Cooperative Caching in Linux Clusters[1]

YING XU BRETT D. FLEISCH

Distributed Systems Lab
Department of Computer Science and Engineering
University of California
Riverside, CA 92521, USA
{yxu, brett}@cs.ucr.edu

Abstract. Most operating systems used in cluster systems manage memory locally without the benefits of cluster-wide cooperation. During typical use there will be some computers in a cluster that are short of memory while others have idle memory wasted. This significant deficiency raises a question: how to improve the cluster operating system to support the use of cluster-wide memory as a global distributed resource. In this paper, we propose a cooperative caching scheme, which not only allows the operating system to avoid many expensive disk accesses by using cluster-wide memory for file caching but also improves the overall throughput of cluster file systems by reading files directly from the cache of other sites. And we also describe corresponding changes in Linux kernel memory management to support this cooperative caching scheme.

Introduction

Recent technology trends in high-bandwidth and low latency networks allow data to be transferred very efficiently between systems. The data transfer rate of modern networks, such as Myrinet and Gigabit Ethernet, has already exceeded that of disk, as shown in Table 1, and continues to improve at a faster rate. The seek and rotational latencies of disk are expensive compared to the initial latency of networks. We could expect fetching data from remote memory through a high-bandwidth, low-latency network to be ten to twenty times faster than reading data from disk on average [1].

Table 1. **Network vs. Disk** [2].

	Data Transfer Rate (Mb/s)	**Latency (μs)**
Fast Ethernet (100-BaseT)	100	60 – 100
Myrinet	1,250	8 – 38
Gigabit Ethernet	1,000	30 – 50
SCSI Disk [8]	400 – 600	3, 000 – 8,000

[1] This research was supported in part by the Office of Naval Research under grant N00014-01-1-0854.

In addition, memory prices continue to decline making it possible for cluster computer systems to continue to increase capacities and offer new services that were once memory-limited. Nonetheless, utilization of the installed additional memory in clusters has been exceptionally low. For example, we analyzed a two-day memory usage trace of one main computing server in the Computer Science and Engineering Department at UC Riverside. This system has 883 MB main memory and is concurrently used by an average of 50 users. Our finding indicated that on average more than 30% of the memory is underutilized. While this simple set of experiments is not entirely conclusive or rigorous, these experiments suggest there is a tremendous amount of cluster-wide memory underutilized. Our findings are reinforced by findings in [9] where experiments showed 60-68% of the memory was underutilized. Both our experiments and published results [9] suggest that if each site in a cluster has a large size memory but only manages its memory locally that memory will be underutilized.

Our approach is to use cluster-wide memory. Cluster-wide memory is implemented using the remote memory of peer sites in the cluster. Remote memory has some interesting properties:

- **Speed.** Fetching data from remote memory is faster than obtaining the data from local disk.
- **Dynamic Size.** The size and the distribution of remote memory are changing dynamically according to the memory usage of each site because as users log in and out of the cluster, and applications begin and end execution, memory requirements of cluster users and applications changes radically.
- **Unreliability.** As the number of sites in a cluster grows, the possibility of a site failure increases. This requires that we treat remote memory as an unreliable resource.

Design Decisions

Our goal is to efficiently use remote memory as a global distributed resource and to make it resilient to site failure. Based on the properties of remote memory, we made our first design choice to use remote memory as a file cache. The file cache is a layer between the memory management system and the file system. It caches file data in main memory to reduce disk related delay and contention. By managing remote memory globally, active sites store file data remotely in the idle memory of other sites. Using remote memory at file cache level also makes remote memory 'transparent', which means other system-level software and user-level applications can easily benefit from remote memory without any modification. By coordinating the cluster-wide file caches, we expect to:

- Avoid expensive disk accesses.
- Improve file system performance. Server load can be reduced by forwarding requests to the clients that cache the requested data blocks. Multiple sites may cache the same data block, therefore they can serve the requests for the data block simultaneously, which results in improved throughput.

There is overwhelming evidence that adequate server memory is the most important factor in overall server performance. A secondary consideration is the exact underlying filesystem the server uses. Some filesystems use cache memory more effectively than do others, yielding overall higher read cache effectiveness at the server. Higher effectiveness of reads satisfied from file cache result in fewer disk read I/O operations and therefore a higher percentage of write I/O operations. Therefore, a second design choice we made was to use the remote memory as a read-only cache. We consider only changes to the filesystem's Read request protocol but not the Write protocol. We concentrate here on designing highly effective read caching using remote memory and assume a modern filesystem that supports highly optimized disk writes (e.g. journaled file systems such as Veritas, JFS, VXFS). Our focus on read caching also addresses the particular concern of site failure. A site storing remote data blocks may fail at any time; remote memory cannot be assumed reliable. While replication of data at multiple sites can be used to increase reliability, maintaining replicas can be costly and replication overheads depend on the frequency of failures expected [10]. However, read-only caches are much easier to handle. If cached data is lost due to a site failure, data can be re-read again from the server.

Cooperative Caching Scheme

In our cooperative caching scheme, we assume that each site in a cluster trusts each other. Sites may fail at any time, however. We also assume there is a file server in the cluster. The file server uses a data structure called *cache directory* to track the file blocks cached by each client. Given a file identifier and a block number within the file, a lookup in the cache directory yields the information about the data block, such as the current version number of the data block and which sites cache the data block.

We extend the read policy of the cluster file system for our system improvements but we don't change the write policy. The write policy can be write through, write-back-on-close or write delay [2]. All modified data blocks are still sent to the server as they otherwise would be without cooperative caching.

Scenario 1 (Fig.1) When Client P does not find a block in its local cache, it sends a READ request to the server. The server consults its cache directory. If no client is caching the data, the server checks its local cache. If the requested data block is not in its local cache, the server reads the data from disk, stores it in its local cache and sends it to P. Otherwise, the server directly sends the data from its local cache. Once P receives the data, the server creates a new cache directory entry indicating P caches the data block.

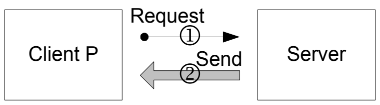

Fig. 1. Scenario 1

Scenario 2 (Fig. 2) If any client (Client Q) is caching the requested data and the server is experiencing heavy load, the server forwards the READ request to that client. Q sends the data to P. Once P receives the data, it sends a NOTIFY message including the *version number* of the data block to the server to indicate that P has received it. After the server receives the notification, it compares the version number it received with the current version number of the data block. If they match, the server updates its cache directory and sends a positive acknowledgement (PACK) to P. Otherwise it sends a negative acknowledgement (NACK) to P. P discards the data block it received if a NACK is received and resends the request.

To avoid high client load, the server needs a policy to forward the requests evenly to available clients. The policy can be a simply round-robin policy or an adaptive policy based on the load on each client.

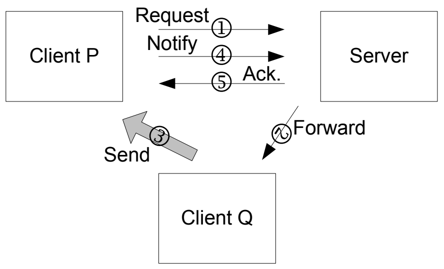

Fig. 2. Scenario 2

Scenario 3 (Fig. 3) To offload requests to the server, not every request needs to go through the server. In step 3 of Scenario 2, when Q sends the data to P, it also tells P what other data blocks in the same file it caches. When P requests blocks within the same file that Q has already cached, it directly sends requests to Q. Once P receives the data, like scenario 2, it notifies the server. The server replies with PACKs or NACKs as in scenario 2.

In case the requested block in Q is no longer available due to the page replacement. Q forwards the request to the server. The server will serve or re-forward the request by looking up the block in its cache directory.

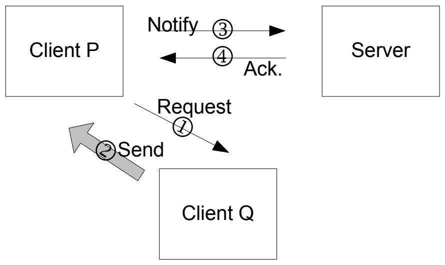

Fig. 3. Scenario 3

The file server maintains global information about all cached data copies in the cache directory. Whenever a file is changed or deleted by any client, the server also updates the records of corresponding data blocks of the file in the cache directory. Depending on the consistency model of higher level network file systems, clients and the file server may have different interactions. If the network file system provides strict consistency, which guarantees clients a consistent view of data, whenever a client opens a file for write, the server must notify 1) the last writer of the file to write all dirty blocks back to the server or 2) ask the clients read-caching the file to invalidate the data blocks of the file. If the network file system doesn't provide strict consistency, for example, NFS, caches may be allowed to be inconsistent within a small time window. Clients never send a message to the server to revalidate any cached data block until the cache has been time out. In this case, Scenario 2 of our cooperative caching scheme can even be simplified (See Fig. 4). The server forwards the request to Q together with current version number of the requested data block. Q compares the version number it receives with the version number of the data block it caches. If they match, Q send the data block to P. Otherwise, it notifies the server that it doesn't have the latest version of the data block. The server will serve or re-forward the request after updating the cache directory.

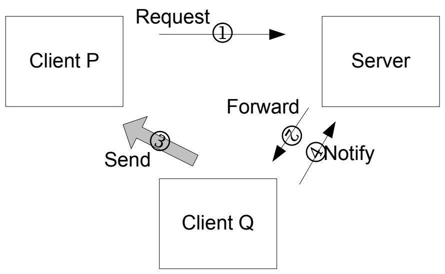

Fig. 4. Scenario 2 modified for less strict consistency

The cache directory at the server is normally kept up-to-date as shown above. However, the cache directory may become inconsistent once a site failure occurs. The file server is enhanced to handle this situation. Assume there is a time window Δ_1 before the server detects the failure and another time window Δ_2 within which the cache directory is not consistent. Let's go back to Scenario 2 and 3, for example, Q fails. Within Δ_1, some requests have already been forwarded to Q and some requests are still being forwarded or sent to Q because P and the server have not noticed the failure of Q yet. After a timeout period, P resends the requests to the server, which tells the server that it cannot obtain the data. The server will then either serve the request or forward it to another client. Once the server detects the failure of Q, it avoids forwarding any requests to Q and starts a thread to clean the cache directory, which checks each entry of the cache directory and removes Q from the clients that cache the corresponding data block. The server is now in time window Δ_2. It continues serving and forwarding the requests but marks Q as unavailable. After the cache directory has been cleaned and become consistent again, the server returns to normal operations.

Adding more clients doesn't add too much load to the server because the added clients can also be leveraged to serve the requests for the server. By coordinating the file caches of all clients, most of the disk read accesses in the server are avoided and instead of serving large data blocks the server only sends small forward messages and receives small notifications and acknowledgements. The extra work added to the server is searching and maintaining the caching directory, which can be easily overlapped with the IO activities by the operating system. Thus the extra work isn't expected to affect the performance of the server very much but the overall throughput of the system is improved.

Remote Page Replacement

The goal of our remote page replacement is to keep the most valuable pages in the cluster-wide remote memory without impeding local performance. We chose Linux to implement our prototype system, so our algorithm is discussed in the context of Linux kernel 2.4 memory management. The name of our prototype is called YFS. We first start with a brief introduction of page replacement in Linux 2.4, and then present our algorithm. A short discussion of our algorithm is at the end of this section. We also compare our algorithm with the global aging algorithm of GMS [3, 4] and N-Chance forwarding [5] in Section Comparison with Other Works.

Page Replacement in Linux 2.4

Linux kernel 2.4 maintains several LRU page lists as follows [11]:

- **Active list** Pages on the active list have age > 0, may be clean or dirty, and may be (but are not necessarily) mapped by process Page Table Entries (PTE).
- **Inactive dirty list** Pages on the inactive dirty list have age = 0, may be clean or dirty, and are not mapped by any process PTE.
- **Inactive clean list** Each memory zone has its own inactive clean list, which contains clean pages with age = 0 and not mapped by any process PTE.

The pages in the inactive clean lists are always the first choice for page replacement. When memory gets low and there are not enough pages left in the inactive clean lists, the inactive dirty list is scanned. The pages in the inactive dirty list that are actually clean will be moved to the inactive clean lists, while the pages that are dirty will be first written out to disk and then moved to the inactive clean lists. The active list is scanned periodically. If the pages in the active list are not referenced, their age will be gradually reduced. Once the pages in the active list have age equal to 0, they will be moved to the inactive dirty list.

The Algorithm

A data block which is the last copy cached by any client on all sites is called a *singlet*. Singlets are more valuable than other cached copies. If the singlet is discarded, the next time when any client wants the data back, the client has to make an expensive read from the server's disk to obtain it. Reading a data block from disk is an order of magnitude slower than obtaining it from remote memory as we mentioned earlier. So our page replacement algorithm tries to avoid replacing singlets.

We changed the original Linux kernel 2.4 page replacement algorithm [3] to fit it into our cooperative caching model by adding two additional LRU lists: the network file cache list and the singlet list. Pages on the network file cache list are clean, have age=0, cache a data block of the network file system and are not mapped by any process PTE. Pages on the singlet list have ever been singlets, have age=0 and are not mapped by any process PTE. The following is a short description of our page replacement algorithm (as shown in Figure 4):

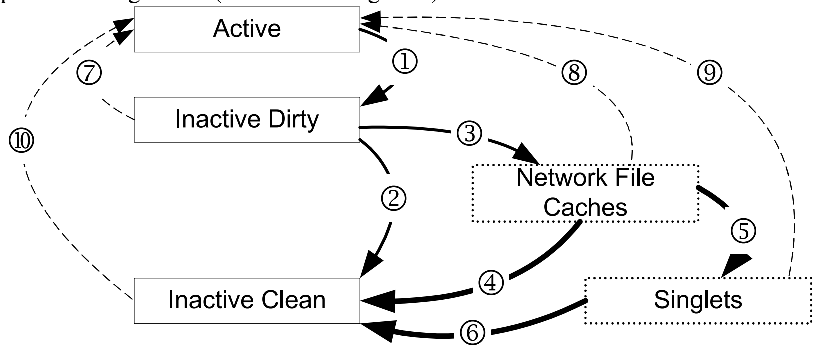

Fig. 5. Page Replacement Algorithm for Cooperative Caching

①: The active list is scanned periodically. If the pages in the active list are not referenced, their age will be gradually reduced. Once the pages in the active list have age equal to 0, they will be moved to the inactive dirty list.

②③: When memory gets low and there are not enough pages left in the inactive clean lists, the inactive dirty list is scanned. The pages in the inactive dirty list that are actually clean and not network file caches will be moved to the inactive clean lists, while clean network file caches will be moved to the network file cache list. The pages that are dirty will be first written out to disk or the file server and then moved to the inactive clean list or the network file cache list.

④⑤⑥: The network file cache list and the singlet list are also scanned periodically. When memory gets low, the two lists are scanned more frequently. When scanning the network file cache list, the client first collects what pages are in its network file cache list and sends a message to the server to differentiate the pages. Upon receiving this message the server searches the cache directory, returns the client which pages are singlets and which are not, and updates its cache directory. The client then moves the singlets to the singlet list and others to inactive clean list according to the returned message. The singlet list is scanned in the similar way like the network file cache list. The client first sends a message telling the server what are in its singlet list. After receives the reply from the server, the client moves the pages that were once singlets but are now not singlets to the inactive clean list.

④⑤⑥ handle pages aggregately so there is no network transaction for each data block to be discarded, which significantly reduces the total number of messages sent to the server to identify singlets.

⑦⑧⑨: Once a page in the inactive dirty list, the network file cache list or the singlet list is referenced it will be moved to the active list.

⑩: When a page in the inactive clean list is referenced, if it is not a network file cache, it will be moved to the active list directory. Otherwise, the client will send a message to the server to check whether it is the current version. If it is, it will be moved to the active list. Or it will send a message to the server to request the data block.

If the memory is still under pressure, the client can only reclaim the pages from its singlet list. It sends to the server how many pages it wants to discard from its singlet list. The server returns it how much idle memory available on other clients respectively. If the total amount of idle memory pages is larger than that of the singlets the client wants to discard, the client then forwards the singlets to other clients' idle memory and updates the server's cache directory. Otherwise, the client has to discard some singlets. To choose which singlets to discard, we use a similar scheme as N-Chance Forwarding [1] by giving each singlet a recirculation count. Each time the singlet is forwarded to another client, the recirculation count is increased by 1. The client chooses to discard the singlets with the largest recirculation count. In some extreme situations, the client may not have time to move the singlets to other clients. It simply discards the singlets in this case and sends a message to the server what singlets have been discarded. Because all singlets are clean, even they are discarded, they can still be read from the server when needed.

When the server notices that the memory in a client is under pressure, for example, when the client frequently reclaims the pages in its singlet list, the server will send a multicast message to other clients asking them to scan their active lists, inactive dirty lists, network file cache lists more often to move more pages to the inactive clean lists.

The page replacement algorithm described above meets our design goal by keeping the singlets, which are more valuable, as long as possible. When the memory is not under pressure, our remote page replacement algorithm runs similarly to the original page replacement algorithm of Linux kernel. Active clients will tend to move file caches that do not belong to them out of their local cache quickly, while idle clients will tend to accumulate file caches of the active clients and hold them in memory for long periods of time.

Comparison with Other Works

The most aggressive approach to use remote memory is the XMM subsystem [4] in the Mach microkernel. XMM integrates all distributed memory management in a single subsystem and enables complete cross-site transparency at the virtual memory level. Any process in the system is allowed to access memory in the same fashion as if it were executing on a single system. Our cooperative caching differs from XMM in the design decision and the underlying assumption. We choose to use remote memory only as read-only file caches. Our concern is the overhead of maintaining the state and the consistency of the remote memory may counteract its benefit. We assume each site except the file server may fail at any time and our cooperative caching can gracefully handle site addition, deletion and failure. However, XMM doesn't have such mechanisms. The possibility of the site failure increases with the number of sites. So the claim that XMM can scale to over 1000 sites [4] bears serious questioning.

Oracle's Cache Fusion [7] is designed for shared-disk clusters. It enables multiple sites in the cluster to share a global block device cache. Although it is not a scheme for file system caching, it does share some similarities with our cooperative caching. Like cooperative caching, cache fusion allows data blocks to be read directly from caches of other sites, which eliminates the need for extra disk I/Os. The cache consistency of Cache Fusion is maintained by a resource control mechanism, namely the Global Cache Service. Both read-sharing and write-sharing are supported in Cache Fusion. Unlike our cooperative caching, Cache Fusion doesn't balance the memory usage across multiple sites

GMS [5, 6] separates global memory from local memory while our algorithm doesn't have such separation. GMS uses a global page aging algorithm. When a faulted page is read from disk, GMS tries to discard the oldest page in the cluster. Our algorithm doesn't use global aging information for page replacement; instead it considers singlets to be more valuable than other pages and keeps them as long as possible. It tries to avoid the overhead of having global information. When the memory is not under pressure, our page replacement algorithm doesn't need any global information. Only when the memory is under pressure and the pages in the network file cache list or the singlet list are being reclaimed will our algorithm ask for the global information from the file server.

Our work is closely related to the cooperative caching proposed by Dahlin et al. [1]. The N-Chance forwarding algorithm is the best algorithm estimated by Dahlin et al [1]. When the algorithm is about to replace a singlet, it forwards the singlet to a random peer. Each singlet has a recirculation count N and the singlet is discarded after it has been forwarded N times. The N-Change forwarding has the advantage that global information is unnecessary, but the performance cost of random forwarding mistakes is also high. In N-chance, a singlet may be forwarded multiple times to sites without idle memory. This causes the unnecessary overhead of the sites to receive the singlet and forward it to other site. Our page replacement algorithm doesn't have this specific problem. The server always has up-to-date information on where the clients forward singlets.

Summary

In this paper, we propose a cooperative caching scheme in the context of cluster systems. We are implementing a prototype system YFS based on efficient use of remote memory for read caching; our scheme avoids expensive disk accesses and we expect will improve the overall throughput of the cluster file system. We also designed a page replacement algorithm for our cooperative caching scheme, which in most cases keeps the most valuable pages in the cluster-wide memory as long as possible without degrading local performance.

References

[1] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. *In Proc. of the First Symp. on Operating Systems Design and Implementation*, pages 267-- 280, November 1994.

[2] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1):134--154, February 1988.

[3] Outline of the Linux Memory Management System. http://home.earthlink.net/~jknapka/linux-mm/vmoutline.html.

[4] Dejan S. Milojicic, Randall W. Dean, Michelle Dominijanni, Alan Langerman, Steven J. Sears. Extended Memory Management (XMM): Lessons Learned. *Software Practice and Experience*, pages 1011--1031, Volume 28, 1998.

[5] M. J. Feeley, W. E. Morgan, and etc. Implementing Global Memory Management in a Workstation Cluster. *In Proc. 15-th Symposium on Operating Systems Principles*, pages 201--212, December 1995.

[6] M. Feeley. Global Memory Management for Workstation Networks. PhD Thesis, University of Washington, 1996.

[7] T. Lahiri, Vinay Srihari, and etc., Cache Fusion: Extending Shared-Disk Clusters with Shared Caches, *Proceedings of the 27th VLDB Conference*, Roma, Italy 2001.

[8] Seagate Technology. http://www.seagate.com/.

[9] A. Acharya and S. Setia. Availability and utility of idle memory in workstation clusters. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems*, pages 35--46, 1999.

[10] O. E. Theel, B. D. Fleisch. Design and Analysis of Highly Available and Scalable Coherence Protocols for Distributed Shared Memory Systems based on Stochastic Modeling, *Proceedings of the 24th International Conference on Parallel Processing*, Oconomowoc, Wisconsin, August 1995, pp. I:126-130.

[11] Linux Memory Management. http://linux-mm.org/.