Request Distribution-Aware Caching in Cluster-Based Web Servers

Vlad Olaru Walter F. Tichy

Computer Science Department
Karlsruhe University , Germany
E-mail: {olaru,tichy}@ipd.uka.de

Abstract

This paper presents a performance analysis of request distribution-aware caching in cluster-based Web servers. We use the Zipf-like request distribution curve to guide static Web document caching. A combination of cooperative caching and exclusive caching provides for a cluster-wide caching system that avoids document replication accross the cluster. We explore the benefits of cooperative caching algorithms that use request distribution information to steer their behavior over general purpose cooperative caching algorithms. Exclusive caching exercises a fine-grained control over replication of data blocks across the cluster. The performance of the system has been assessed by using the WebStone benchmark. Our cluster-based server employs Linux kernel-level implementations of cooperative caching and exclusive caching. Current results show that request distribution-aware caching outperforms general-purpose caching algorithms, makes up for the performance loss of non-replicated data solutions and compares favorably to fully-replicated solutions.

1 Introduction

In this paper, we address two issues regarding the disk I/O subsystem of cluster-based servers. First, locality-aware request dispatching algorithms assume most of the time that documents are fully-replicated across the cluster. One alternative is to use virtual disks but then uncached concurrent non-overlapping block requests have to be serialized at the disk controller. In a system using replicated documents, such requests take advantage of the disk controller level parallelism. We assess the benefits of cooperative caching as a driving engine to bridge the gap between the performance of virtual disks vs. replicated documents. Second, we try to exploit information from the Zipf-like [20] Web document request distribution curve in order to improve the caching performance over general purpose algorithms. To be more specific, we show that request distribution aware cooperative caching outperforms its corresponding general purpose counterpart. Especially the class of large, unpopular files benefits of this approach. Moreover, handling block eviction in a cooperative cache pays off only for significantly large, unpopular documents. To the best of our knowledge, this is the first attempt to analyze the impact of application-aware (request distribution aware, in fact) cooperative caching.

Exclusive caching [19, 5] exercises a fine-grained control over multiple copies of the same data found in independently managed caches in the system (filesystem cache, onchip disk cache, etc). We extended the concept cluster-wide [13] by offering control over which copies of a disk block are kept around in response to a remote request and we investigate to which extent exclusive caching can help cooperative caching by avoiding unnecessary copies of certain classes of documents. To the best of our knowledge, this is the first attempt to combine exclusive caching with cooperative caching in a cluster-wide, application-aware caching system.

2 Server architecture overview

Our cooperative caching system is based on Cluster-Aware Remote Disks [12], which are kernel device drivers that represent local stand-ins for remote disk drives and fetch remote blocks over a System Area Network (SAN). Their strategy routine is driven by cooperative caching policies (algorithms) that allow checking block requests missing in the local cache against remote caches as well. The access mechanism of CARD drivers implements exclusive caching capabilities. In order to establish a global cache, several nodes in the cluster mount locally a remote disk via CARD drivers. If no cooperative caching policy is in use, the CARD acts as a virtual disk. Otherwise, the CARDs cooperate through the policy in order to jointly manage their corresponding parts of the local caches.

From a software perspective, a caching subsystem (as any other kernel subsystem, in fact) faces a serious problem: the more general it is, the little the chances to properly suit the needs of particular problems. Therefore, the design of our server is policy-oriented, rather then service-oriented.

That means applications may download in the kernel their own policies at will, in the spirit of extensible/grafting kernels [3, 9]. For instance, cooperative caching policies can be downloaded in the CARD driver to steer its operation. We will use this facility to experiment with several policies as explained in the following sections.

3 Caching on a curve

Web workloads exhibit a certain request distribution, amenable to a Zipf-like [20] formula. Of particular interest is the so called "heavy tail" of the distribution curve, consisting of unpopular static documents that account for a significant part of the servicing time. Moreover, it is possible to identify entire classes of such static documents according to their popularity. For instance, WebStone [17], a commercial benchmark for Web servers, identifies by default four classes of static documents: files smaller than 1KB, files in the (1KB, 10KB) and (10KB, 100KB) ranges and files larger than 100KB, denominated by class0, class1, class2 and class3 respectively. In terms of their popularity, class0 accounts for 50% of the requests, class1 for 35%, class2 for 14% and class3 for 1%. However, the servicing time for class3 accounts for roughly one quarter of the total time while servicing class3 comes close to 40% of the total time.

A legitimate question is to ask whether special treatment for each class could improve the overall server performance. Some results from connection scheduling in standalone servers [7, 10] are an encouraging starting point as favoring short-lived connections in a SRPT (Shortest Remaining Processing Time) connection scheduling improves significantly the response time without affecting the overall behavior of the server. Beforehand knowledge of the life length of a connection is shown to vary consistently with the size of the documents. Otherwise put, if one singles out large files as a separate class and schedules connections according to this simple classification, the overall performance of the standalone server improves.

By further refining the document taxonomy, we are using a combination of cooperative caching and exclusive caching to manage a the global cluster-wide cache according to the characteristics of the workload. Such an approach needs to respond several challenges. First off, it must be assessed whether cooperative caching compares favorably to simple solutions like document replication. Second, it is unclear whether general-purpose cooperative caching algorithms wouldn't do equally well. Finally, since a previous attempt to use general-purpose cooperative caching for distributed Web servers [14] relied on simulation, it is interesting to validate our solution through a real implementation.

3.1 Cooperative caching

Two of the main features of cooperative caching are of particular interest to our work. First, client disk block requests missing in the local cache are checked against remote client caches as well before going to the server. Second, cooperative caching implements a global replacement policy for locally evicted blocks. Due to the aforementioned flexibility of our CARD drivers, we can implement various lookup and eviction handling procedures. That allows us to test common sense intuitions about caching documents in the global cache. For instance, globally caching highly popular documents seems a good idea because it may yield high hit ratios. Also, saving evicted blocks for unpopular documents seems equally important because they account for a significant part of the total servicing time. But since all the classes compete for the same memory, would a general purpose cooperative caching algorithm do? Would a policy trading off among classes of documents do better?

Research showed that servers operating in complex environments using proxies, content delivery networks, etc. serve mostly the unpopular documents as the popular ones are filtered out of the request stream at early stages by the proxies, CDNs, etc. [15]. This is yet another argument to assess the performance of caching exclusively the class of unpopular documents.

In our previous work on cooperative caching [12], we showed that handling block eviction irrespective of the loads of the clients participating in the global cache may cause severe performance loss (the performance can plummet below that of a cold global cache). However, these results are valid for cooperative caching in a distributed enviroment. A natural question asks whether global eviction handling is a right idea in a cluster-based Web server, since such a server can be considered a parallel machine in which all nodes are (more or less) equally busy (especially when load balancing mechanisms are used). That is to say, the chances for finding lightly loaded nodes to host remotely evicted blocks are small and our previous experience argues against global eviction handling in such circumstances.

One final question asks whether cooperative caching can compare to a simple technique like replication. If not, one loses an important advantage of replication, namely that of the disk controller parallelism. In the worst case when all the requests have to go to the disk, cooperative caching pays double: the protocol overhead and the sequential disk processing, the latter being probably the heaviest price. And if this is the case, is it possible to alleviate the consequences by using a mixed approach, that is replicating some classes of documents while cooperatively caching the other?

We responded all these questions by writing specific algorithms that all build upon the HSCC algorithm [12], a general purpose cooperative caching algorithm. By specific algorithms we mean actually various versions of the lookup and eviction procedures of HSCC. For instance, deciding not to cooperatively cache a certain class of static documents is simply a matter of downloading in the CARD driver a lookup procedure that lets requests addressed to files from that class to be directly forwarded to the disk node. Similarly, for handling evictions of blocks belonging to files in a certain class of documents. For the case when no eviction was handled, a null eviction procedure has been downloaded in the kernel to disregard block eviction events.

3.2 Exclusive caching

Our exclusive caching solution [13] operates cluster-wide by avoiding double buffering as a host mouting a CARD requests a block from the remote disk. To avoid leaving a copy at the remote site we use a technique that allows a DMA transfer from the disk driver into the response socket buffer. When used with cooperative caching system, exclusive caching concerns only requests that need to go to the disk. Requests serviced from remote client caches don't need to worry about remote copies, because they got there according to the joint management algorithm of the global cache (loaded on demand or saved as a result of remote eviction) while eviction from that cache is regulated by the global replacement policy.

Exclusive caching allows such computer-attached disks in COTS clusters to exhibit also network-attached behavior, provided that no other useful computation takes place on that node. A disk with network-attached behavior mounted remotely through CARD drivers opens also possibilities for selective caching according to certain criteria. For instance, in terms of Web workloads, the page/buffer cache at the disk node can be used to store unpopular documents and thus act as a discard cache. This is simply done by suppressing the exclusive caching bit in the requests for uncached blocks belonging to files of that class. As a result, the disk node cache and its *local* replacement policy govern the caching of that class of documents which may make little sense to be cached across the cluster. Naturally, the effectiveness of such a method depends on the memory size of the disk node and the size of the class.

4 Performance evaluation

The performance evaluation of our caching system used WebStone [17], a well known commercial benchmark for Web servers respecting a Zipf-like [20] document retrieval distribution. The WebStone software has been configured to retrieve static documents only, to use HTTP 1.0 and a workload around 1 GB of data (corresponding to 300 simultaneous connections maintained by the client with the distributed server). The servers used a Linux *ext2* filesystems both on top of local disks and remotely mounted disks. The remotely mounted disks used CARD drivers with exclusive caching turned on by default (if not otherwise stated).

4.1 Experimental setup and methodology

We ran experiments on a 3-node Linux cluster interconnected through a Myrinet switch and LANai 7 cards (133 MHz processor on board, 2 Gb/sec in each direction). The host interface is a 64 bit/66 MHz PCI sustaining a 500 MB/sec throughput. The Myrinet cards are controlled by the GM 1.6.4 driver of Myricom [18]. The PCs are 350 MHz Pentium II machines with 256 MB of RAM. All the systems run Linux 2.2.14. The test disks are IBM DCAS-34330W Fast/Ultra-SE SCSI wth non-aged filessytems on them (0.7% non-contiguous, as reported by the *fsck* command). Our Web server choice was Apache 1.3.20 [2]. A Linux router stays between the client and the server machines. Both the client and the router are Athlon AMD XP 1.5 GHz PCs with 512 MB of RAM and run Linux 2.4.18. The client, the router and the server(s) are all interconnected through regular 100Mb/s Ethernet. Figure 1 describes visually the experimental setup.

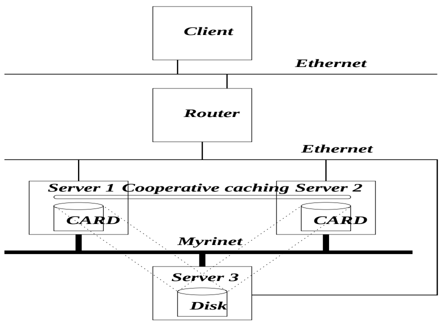

Figure 1. Experimental setup

WebStone runs on the client machine and sends requests to the cluster-based server through the router that dispatches them according to a round robin policy to two of the server machines in order to yield perfect load balancing (in terms of the number of requests serviced). The two cluster nodes build a cooperative cache of at most 512 MB (somehwat smaller, in fact, due to the space occupied mainly by the operating system and the server program). This amounts to at most half of the aggregated storage size of the working set of the workload (1GB of data). Thus, we avoid having the working set fit entirely either in any of the local memories or in the global cache. The two nodes mount the disk containing the benchmark file set through CARD drivers. The disk itself resides on the third server node and emulates network-attached behavior. The page/buffer cache at the disk node (the third server machine) is referred throughout the rest of the section as the "discard cache".

4.2 The performance of CARDs as simple remote disk interfaces

In order to have an idea about the performance loss of a server using a single disk remotely mounted when compared to a fully-replicated solution (that is, each server has all the needed documents stored on local disks), we drove a first experiment by running the benchmark with two servers in three setups. In the first setup, the two machines serve the requests from the local disks (same file set, replicated on both disks). In a second scenario, one of the two servers uses a CARD driver to mount the remote disk locally while the second server relies on a local disk to deliver the data. This case corresponds to the computer-attached operation of the disk. In the third setup, two machines mount locally the remote disk as a network-attached disk. The results are reported in Figure 2.

The two servers equipped with local disks perform best. The load is almost equally split between the two disk drives and that maximizes the amount of disk parallelism. The other two cases show that the task is disk and not computational dominated. For the second scenario, we assume that the additional load placed by the server software on the node hosting the disk is responsible for the light performance degradation.

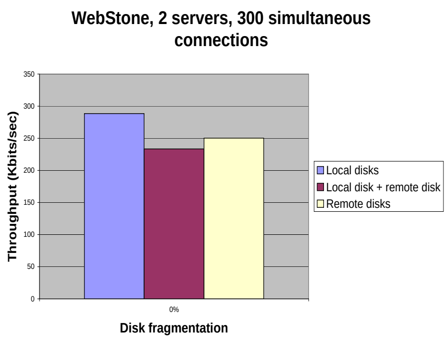

Figure 2. Remote disk vs. replication

4.3 The performance of cooperative caching without eviction handling

One of the questions we would like to ask is whether checking remote client memories for cached copies of a locally missing requested block is a technique that can fill out the performance gap that we saw in our previous experiment. In order to assess that, we wrote three policies and we compared their performance to that of the fully-replicated solution. Using the class definitions in Section 3, a brief description of these policies follows:

- cooperatively cache classes 2 and 3 of files, that is the files that require most of the servicing time. The small and popular documents are cached at the discard cache
- cooperatively cache classes 0 and 1 (small and popular documents) and rely on the discard cache to cope with classes 2 and 3
- cooperatively cache all the requested documents

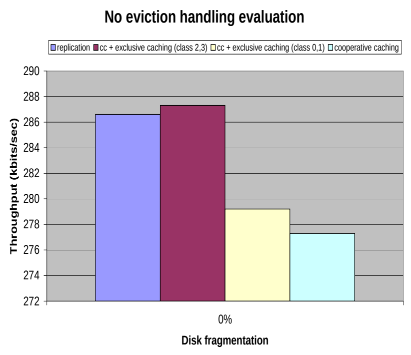

Figure 3. Evaluation of cooperative caching without eviction handling

The results are presented in Figure 3. Notice that the first policy has the best results (even slightly better than the replicated solution), which stresses on the importance of caching the heavy tail of the request distribution curve. The last two policies exhibit comparable performance (with a slight degradation for the results of plain cooperative caching). This outcome can be explained if we remember that classes 0 and 1 account for 65% of the requests. Thus, using the limited capacity of the discard cache to store the big files of classes 2 and 3 doesn't improve significantly the performance.

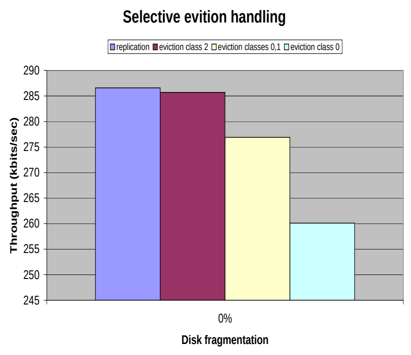

Figure 4. Selective block eviction handling according to classes of documents

4.4 Selective eviction handling

We pushed our investigation further by attempting to assess to which extent saving locally evicted blocks in remote client memories affects the performance of cooperative caching operating in a cluster-based server environment. Again using the notations from Section 3, here is the definition of these policies:

- handle only evictions of blocks belonging to files from class 2, those that account for almost 40% of the total servicing time
- handle the evictions of the blocks in classes 0 and 1, that is popular documents less than 10 KB
- handle evictions for class 0 only, i.e., the most popular documents, representing 50% of the requested files

The results are reported in Figure 4 which also compares them with those of the fully-replicated solution. Notice that the first policy comes very close to the performance of the replicated solution which underlines again the importance of keeping class 2 files cached in memory, this time due to eviction handling. The performance degradation of the other two policies shows that the smaller and the more popular the file is, the less important the eviction handling. That can be understood if we remember again that class 0 accounts for 50% of the requests which makes very probable to find a cached copy of the block in remote client memories and thus rendering futile the saving of such blocks. As the sizes of the document grow and their popularity decreases, eviction handling becomes more important.

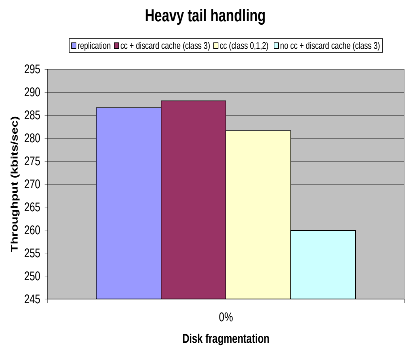

Figure 5. Heavy tail caching

4.5 Handling the heavy tail of the request distribution curve

This part of the request distribution curve represents 1% of the requested documents but takes some 25% of the total servicing time. Our previous experiments showed the importance of caching large documents from classes 2 and 3. In this subsection we try to get more fine-grained insight about this issue by separately treating the unpopular documents. We wrote three other policies in which we attempt to cache these documents at the discard cache. The policies using cooperative caching handled evictions in class 2. The three policies are:

- cooperatively cache all documents and use the discard cache to keep copies of the large documents of class 3
- cooperatively cache classes 0, 1 and 2 without caching class 3 at all
- cache only class 3 documents using the discard cache

The results are shown in Figure 5 and, as usual, they are compared to those of the replicated solution. The poor performance of the third policy shows that caching the large files doesn't help if all the other requests go to the disk. That becomes clear when comparing the performance of the third policy with that of the second policy which doesn't cache class 3 at all and yet performs significantly better. The best solution is the mix of the last two which shows slightly better performance than the solution using replication.

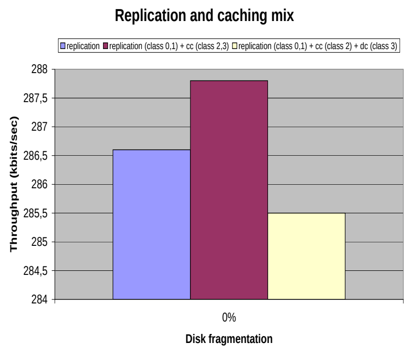

Figure 6. Combining replication with caching

4.6 Mixing replication with cooperative caching

In this section we present the results of a mixed solution that uses both replication and cooperative caching (see Figure 6). We wrote and tested two policies. In the first one, we replicate classes 0 and 1 on local disks of the two servers and use cooperative caching for classes 2 and 3. The second policy uses replication for small and popular documents, cooperatively caches class 2 and uses the discard cache to store unpopular documents. Both policies handle eviction for class 2 only. The first policy outperforms the fully-replicated solution and its results are consistent with our previous observation according to which caching classes 2 and 3 yields the best performance. The fact the using replication for the small and popular files doesn't affect this conclusion indicates that these classes enjoy enough locality due to their popularity (since there are only two servers, there is a 50-50 probability that a second request for the same document will hit the same server).

5 Related work

A fairly broad survey on locally distributed Web servers can be found in Cardellini et al. [4]. Locality-aware request distribution [14] aims at reconciling load balancing with data reference locality at the back-end level. An alternative to this approach would be to use cooperative caching [8, 16], but the results were shown to be similar [14].

Exclusive caching systems [19, 5] strive to avoid double buffering occurring in independently managed caches (e.g., filesystem page/buffer cache and disk controller cache). Serverless filesystems [1] aim to improve the I/O performance by distributing the metadata management and by using cooperative caching [8, 16]. Through cooperative caching, the traditional distributed filesystem memory hierarchy (local cache, server cache, server disk) is extended through the support of client caches, as long as reading from a remote client cache takes less time than accessing the disk. With such systems the working set can scale beyond the limit of the locally available memory. PACA [6] is a parallel filesystem that mixes cooperative caching with global memory and Remote DMA (RDMA). The Clusterfile [11] parallel filesystem uses cooperative caching to improve collective I/O.

6 Conclusions

This paper describes request distribution-aware caching for cluster-based Web servers. Our caching system uses cooperative and exclusive caching to speculate on the properties of Zipf-like request distribution curves for static Web documents by selectively caching classes of documents according to their popularity. Our evaluation showed that cooperative caching bridges the performance gap between virtual disks and fully-replicated solutions especially when targeting the class of large, unpopular files. Handling block eviction pays off for unpopular and large documents only. Separate handling of the heavy tail of the request distribution curve may bring further benefits.

References

[1] T. Anderson, M. Dahlin, J. M. Neefe, D. Patterson, D. Rosseli, and R. Y. Wang. Serverless Network File Systems. In *The 15th Symposium on Operating System Principles*, December 1995.

[2] Apache. *http:/www.apache.org/*.

[3] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, December 1995.

[4] V. Cardellini, E. Casalicchio, M. Colajanni, and P. S. Yu. The State of the Art in Locally Distributed Web-Server Systems. In *ACM Computing Surveys, Vol. 34, No.2, pp. 263-311*, June 2002.

[5] Enrique V. Carera and Ricardo Bianchini. Improving Disk Throughput in Data-Intensive Servers. In *Proc. of the 10th IEEE Int. Symposium on High Performance Computer Architecture*, February 2004.

[6] T. Cortes, S. Girona, and L. Labarta. PACA: A Distributed File System Cache for Parallel Machines. Performance under Unix-like workload. Technical Report UPC-DAC-RR-95/20 or UPC-CEPBA-RR-95/13, Departament d'Arquitectura de Computadors, Universitat Politecnica de Catalunya, 1995.

[7] M. E. Crovella, R. Frangioso, and M. Harchol-Balter. Connection Scheduling in Web Servers. In *Proc. of the 2nd Usenix Symposium on Internet Technologies and Systems*, October 1999.

[8] M. Dahlin, R. Yang, T. Anderson, and D. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *The 1st Symp. on Operating Systems Design and Implementation*, November 1994.

[9] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proc. of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, December 1995.

[10] M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal. Size-Based Scheduling to Improve Web Performance. In *ACM Transactions on Computer Systems, Vol. 21, No.2, pp. 207-233*, May 2003.

[11] F. Isaila, G. Malpohl, V. Olaru, G. Szeder, and W. Tichy. Integrating collective I/O and cooperative caching into the Clusterfile parallel file system. In *Proc. of the 18th ACM Int. Conference on Supercomputing (ICS)*, June 2004.

[12] V. Olaru and W. F. Tichy. CARDs: Cluster Aware Remote Disks. In *Proc. of the Third IEEE/ACM Int. Symp. on Cluster Computing and the Grid (CCGrid 2003)*, May 2003.

[13] V. Olaru and W. F. Tichy. On the Design and Performance of Remote Disk Drivers for Clusters of PCs. In *Proc. of the Int. Conference on Parallel and Distributed Processing, Techniques and Applications (PDPTA'04)*, June 2004.

[14] V. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Proc. of the ACM Eighth Int. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)* , October 1998.

[15] S. Gadde R. P. Doyle, J. S. Chase and A. Vahdat. The Trickle-Down Effect: Web Caching and Server Request Distribution. In *Proc. of 6th Int. Workshop on Web Caching and Content Distribution (WCW'01)*, June 2001.

[16] Prasenjit Sarkar and John H. Hartman. Efficient Cooperative Caching using Hints. In *Proc. of the 2nd Symp. on Operating Systems Design and Implementation*, October 1996.

[17] The Standard Performance Evaluation Corporation (SPEC). *http://www.spec.org/web99/*.

[18] Myricom Inc. GM: the low-level message-passing system for Myrinet networks. *http://www.myri.com/scs/index.html*.

[19] Theodore M. Wong and John Wilkes. My Cache or Yours? Making Storage More Exclusive. In *Proc. of the USENIX Annual Technical Conference*, pages 161–175, 2002.

[20] G. Zipf. *Human Behavior and the Principle of Least Effort*, 1949. Addison Wesley.