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 locality-conscious servers that implement content-aware request distribution to address this problem [2, 17, 4, 5, 6]. During this development, it has become conventional wisdom that cooperative caching cannot match the performance of these servers [17]. Unfortunately, while locality-conscious 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 locality-conscious servers. Specifically, we compare the performance of a cooperative caching-based web server against L2S, a highly optimized locality-conscious server. Our results show that by modifying the replacement algorithm 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 [8]. 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 and performance vs. cost [8], 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 [17, 5].

To address this problem, a number of researchers have proposed locality-conscious servers that implement content-aware request distribution [2, 17, 4, 5, 6]; 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 locality-conscious server. In particular, we simulate a web server built on top of an optimistic block-based cooperative caching layer to L2S, a highly optimized locality-aware server that implements both content- and load-aware request distribution [5].

Our 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 90% or matching L2S's throughput in most cases.

These results strongly support the use of a middleware cooperative caching layer as a building block to ease the task of designing and implementing scalable cluster-based services while providing much of the performance achievable via content-aware request distribution.

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 discusses future work and Section 7 concludes the paper.

2 Related Work

Cooperative caching has been used to improve client latency and reduce server load for some time [14, 11, 18]. 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.

Work in cooperative caching for distributed file systems is more closely related to our work here [7, 1, 10]. However, these efforts were specific to file systems and did not consider whether cooperative caching is competitive with content-based request distribution. We are interested in building a cooperative caching middleware layer that can be used by diverse distributed services, not just file systems. Further, we compare the performance of cooperative caching to content-based request distribution.

In the context of Gigantic Internet servers [8], 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. Also, we are concerned with whether cooperative caching is competitive with content-based request distribution.

3 Overview of the Cooperative Caching Middleware Layer (CCM)

CCM is a block-based cooperative caching system–since we are studying CCM in the context of a web server, we assume a read-only request stream (and so do not deal with a write protocol). CCM implements the following basic algorithm. When a file block is read from disk, it is designated as the *master* copy. The location of each master copy is maintained in a global directory. When a request for a block b 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 the global directory 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 from m. On receiving m's reply, n keeps the copy of b in its memory and services the request. If b_{mc} is not in memory anywhere, then n requests a master copy of b from the file's *home* node. (See below for a description of the file to home mapping.) The home node reads b from its disk and forwards the master copy to n. The global directory is updated to record the fact that n now has b_{mc}.

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 always knows the age of the

oldest blocks 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 oldest block is a master block and one of its peer has an older block, then it forwards the evicted 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. Also, the global directory is updated to contain the new location of the forwarded master 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.

Currently, we make several optimistic assumptions in our simulation of CCM. We assume that: *(i)* CCM has a perfect global directory of master blocks, *(ii)* it costs nothing to maintain this global directory, and *(iii)* CCM has perfect global knowledge of the age of the oldest block on each node. While these assumptions probably mean that our current results are upper bounds, we believe a practical cooperative caching layer can achieve much of this performance. First, as shall be seen in Section 5, the replacement policy of our current best-performing algorithm can likely be improved. Second, our optimistic assumptions are quite limited. For example, we only assume instantaneous global knowledge of the master block directory. That is, suppose that a request for block b arrives at node n. At that instant, b_{cm}, the master copy of b, is cached at node m. Using the global directory, n will request a copy of b from m. During the time that the request for b travels from n to m, however, m may discard b, resulting in an eventual disk read. Finally, Sarkar and Hartman have shown that it is possible to achieve very high location accuracy for master blocks (on the order of 98%) using a hint-based directory; exchanging hints only imposed an overhead of 0.04% on a 100 Mb/s LAN as hints are mostly exchanged as piggybacked information on required messages [18].

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. A node holding some file f on its disk is called f's *home*.

4 Experimental Environment

4.1 L2S

We compare CCM's performance to L2S, a highly optimized locality-conscious server that uses content- and load-aware distribution to provide very high 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.

L2S currently differs from CCM in that it assumes files are replicated everywhere. We are in the process of modifying L2S to have the same file distribution as CCM to remove this difference but believe that it will not affect performance significantly.

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 high-performance LAN, a router, and 4-8 cluster nodes. Each node is comprised of a CPU, NIC, and disk, all connected by a bus.

In our simulation, client requests are distributed among the cluster's nodes using a round robin DNS scheme [16]. New requests are routed in accordance with the Cisco 7600 performance specifications [9]. Currently, we assume the same network is used to field/service client requests and for intra-cluster communication.

The modeling constants for all the major simulation components are shown in Table 1. The block-based operations

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 and 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 Latency	0.0038ms

Table 1: *Simulation parameters.*

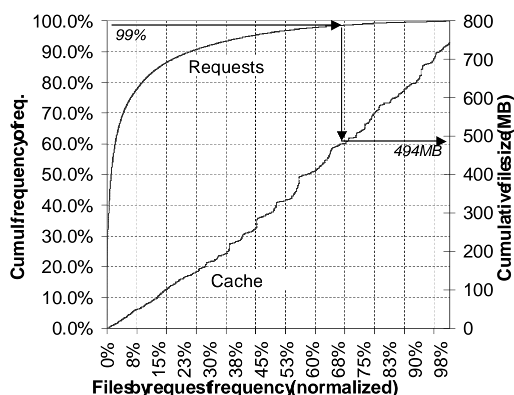


Figure 1: *Rutgers University trace. On the X axis is the file set of the trace sorted in decreasing order of request frecuency. The left Y axis shows the normalized cumulative fraction of requests while the right Y axis shows the total data set size.*

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 simulated parameters approximate a VIA Gb/s LAN [13], 800 MHz Pentium III CPU with 133 MHz main memory, and an IBM Deskstar 75GXP disk [15]; we derived these parameters using careful single-node measurements and some extrapolation. In order to model file system effects, we charge an extra seek for getting 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), 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 1 shows the cumulative distribution frequency and size for the Rutgers trace. Observe that

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.*

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

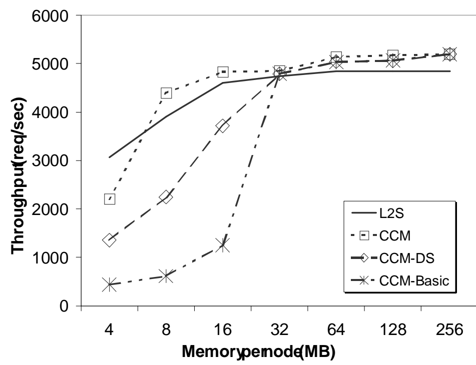
We have simulated CCM and L2S on clusters of 4 and 8 nodes. Figure 2 shows the throughput achieved by L2S and three variants of CCM when running on 8 nodes with varying amounts of memory; results for 4 nodes show the same trends and so are not shown here because of space constraints. The algorithm described in Section 3 is labelled as CCM-Basic; the other variants are discussed below.

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

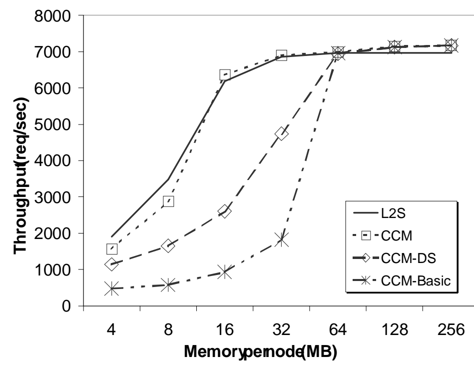
- One disk is always the performance bottleneck because of interleaving of request streams. That is, suppose stream $s1$ is asking for blocks a, b, c that are within a contiguous 64KB unit on disk. If $s1$ is served uninterrupted, then only 2 seeks would be needed. If another stream, $s2$, shows up at 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.
- CCM-Basic is similar to existing cooperative caching algorithms in that it gives a master block being evicted based on local LRU replacement a "second chance" if it is not the globally oldest block (e.g., [11, 18]). Despite this favoring, master blocks are often discarded when multiple copies of other blocks exist in cluster memory– this is because a number of files are accessed infrequently enough that their content almost always become the oldest before being accessed again. Unfortunately, while this increases the local hit rate–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.

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. This leads to the CCM-DS performance curves, which are better than CCM-basic but still significantly worse than L2S.

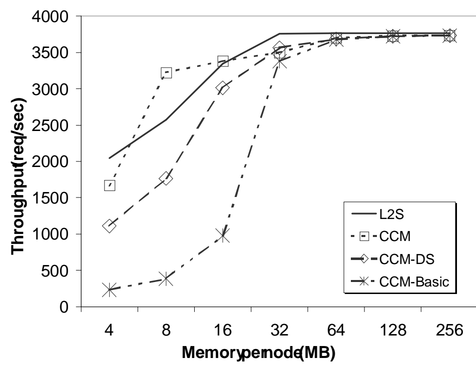
This led us to address the second problem, postulating that in a server environment, where network performance is increasing rapidly relative to disk performance, stronger avoidance of the eviction of master blocks would lead to increased performance. There were many potential ways to modify the replacement algorithm. We chose a very



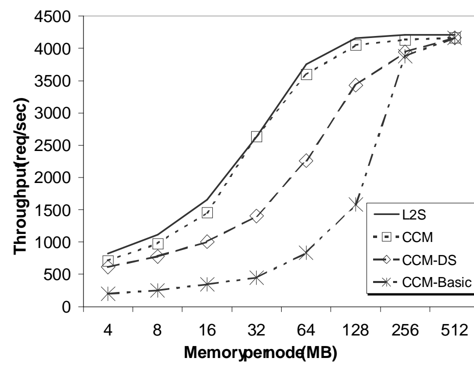
(a) Calgary



(b) Clarknet

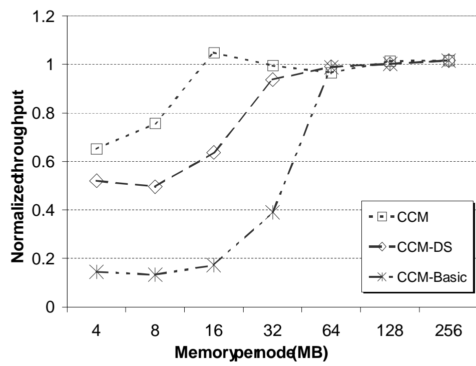


(c) NASA

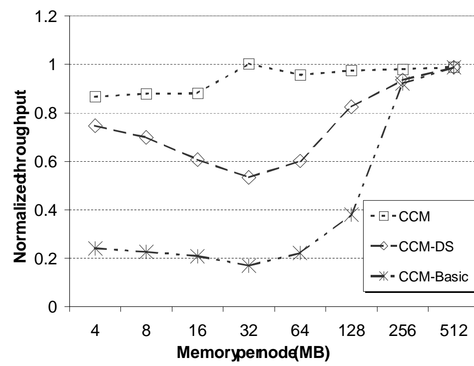


(d) Rutgers

Figure 2: *Throughput for L2S and CCM when running on 8 nodes.*



(a) Calgary: 4 nodes



(b) Rutgers: 8 nodes

Figure 3: *CCM's throughput normalized against L2S.*

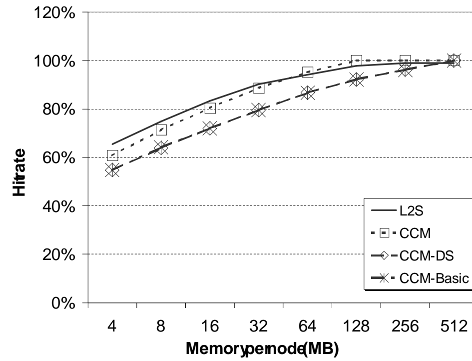
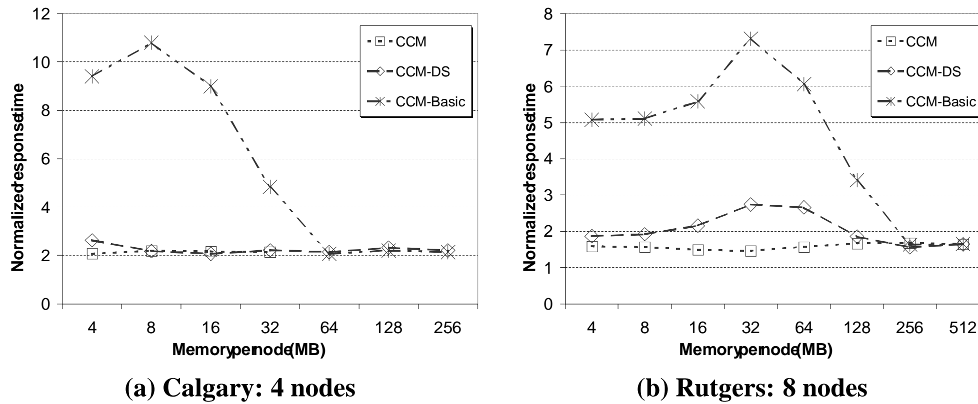


Figure 4: *L2S and CCM hit rates (Rutgers 8 nodes).*



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

Figure 5: *CCM's average request response time normalized against L2S.*

simple adaptation: 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.

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. For example, for the Rutgers trace running on 8 nodes with up to 64MB per node, CCM's local hit rate ranges from 12-21% while the remote hit rate ranges from 60-75%. This means that nodes have to fetch data remotely to serve most requests, paying the attendant communication and processing costs.

Despite CCM's limitation, it is a good starting point to evaluate whether cooperative caching can be more competitive with content-aware 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 90% or matching L2S's throughput in most cases. Figure 3 shows two representative graphs for CCM's throughput when normalized against that of L2S to illustrate this observation more clearly.

Figure 4 shows that CCM achieves much of L2S's performance because it makes efficient use of main memory, providing close to L2S's hit rates (albeit most are remote hits). 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 1).

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. It would be interesting to observe CCM's performance under a forced concentration of hot files on a single node (or subset of nodes).

Unfortunately, while CCM comes close to matching the throughput of L2S, its average response time can be 50-100%

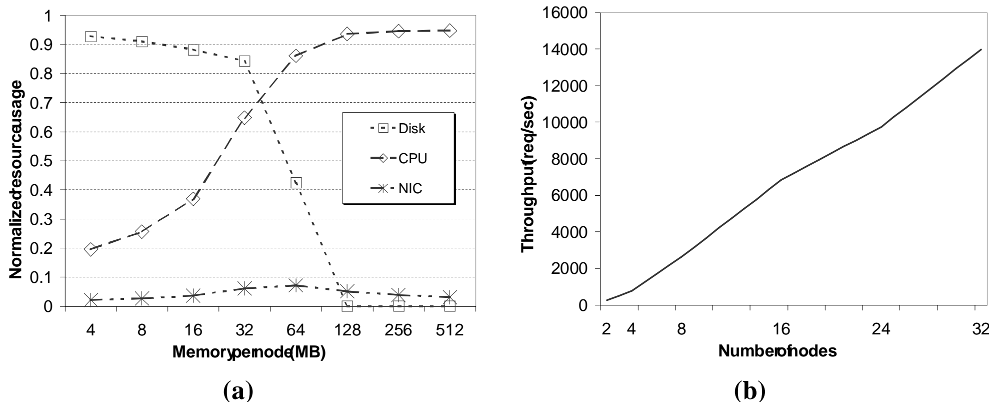


Figure 6: *(a) CCM's resource utilization when serving the Rutgers trace on 8 nodes. (b) CCM's performance when serving the Rutgers trace on varying cluster sizes; each node has 32MB of memory.*

worse than L2S. Figure 5 shows two representative graphs for CCM's average response time when normalized against that of L2S. With respect to wall clock time, CCM's average response time is very acceptable for a web server, being on order of 200-300ms. Still, we are currently investigating whether this degradation is inherent to CCM or an artifact of our simulation system–we model CCM at a finer granularity than L2S because of the necessity of simulating block operations. This leads to many more queues at the CPU, possibly leading to higher contention in our simulation system.

Note that one definite source of response time degradation is the extra network activity that must go on in CCM. However, Figure 6(a), which plots CCM's average resource utilization against the amount of memory on each node (for Rutgers running on 8 nodes), shows that the network is mostly idle. Thus, the added latency should not be much beyond one round trip of 8-10us and so cannot account for the added latency being observed.

Finally, Figure 6(b) plots CCM's performance against cluster size, showing that CCM scales quite well up to 32 nodes.

In summary, we have shown that a server employing cooperative caching has the potential to achieve much of the performance of locality-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-aware request distribution, L2S, by ensuring that memory is first used to hold the working set of master copies before replicas are made.

6 Future Work

Within the context of this paper, we are currently implementing a variant of the hint-based cooperative caching algorithm [18] in our simulator. This should remove any advantage that CCM derives from our current optimistic assumptions. We are also in the process of isolating the causes of the remaining performance differences between L2S and CCM, which differ in several dimensions. For example, L2S assumes TCP-hand-off. Bianchini and Carrera have shown that this can provide a performance advantage of approximately 7% over a server that does not use TCP-hand-off.

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, this paper assumes a very specific set of hardware characteristics. We will investigate the effects of different hardware configurations on the cooperative caching algorithm. Eventually, this work should lead to an implementation, where issues with interfacing to file systems and file buffer caches would be

investigated.

7 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 content- and load-aware 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 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.

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.

[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] R. Bianchini and E. V. Carrera. Efficiency vs. Portability in Cluster-Based Network Servers. Technical Report DCS-TR-427, Rutgers University, Nov. 2000.

[7] 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. ftp://gatekeeper.dec.com/pub/DEC/SRC/research-reports.

[8] E. Brewer. Lessons from Giant-Scale Services.

[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*.

[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.*, pages 267–280, 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*.

[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, 1988.

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

[16] E. D. Katz, M. Butler, and R. McGrath. A Scalable HTTP Server: The NCSA Prototype. *Computer Networks and ISDN Systems*, 27(2).

[17] 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*.

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