches p. (5) p sends a non-master copy of b to n*

the oldest block of each of its peers. When a node brings a block to its memory to service some request, if its memory is full, it evicts its oldest block. If its oldest block is a non-master copy or is the oldest block in the system, then it simply drops the block. If, however, the node's oldest block is a master block and one of its peer has an older block, then it forwards the evicted master block to the peer with the oldest block. When a node receives a forwarded block, it must drop its oldest block to make place for the forwarded block. Two important properties should be noted: (1) blocks forwarded to peers do not cause cascaded evictions, and (2) when a forwarded block arrives at its destination, all blocks at the destination may now be younger than the forwarded block; in this case, the forwarded block is dropped.

Given the above description of CCM's block-based cooperative caching algorithm, two questions remain: (1) how does a node n locate some block b that is not resident in n's memory but may be in the global cache somewhere? and (2) how does each node track the ages of the oldest blocks at its peers? CCM uses a hint system similar to that described in [19] to address both of these issues.

CCM implements two types of hints: location hints and age hints. Whenever the home node of a file f reads a master copy of a block b_{mc} and forwards it to a node n, it inserts a location hint saying that n is caching b_{mc} into a local pool of such hints. Whenever a node m needs to service a request for block b, it first looks in its local cache. If b is not there, it looks for a location hint for b in its local pool of hints. If there's a hint saying that b_{mc} is resident at node n, then m would request a non-master copy of b from n. If m does not have a location hint for b, it sends a request for b to f's home f_{home}. When the request arrives at f_{home}, if f_{home} contains a location hint for b_{mc}, then f_{home} would forward the request to the caching node, say n. If n still has b_{mc} in its memory, then it sends a non-master copy to m. When m receives b, it puts a hint saying that b_{mc} is at n into its local pool of hint. If n has forwarded b_{mc} to a peer p, it would have kept a hint saying that b_{mc} is now at p. It uses this hint to pass on m's request to p. If n previously evicted b_{mc} without forwarding it anywhere, n would now notify f_{home} that it no longer has b_{mc}, at which point f_{home} would read b_{mc} from disk, forwards it to m, and update its hint to say that m now has b_{mc}. Figure 1 illustrate this process for one example sequence of requests and evictions. Note that hints are updated lazily: a node caching b_{mc} does not tell f_{home} if it decides to evict or forward b_{mc} until it receives a request for b. In fact, the home is never notified of the forwarding of master copies; each request simply follows a chain of hints to the current node caching the master copy of a block.

Nodes use age hints to track the age of the oldest block at each of its peers. Age hints are exchanged as follows: when a node forwards a master block to a peer because it is evicting that block and, according to its age hints, the peer has an older block, it piggybacks the age of its oldest remaining block. When the peer receives the forwarding message, it replies with the age of its oldest block, taking into account the age of the block that it just received. As Sarkar and Hartman noted in [19], under this exchange protocol, a node that is evicting blocks frequently will be updating its hints often while a node that doesn't have much activity will tend to be a target for remote evictions and therefore it will be receiving up-to-date age hints as well. If a node does not have age hints for all peers, it chooses one of the peer for which it does not have an age hint and forwards the evicted master block there. In this way, eventually, each node will gather age hints for all peers.

Orthogonal to the issue of cache management is the issue of file distribution and location. Currently, we assume the general case of files being distributed across all nodes, with each node having a copy of the global file-to-node mapping. The actual file distribution and location scheme can be implemented in a variety of different ways, depending on where CCM is employed. For example, in Archipelago, Ji and Felten use a hash to map pathnames to cluster nodes [16] .

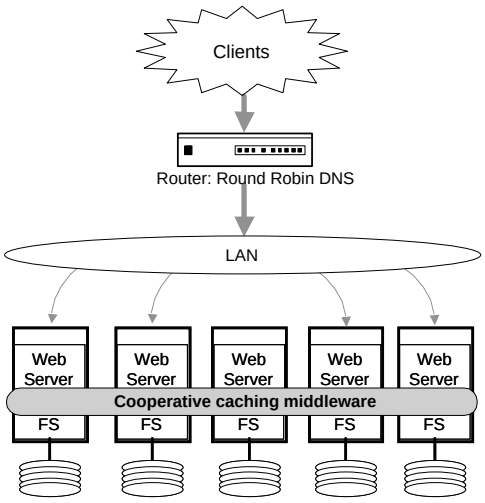


Figure 2. *Simulated architecture.*

4. Experimental Environment

4.1. L2S

We compare CCM's performance to L2S, a highly optimized content-aware server that uses locality- and load-conscious request distribution to provide good performance in a wide range of scenarios [5]. In particular, L2S tries to migrate all requests for a particular file to a single node so that only one copy of each file is kept in cluster memory. If a node becomes overloaded, however, L2S will replicate a subset of the files, sacrificing memory efficiency for load balancing.

L2S uses whole files as the caching granularity, employing a custom de-replication algorithm instead of block replacement. This algorithm behaves like local LRU, but incorporates load statistics and tries to keep at least one copy of each file in memory whenever possible.

In addition to differences in the memory management strategy, L2S also differs from CCM "implementation-wise" in two respects. First, L2S uses TCP hand-off to migrate client requests. Second, all simulations of L2S currently assume that each node contains the entire web content on its disk (although this assumption is not inherent to L2S). We discuss the potential implications of these differences in Section 5.

4.2. The Simulator

Our simulator derives from the one used to study L2S in [5]. It is event driven and models hardware components as service centers with finite queues. Using this framework, we model a distributed web server running on 4-8 cluster nodes connected by a high-performance LAN. Clients are connected via a router with round-robin DNS capabilities [17]. Figure 2 shows this setup. Note that, currently, we assume the same network is used to field/service client requests and for intra-cluster communication. This assumption does not affect our results significantly as the network is never the performance bottleneck.

At a more detailed level, each node is comprised of a CPU, memory, a NIC, and a disk, all connected by two buses, a system bus and an I/O bus. Each of these device is modeled as a service center with one or more finite queues. Each queue typically represents a particular subset of the possible events that use that resource. For example, arriving client requests are placed in one queue, peer block requests are placed in another queue, etc. These queues really model the capability of each resource to multi-task and are necessary to avoid simulation deadlocks. We model contention for the I/O bus explicitly but the effects of contention for the memory bus and all other costs of memory accesses are implicitly included in the CPU overheads for various processing events.

The modeling constants for all the major simulation components are shown in Table 1. The block-based operations are specific to CCM. The parsing and serving times represent the time to parse URL requests and the time to actually send content cached in local memory in reply to a request. Overall, our simulation parameters approximate a Cisco 7600 router [9], a VIA Gb/s LAN [13], an 800 MHz Pentium III CPU with 133 MHz system bus, and an IBM Deskstar 75GXP disk [15]; we derived these parameters (with help from Bianchini and Carrera [8]) using careful single-node measurements and some extrapolation. In order to model file system effects, we charge an extra seek for accessing the metadata on every 64KB access; we also assume that the file system provides a pre-allocation mechanism that ensures that files will be contiguous within 64KB blocks. The values presented in Table 1 for the modeled disk are probably on the conservative side. We chose these parameters as they are comparable to those used for L2S (and LARD [18]), making it easier to compare simulation results.

4.3. Web Traces

We use four web traces obtained from the University of Calgary, Clarknet (a commercial Internet provider), NASA's Kennedy Space Center, and Rutgers University to drive our simulator. Table 2 gives relevant details on these traces. The Calgary, Clarknet, and NASA traces were studied in detail in [3].

We use these four traces because they have relatively large working set sizes compared to other publicly available traces. For example, Figure 3 shows the cumulative distribution frequency and size for the Rutgers trace. Ob-

Events	Time (ms)
Request Processing	
Parsing time	0.10ms
Serving time	0.1 + (Size/11500)ms
Block Operations	
Process a file request	0.003 + (NBlocks*0.010)ms
Serve peer block request	0.007ms
Cache a new block	0.01ms
Process an evicted master block	0.16ms
Disk Operations	
Disk reading time (non-contiguous blocks)	18.8 + (Size/3000)ms
Disk reading time (contiguous blocks)	(Size/3000)ms
Bus & Network	
Bus transfer time	0.0001 + (Size/131072)ms
Network transfer time	0.003 + (Size/128000)ms
Network Latency	0.0038ms

Table 1. *Simulation parameters.*

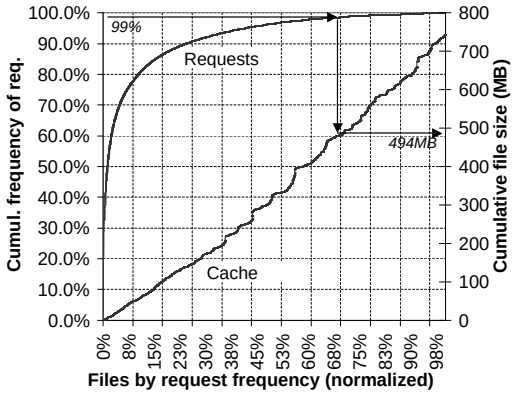


Figure 3. *Rutgers University trace. On the X axis is the file set of the trace sorted in decreasing order of request frequency. The left Y axis shows the normalized cumulative fraction of requests while the right Y axis shows the total data set size.*

serve that in order to cache 99% of the requests, 494MB of memory is needed.

Even though we chose the biggest traces available, their working sets are still small. Thus, we found it necessary to simulate small memories (4-512 MB per node) to reproduce situations in which the working set size is larger than the aggregated memory of the cluster.

To measure the maximum achievable throughput of the cluster, we ignore the timing information present in the traces. Each HTTP client generates a new request as soon as the previous one has been served. We also measure throughput only after the caches have been warmed up in order to reflect their steady-state performance.

5. Results

We have simulated CCM and L2S on clusters of 4 and 8 nodes. Figure 4 shows the throughput achieved by L2S and three variants of CCM when running on 8 nodes with varying amounts of memory; results for 4 nodes show the same trends and so are not shown here because of space constraints[1]. The algorithm described in Section 3 is labeled as CCM-Basic; the other variants are discussed below. In our discussion of these results, we will concentrate on the cases where the disk is still part of the performance bottleneck; that is, we will derive most of our observations from the portions of the throughput curves where performance is not yet saturated due to CPU overheads.

From the above results, we observe that CCM-Basic's performance lags that of L2S significantly. In many cases,

[1]The reader is referred to http://www.panic-lab.rutgers.edu/Research/ccm/ for the full set of simulation results.

Trace	Num. of files	Avg. file size	Num. of requests	Avg. request size	File set size
Calgary	8363	31.66KB	567823	13.67KB	258.57MB
Clarknet	28864	14.20KB	2978121	9.50KB	400.20MB
NASA	5486	41.70KB	3147685	20.33KB	223.41MB
Rutgers	25530	28.43KB	745815	17.54KB	708.93MB

Table 2. *Characteristics of the WWW traces used.*

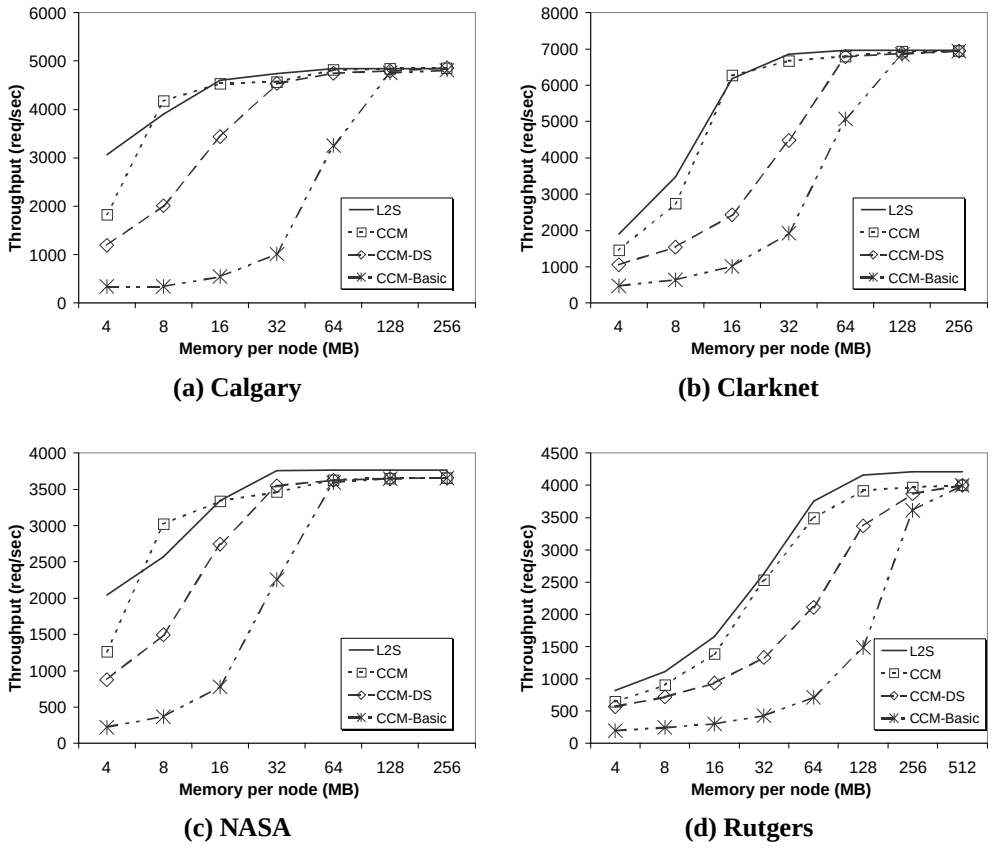
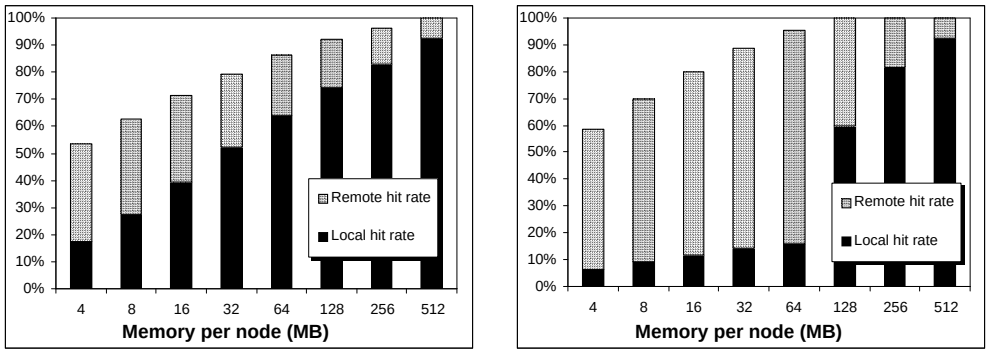


Figure 4. *Throughput for L2S and CCM when running on 8 nodes with varying amount of memory.*



(a) CCM-Basic **(b) CCM**

Figure 5. *Local and global block hit rates for the Rutgers trace on 8 nodes when using (a) CCM-Basic and (b) CCM.*

CCM-Basic only achieves about 20% of L2S's throughput. When we looked closely at the gathered statistics, the following reasons for CCM-Basic's poor performance became apparent:

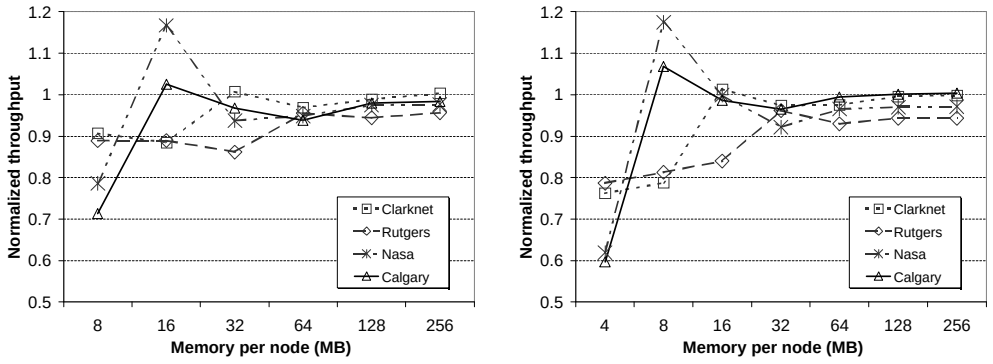
- One disk is always the performance bottleneck because of interleaving request streams. That is, suppose stream $s1$ is asking for blocks a, b, c that are within a contiguous 64KB unit on disk. If $s1$ is served uninterrupted, then only 2 seeks would be needed. If another stream, $s2$, shows up at the disk asking for blocks x, y, z that is in a different 64KB unit than a, b, c, then the two streams might interleave. This would result in a total of 12 seeks instead of 4 if the two streams were perfectly interleaved as a, x, b, y, c, z. Since nodes are often servicing multiple streams (local as well as remote), this interleaving is quite possible. In fact, in each simulation, the first disk that is "slowed down" because of interleaving falls behind and becomes a consistent source of interleaving for the remainder of the simulation. This disk becomes the performance bottleneck for the entire system. (Note that this is not a problem for L2S since L2S always request the entire 64KB chunk at a time from the disk.)

- CCM-Basic is similar to existing cooperative caching algorithms in that it gives a master block being evicted based on local LRU replacement another chance if it is not the globally oldest block (e.g., [11, 19]). Despite this favoring, master blocks are often discarded when multiple copies of other blocks exist in cluster memory—this is because hot files are often accessed on multiple nodes of the cluster because of the round-robin distribution of client requests. Thus, master copies of blocks of files that are accessed relatively infrequently are often flushed from the cache in favor of non-master copy of blocks from hot files. Unfortunately, while this replacement policy increases the local hit rate of each node, the percentage of time a node has a copy of a requested block in its memory, it can decrease the global hit rate, the percentage of time a node finds a master copy of the requested block in one of its peers' memories (when the block is not cached in its own memory).

We believe that the first problem arises from an inaccuracy in our simulator: a reasonable system would likely implement some form of request scheduling, caching, and/or prefetching and so the seek penalty would not be nearly as large. To correct this inaccuracy, we implemented a simple scheduling algorithm in our queue of disk requests; whenever a block is read from a 64KB chunk of a file, if there are requests for any other blocks within that chunk, they are served at the same time in order to avoid additional seeks. This leads to the CCM-DS performance curves, which are better than CCM-basic but still significantly worse than L2S.

This led us to examine the second problem, postulating that in a server environment, where network performance is increasing rapidly relative to disk performance, stronger preference for keeping master blocks in memory at the expense of traversing the network more often for blocks from hot files would lead to increased performance. There were many potential ways to modify the replacement algorithm. We chose a simple adaptation to explore the correctness of this theory: when eviction is necessary, never evict a master copy if the evicting node is still holding a non-master copy; instead, evict the oldest non-master copy. If the node is only holding master copies, then perform the global LRU eviction as before. This modification leads to the CCM performance curves.

Figure 5 shows the effect that this modification to the re-



(a) 4 nodes **(b) 8 nodes**

Figure 6. *CCM's throughput normalized against L2S.*

placement algorithm has on the local vs. global block hit rates. Observe that, indeed, CCM-Basic has higher local hit rates than CCM but that the global hit rate for CCM is higher than that of CCM-Basic. Further, the in-memory hit rate (local + global) is significantly higher for CCM, which is as expected since CCM was modified specifically to increase the size of the data set cached in memory.

Note that CCM's replacement algorithm is rather extreme; it leads to all memories holding only master copies, which does not necessary lead to best performance since nodes have to fetch data remotely to serve most requests (see Figure 5(b)), paying the attendant communication and processing costs. An algorithm that allows some discard of master blocks before all non-master blocks are gone may lead to better performance.

Despite CCM's limitation, however, it is a good starting point to evaluate whether cooperative caching can be more competitive with locality- and load-conscious request distribution. Clearly, it can! CCM is quite competitive with L2S, achieving over 80% of L2S's throughput in almost all cases, and achieving over 92% of L2S's throughput in most cases [2]. To show these differences more clearly, Figure 6 normalizes CCM's throughput against that of L2S. Except for systems with very small memory sizes (4MB per node and sometimes 8MB per node), CCM is very competitive with L2S. Moreover, L2S currently has two performance advantages over CCM: (1) L2S uses TCP hand-off, and (2) current simulations of L2S assume that files are replicated on all disks in the cluster. Bianchini and Carrera have shown that a server that doesn't make use of TCP hand-off can loose up to 7% throughput in one particular cluster environment [8]. Figure 7 shows an estimate of the performance advan-

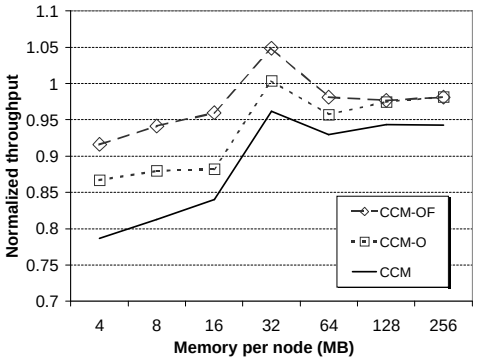


Figure 7. *Throughput of three variants of CCM normalized against that of L2S for the Rutgers trace running on 8 nodes. CCM-O is an optimistic implementation of CCM where hints are always correct at the time they are consulted. CCM-OF is CCM-O with files replicated on the disk of every node.*

[2] Curiously, CCM outperforms L2S for Calgary and NASA at 8MB per node. At these points, CCM outperforms L2S because it achieves a higher byte hit rate (91.4% vs. 87% for Calgary and 91.4% vs. 83.7% for NASA). We speculate that this is because CCM is able to maintain portions of files in memory whereas L2S always discard entire files.

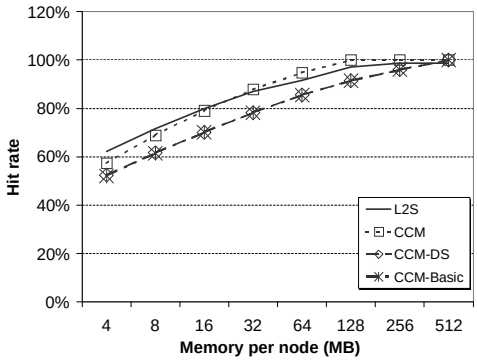


Figure 8. *L2S and CCM byte hit rates for the Rutgers trace running on 8 nodes.*

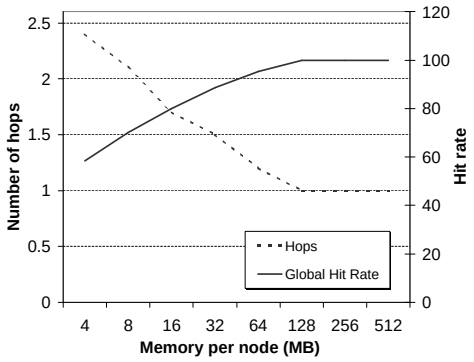


Figure 9. *Effectiveness of location hints for the Rutgers trace running on 8 nodes.*

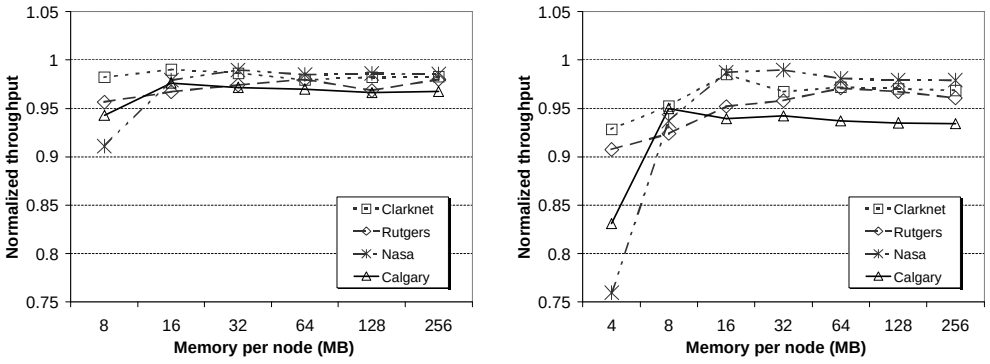
tage of full data replication on disks for the Rutgers trace when running on 8 nodes. In this figure, we plot normalized throughput for two additional variants of CCM, an optimistic version (CCM-O) where hints are always accurate at the time when they are used (although actions based on these hints may turn out to be incorrect because of changes in the system from when the hints were consulted to when a resulting action is completed), and the optimistic version with full replication (CCM-OF). CCM-OF gives an additional 3-4% performance improvement over CCM-O (over all traces). It is not clear that CCM can leverage full file replication on disk to improve performance since not going through the home to read master blocks from disk may lead to multiple master copies for a single block in memory—we are currently investigating how to leverage such replication in CCM. Further, the 7% performance advantage of TCP hand-off clearly does not translate literally to our experiments. These differences argue, however, that we may be able to further improve CCM performance so that there is little or no performance loss from that of L2S.

Figure 8 shows that CCM achieves much of L2S's performance because it makes efficient use of main memory, providing close to L2S's byte hit rates, albeit most are global hits. (As already noted, in some cases, CCM achieves better byte hit rates than L2S.) Further, CCM's hit rates come close to the theoretical maximum possible; for example, CCM's hit rate for the Rutgers trace is 96% with 64MB per node compared to the theoretical maximum of 99% for 512MB of total memory (see Figure 3).

Figure 9 shows the effectiveness of the location hint system, plotting the number of hops a request must travel on average before the master copy is found (or before the request is rerouted to the home to be serviced from disk). This figure only include requests that miss in the local cache of a node and so the minimum number of hops is 1. This data explains why CCM can perform poorly when system memory is severely limited; hints become stale rapidly, causing requests to inaccurately "chase" the caching of blocks through the system. Further, age hints also become stale rapidly, leading to incorrect eviction of master blocks. On extreme cases the error rate reaches 20% of incorrect evictions. blocks. Figure 10 shows the loss of throughput due to the overhead of maintaining hints and hint inaccuracies by comparing CCM's performance against that of the optimistic version of CCM (CCM-O). This Figure clearly shows that, in the case where CCM performs badly compared to L2S, it also performs badly compared to CCM-O and so the performance loss can be attributed to failures of the hint system.

Surprisingly, CCM's complete lack of load balancing does not hurt its performance compared to L2S. This is because the round-robin distribution of requests diffuse the hot files throughout the cluster. Thus, no single node is overwhelmed by peers' requests for copies of hot blocks.

Of course, this round-robin request distribution coupled with CCM's complete favoring of master blocks may lead to increased response time (because of the additional communication needed to follow hints and to obtain copies of needed blocks from peers). Figure 11 plots CCM's average request response times, normalized to those of L2S, against varying memory sizes. These results show that there is some degradation of response time—in most cases, less than 20%. At very small memory sizes (e.g., 4MB per node), CCM's response time can degrade more significantly (20-70%) because of the inaccuracy of the hints. In one case, the Clarknet trace, CCM achieves better response time than L2S when there is more than 8MB of memory per node because of its greater byte hit rates. At least a portion of these differences is likely due to artifacts of our simulation system; we had to model CCM at a more detailed level in the simulator (because of the many more possible events in a block-based system), creating greater potential for inter-



(a) 4 nodes **(b) 8 nodes**

Figure 10. *CCM's throughput normalized against the optimistic version (CCM-O).*

request interference. Thus, we do not believe that the difference in response time is significant, except in the cases of very small memory sizes.

To examine CCM's scalability vs. cluster size, we simulated the Rutgers trace on systems with up to 32 nodes, with 32MB of memory on each node. We only look at the Rutgers trace because the other traces' working sets are too small for such a scalability study. Figure 12(a) plots the resulting throughput, showing that CCM scales quite well up to 32 nodes.

Finally, we wanted to examine the effects of CCM's trade-off of increased communication for greater in-memory hit rates. Figure 12(b) plots the average disk, CPU, and network utilization of a cluster of 8 nodes running the Rutgers trace. Clearly, given the relative performance of Gb/s LANs and disks, this was the right trade-off. We were curious to see what would happen if we considered the use of only a 100Mb/s network. Figure 13 shows the resulting throughput and resource utilization for the Rutgers trace running on 8 nodes. Even with this slower network, communication is not the performance bottleneck (although it is a close second to the CPU at larger memory sizes).

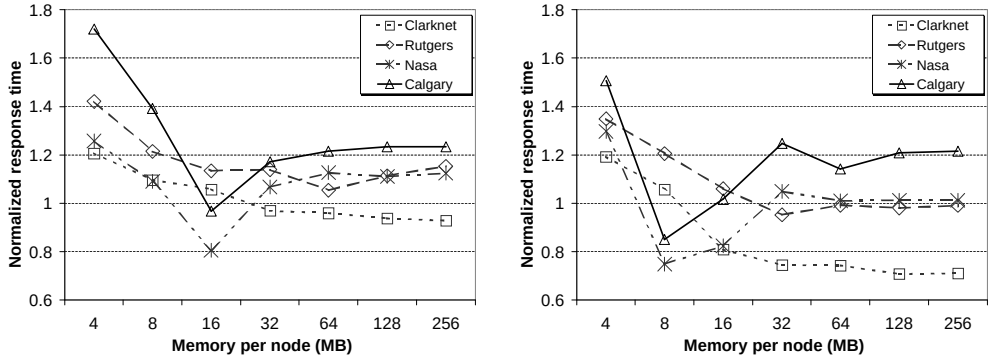
In summary, we have shown that a server employing cooperative caching has the potential to achieve much of the performance of locality- and load-conscious servers. Given the parameters of current clusters (e.g., Gb/s LANs), it is important to modify the cooperative caching algorithm from the ones proposed previously for client-side cooperative caching. In particular, our results point to the need for strongly avoiding the eviction of master copies, which leads to disk accesses. Our modified algorithm, CCM, achieves similar hit rates to a server implementing locality- and load-conscious request distribution, L2S, by ensuring that memory is first used to hold the working set of master copies before replicas are made. Of course, this trades increased network communication for higher in-memory hit rates. The low CPU overhead for network communication that comes with modern high-performance LANs is probably critical to the success of this trade-off.

6. Conclusion

We have studied the performance of cooperative caching in the context of scalable Internet servers; specifically, we have compared the performance of a web server built on top of a generic block-based cooperative caching layer to that of a server employing sophisticated locality- and load-conscious request distribution. Our results show that, when combined with an off-the-shelf web server and round-robin DNS, cooperative caching has the potential to achieve much of the performance of locality-conscious servers. This trade-off of a small (if any) amount of performance can be extremely beneficial as it means that designers of Internet services can use a generic cooperative caching layer as a building block instead of re-implementing service-specific request distribution and cache coherence algorithms.

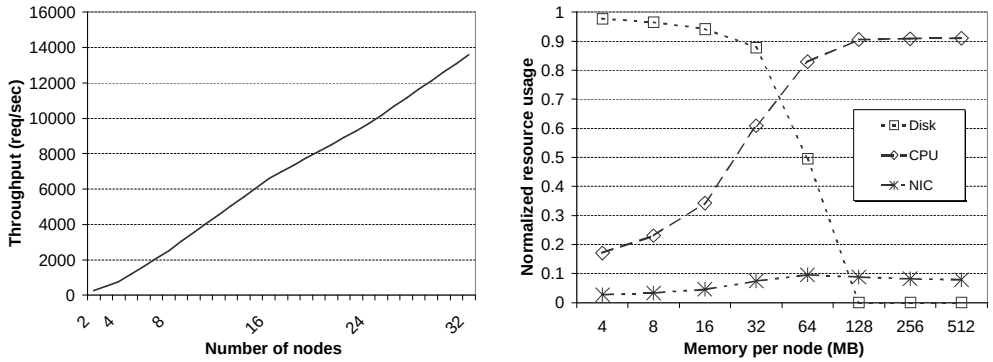
Beyond this paper, we plan to investigate how to support writes as well as reads in CCM. We will also investigate how to parameterize CCM so that it can be adapted to particular applications. For example, we will investigate whether CCM can easily be adapted for servers that always use whole files (e.g., a web server) and whether such an adaptation would improve performance. Finally, it would be interesting to consider how cooperative caching might be used together with content-aware request distribution in generic middleware layers.

Acknowledgement. We thank Liviu Iftode and the DISCO group at Rutgers University. This paper originated in a joint project with Liviu and the DISCO Laboratory. We also thank Ricardo Bianchini, Enrique V. Carrera, and Xiaoyan Li. Ricardo and Enrique provided us with the sim-



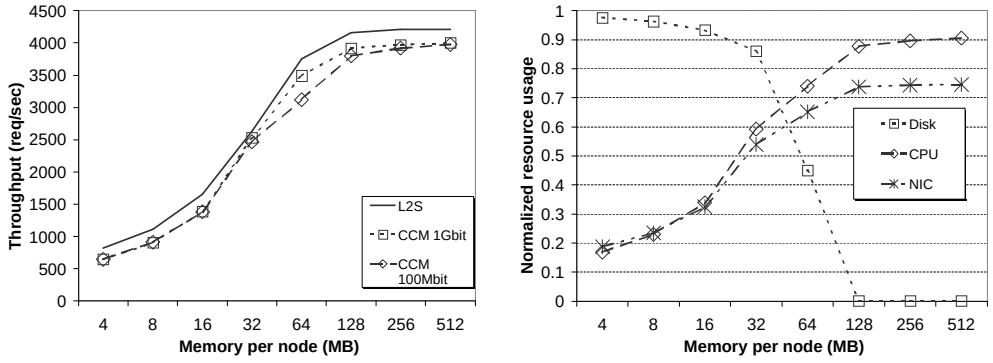
(a) Calgary: 4 nodes **(b) Rutgers: 8 nodes**

Figure 11. *CCM's average request response time normalized against L2S.*



(a) **(b)**

Figure 12. *(a) CCM's performance when running the Rutgers trace on varying cluster sizes, where each node has 32MB of memory. (b) CCM's resource utilization when running the Rutgers trace on 8 nodes.*



(a) **(b)**

Figure 13. *(a) CCM's throughput when running the Rutgers trace on 8 nodes with a 100Mb/s network vs. a 1Gb/s one. (b) CCM's resource utilization on a system with a 100Mb/s network.*

ulator for L2S and much help with our comparative study. Xiaoyan helped get some of the simulation results for LAN speed of 100 Mb/s.

References

[1] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless Network Files Systems. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Dec. 1995.

[2] D. Andresen, T. Yang, V. Holmedahl, and O. H. Ibarra. SWEB: Towards a Scalable World Wide Web Server on Multicomputers. Technical Report TRCS95-17, 2, 1995.

[3] M. Arlitt and C. L. Williamson. Web Server Workload Characterization: The Search for Invariants. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, May 1996.

[4] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel. Scalable Content-aware Request Distribution in Cluster-based Network Servers. In *Proceedings of the USENIX 2000 Annual Technical Conference*, June 2000.

[5] R. Bianchini and E. V. Carrera. Analytical and Experimental Evaluation of Cluster-Based Network Servers. *World Wide Web Journal*, 3(4), Dec. 2000.

[6] A. D. Birrell, A. Hisgen, C. Jerian, T. Mann, and G. Swart. The Echo Distributed File System. Report #111, DEC SRC, Palo Alto, CA, Sept. 1993.

[7] E. Brewer. Lessons from Giant-Scale Services—DRAFT.

[8] E. V. Carrera and R. Bianchini. Efficiency vs. Portability in Cluster-Based Network Servers. In *Proceedings of the 8th Symposium On Principles and Practice of Parallel Programming, ACM/SIGPLAN*, June 2001.

[9] Cisco 7600, http://www.cisco.com/, 2001.

[10] T. Cortes, S. Girona, and J. Labarta. PACA: A Cooperative File System Cache for Parallel Machines. In *Proceedings of the 2nd International Euro-Par Conference*, 1996.

[11] M. Dahlin, T. Anderson, D. Patterson, and R. Wang. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the First Symposium on Operating Systems Design and Implementation.*, Nov. 1994.

[12] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-Based Scalable Network Services. In *Proceedings of the 16th Symposium on Operating Systems Principles*, 1997.

[13] Gigabit Clan 1000, http://wwwip.emulex.com/ip/products/clan1000.html, 2001.

[14] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1), Feb. 1988.

[15] IBM Deskstar 75GXP, http://www.storage.ibm.com/, 2001.

[16] M. Ji, E. Felten, R. Wang, and J. Singh. Archipelago: An island-based file system for highly available and scalable internet services. In *Proceedings of 4th USENIX Windows Systems Symposium*, Aug 2000.

[17] E. D. Katz, M. Butler, and R. McGrath. A Scalable HTTP Server: The NCSA Prototype. *Computer Networks and ISDN Systems*, Nov 1994.

[18] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. M. Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1998.

[19] P. Sarkar and J. Hartman. Efficient Cooperative Caching Using Hints. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, Oct. 1996.