Workshop 04

Distributed Systems and Algorithms

PACA: A Cooperative File System Cache for Parallel Machines*

Toni Cortes, Sergi Girona and Jesús Labarta

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya - Barcelona
E-mail: {toni, sergi, jesus}@ac.upc.es
URL: http://www.ac.upc.es/hpc

Abstract. A new cooperative caching mechanism, PACA, along with a caching algorithm, LRU-Interleaved, and an aggressive prefetching algorithm, Full-File-On-Open, are presented. The caching algorithm is especially targeted to parallel machines running a microkernel-based operating system. It avoids the cache coherence problem with no loss in performance. Comparing our algorithm with another cooperative cache one (N-Chance Forwarding), in the above environment, better results have been obtained by LRU-Interleaved. We also evaluate an aggressive prefetching algorithm that highly increases read performance taking advantage of the huge caches cooperative caching offers.

1 Introduction and Related Work

In this paper we present PACA, a specific cooperative caching mechanism built on top of a microkernel-architecture operating system. PACA defines a single parallel global cache built from the union of all the small local caches across the different nodes. As part of PACA, we have studied several caching and prefetching policies. From all the considered policies, special interest will be placed on LRU-Interleaved (caching algorithm with no cache coherence problems) and Full-File-On-Open (aggressive prefetching) algorithms.

All performance data presented in this paper is obtained through simulation. In order to understand the advantages our algorithm has, we compare it with the one presented by Dahlin et al. [4]. In addition to the simulation results presented in this paper, a working prototype has been implemented on top of the PAROS operating system microkernel [9][14].

Most of the work along this line has not been done on parallel machines but on networks of workstations running a *Unix-like* operating system which offers most services. Leff et al. studied the impact distributing cached objects over the network could have [11]. A more practical project was done by Dahlin et al. [4] as part of the xFS file system [1]. They proposed several cooperative caching algorithms and simulated their performance. In their work, N-Chance

* This work has been supported by the Spanish Ministry of Education (CICYT) under the TIC-94-537 and TIC-95-0429 contracts.

Forwarding was identified as the best algorithm. Our work differs from the one presented by Dahlin et al. in two main issues. First, only read operations are studied in their work while both reads and writes are covered in this paper. The second difference is the environment the cooperative caching algorithm will work on. In both previous works, the test bed was a network of workstation running a full *Unix-like* operating system. We work on a parallel machine with a microkernel-based operating system on each node. Most services, like file system operations, are implemented by a user-level server.

Similar approaches to cooperative caching have been taken in data base implementation [7], remote memory paging [13, 12] and memory management [6].

2 Target Environment

The parallel machine this file system is targeted for is made of several nodes with local memory. Full connectivity is offered by the interconnection network. Besides, each node may have none, one or even several disks.

In this work, a couple of differences between parallel machines and network of workstations are assumed. While the second one needs reasonable fault tolerance mechanisms, a parallel machine works as a unit and a node failure means whole machine failure. Besides, parallel machines have a higher interconnection network bandwidth than a network of workstations have.

In our environment, a parallel machine runs a microkernel-based operating system instead of a full *Unix-like* one. All functionalities not offered by the kernel are implemented by servers. This is also the case for the file system operations.

This work started as a file system cache prototype for the PAROS operating system microkernel [9]. This target platform defined the environment we work with. In order to be able to implement our distributed cache, the underlying microkernel should offer, besides the usual abstractions, a *memory_copy* operation. This mechanism will be used to transfer data between nodes. Our assumption is that any processor can set up a data transfer between any other two processors. The processor that invokes the copy is charged with all the overhead. When we refer to a memory copy the copy request and copy itself are all included.

3 Design

3.1 PACA (PArallel CAche)

PACA is a specific cooperative caching mechanism built on top of a microkernel-architecture operating system. PACA defines a single parallel global cache build from the union of all the nodes' local cache. This global management can lead to high performance through a high global hit ratio and good adaptability to the changing needs of the nodes. This will increase the overall system performance. When this mechanism is used we can observe two kinds of cache hits. If the requested block is kept in the local memory we will have a *local hit*. If this block is kept on a remote node we will have a *remote hit*.

As a first step, we have studied the behavior of a centralized single server. A centralized control should be able to cope with a reasonable number of processors. The simulation results show that, in most of the parallel machine installations (with less that 50 nodes), the single server should not cause a bottleneck. The causes behind this reasonable scalability are little server overhead and distributed data transfers. Regardless of these results, work on the distribution of PACA for larger systems is ongoing.

3.2 LRU-Interleaved Algorithm

LRU-Interleaved is a very simple algorithm designed to work as a caching method of PACA. It uses all the available cache in the system as a single cache. It also takes advantage of the parallelism and the high data transfer bandwidth offered by a parallel machines.

We use a set associative block placing algorithm. The number of sets equals the number of nodes and the size of each set is the size of each local cache measured in blocks. When the server has to cache a new block, it applies a hash function to the file name (or file-id) and block number. The result of this function indicates which node will cache the block. Next, a place in the local cache of that node is found using a LRU replacement algorithm. If the hash function is good enough, one of the oldest blocks in the cache will be replaced and the behavior will be an approximation to a global LRU replacement algorithm [3].

As performance is a very important issue in the design of LRU-Interleaved, we have taken several steps in order to increase it as much as possible. The first step towards it consists of designing a simple algorithm. We have eliminated all replication and cache coherence mechanisms. A second step consists of using the potential communication parallelism. When a user requests more than one cache block, these blocks, if in cache, can be sent to the client node in parallel. This parallelization decreases the overhead produced by bringing the data from a remote node.

3.3 Prefetching

Cooperative caching on a parallel machine allows the system to have huge caches. Given these cache sizes, it takes many hours to fill the cache and most of the cached data is more than several hours old. In our simulations (50 nodes and 16MB local caches) the cache needed 13 hours to be filled. This leads us to believe that huge caches should be used for aggressive prefetching. It is well known that prefetching is not always a good idea as it may end up delaying the application if many mispredictions are made [16] [8]. Nevertheless, if the cache is big enough, these mispredictions will not affect the overall cache performance as prefetched blocks will replace very old data. In this work we have studied two different prefetching algorithms: One-Block-Ahead and Full-File-On-Open.

One-Block-Ahead queues the next block to the prefetching queue after each read or write operation. As soon as the disk becomes idle, the first block in the prefetching queue will be fetched.

The best aggressive policy we have simulated so far is the one presented in this paper (Full-File-On-Open). It consists of queuing the whole file on the prefetching queue as soon as the file is opened. With this mechanism most blocks are already in the cache when the application requests them. This algorithm may be too aggressive if the files are very large. As no problems have been detected with the used workload we have not taken this into account.

4 N-Chance Forwarding Algorithm

In order to test how good our algorithm is, we compare it with N-Chance Forwarding [4] which is one of the latest cooperative cache algorithms found in the bibliography. This algorithm divides the cache a workstation has, into two parts. The first one is used to cache the local data and the second one will hold data cached by remote workstations. The size of these two parts is not fixed but dynamically adjusted depending on the node I/O activity.

In order to adapt this algorithm to our environment some changes have been made. The most significant modification is due to the microkernel architecture we work with. In the original version of N-Chance Forwarding algorithm, a *local hit* had no need to access a remote node. The workstation's operating system recognized the requested block as a *local hit* and delivered it directly to the local client. In our model, as the file system is managed by a server, all requests have to be sent to a possibly remote server. We have to notice that this implementation will increase the *local hit* access time and decrease the *remote hit* one. Each *local hit* will be increased the time needed to send the message to the server. On the other side, *remote hits* will only send one message to the server per user request instead of one message for each *remote hit*.

N-Chance Forwarding allows each node to cache the blocks its applications request. This algorithm attempts to avoid discarding unreplicated blocks (singlets) from the cache. When a client discards a block, the server checks to see if that block is the last copy in the whole cache. If the block is a singlet, rather than discarding it, it forwards the data to a random peer. The peer that receives the data adds the block to its LRU list as if it had been recently referenced. This forwarding can only take place N times before the block is referenced again. After N forwardings, if nobody references it, the block is discarded. If a client has a *remote hit*, the block is replicated from the remote cache to the local one of the requesting client.

In this paper we have used two different values for N. The first one, N=0, implements a cache where *remote hits* are possible but no coordination between nodes is done (Greedy policy). The second value, N=2, is used because it was described as the best choice in [4].

Our N-Chance Forwarding version has been implemented as a single server to avoid worrying about cache coherence. In this centralized version, the cache coherence algorithm will not need to communicate different servers decreasing the control traffic between nodes. This simplification has to be taken into account as it may speed write operations.

5 Algorithms Comparison

Before getting into performance details, it would be very useful to compare both cooperative caching algorithms: LRU-Interleaved and N-Chance Forwarding.

LRU-Interleaved is a very simple algorithm. The way blocks are distributed among the nodes makes searching a block very easy and efficient. Straight block location is achieved without using large directory tables. In N-Chance Forwarding there is a special cost on maintaining information about where the blocks are placed. They also have to keep track of which blocks are replicated in order to detect if a block is a singlet or not.

In LRU-Interleaved, much more interest is placed in obtaining full utilization of the cache and avoiding replication than minimizing the amount of data transfer between nodes. On the other hand, the N-Chance Forwarding algorithm places special interest in *local hits* trying to minimize the number of block transferences between nodes.

A third important difference is the cost *remote hits* have on both algorithms. On N-Chance Forwarding a *remote hit* implies a remote copy from the remote cache to the local cache and a local copy from the local cache to the user. On the other hand, a remote copy in LRU-Interleaved only implies a remote copy from the remote cache to the user.

Another difference between the algorithms is the way the cache coherence problem is solved. In our proposal no replication is allowed and thus no cache coherence problems appear. In N-Chance Forwarding some kind of cache coherence algorithm has to be implemented.

6 Simulation Methodology

6.1 Simulator

The file-system cache simulator used in this work is part of DIMEMAS [2] [10], a distributed memory parallel machine simulator.

We will not get into detail of the simulator functionality but the communication model should be explained in order to understand the figures presented in this work. Communications are divided into two parts: a startup and a data transfer. The startup is constant for each type of communication (port or *memory_copy*) and it is assumed to require CPU activity. The data transfer time is proportional to the size of the data sent and the intercommunication network bandwidth. In our model, all communications are synchronous. Asynchronous communication can be achieved by creating new threads.

6.2 Implementation Details

Although we don't want to get into many implementation details, some of them are very important in order to understand the results presented in this paper.

[2] DIMEMAS is a performance prediction simulator developed by CEPBA-UPC and it is commercially available from PALLAS

The file server is a high-priority multi-threaded application which shares the node with other applications.

It is important to note that cleaning or forwarding a block is done after the operation is finished and the user has been notified. This takes the overhead away from the critical path of the operation. From now on, we will refer to these situations as *delayed clean* and *delayed forward*. These delayed operations will be possible as long as the file server does not run out of auxiliary buffers.

As we want to optimize the critical path of the read and write operations we give them higher priority over prefetching operations.

7 Performance

7.1 Sprite Workload

In order to get the results presented in this paper, we have used the Sprite workload [2]. These traces contain the activity of 48 client machines and some servers over two day period measured in the Sprite operating system. All measures presented in this paper are taken from the 15th hour to the 48th hour in order to study the behavior of a warm cache. We have used this trace as we believe that parallel machines should not only be used for parallel applications but also for *Unix-like* ones in a time sharing manner.

7.2 General Information and Simulation Parameters

In the following subsections, most of the parameters used in the simulation are fixed. Unless otherwise specified, all runs simulated a 50 nodes machine with a 16MB of local cache, 8KB cache blocks, no prefetching and a 30 seconds sync.

We assume disk time accesses as described by Ruemmler and Wilkes [15]. Reading a 8KB disk block takes 14.7 milliseconds while writing it takes 18.3 milliseconds. All measures in this paper are taken with only one disk connected to the node where the centralized file server runs.

Unless otherwise specified, nodes are connected through a 155 Mbits/s interconnection network and local copies are done at 320 Mbits/s. We assumed a 100 microseconds port startup and a memory copy one of 50 microseconds.

In the following subsections we will use a few short expressions in order to identify several common situations. When a user requests a block that it is not in the cache it is called cache miss. This requested block has to replace another block already cached. This replaced block may have been modified since its last update to disk, or may not have. We will refer to the first situation as a *miss on dirty* and to the second one as a *miss on clean*.

7.3 Read and Write Performance

In this subsection we will study the performance of read and write operations with LRU-Interleaved, N-Chance Forwarding and Greedy policies (Fig. 1).

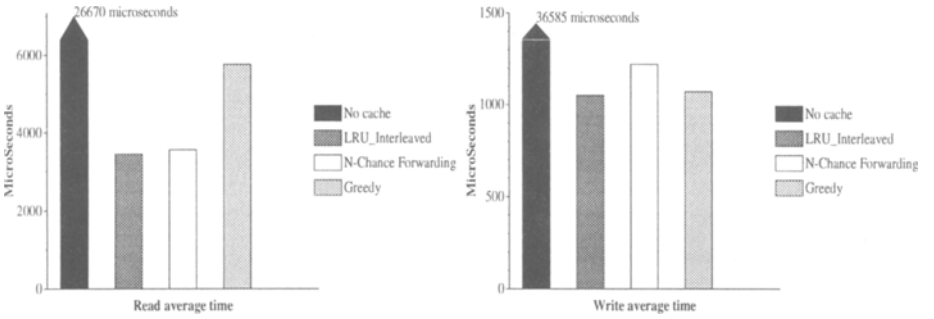

Fig. 1. Average READ and WRITE time for the various caching policies.

An important result is that both cooperative caching algorithms nearly double the read operations bandwidth if compared to non cooperative ones (Greedy and no cache).

If LRU-Interleaved and N-Chance Forwarding are compared, a 3.1% gain from the first algorithm over the second one is observed. Even though Dahlin's algorithm has a much higher *local hit* ratio no proportional gain is observed due to two main reasons: different costs of remote operations and forwardings.

A *remote hit* in N-Chance Forwarding takes longer than in LRU-Interleaved as was explained in Section 5.

There are also quite a few block forwardings that cannot be delayed due to a lack of auxiliary buffers. They have to be included in the critical path of the operation. This usually happens with large requests (100KBytes or more). We have to recall that there is a limit of 16 auxiliary buffers per node.

In Figure 1, we also observe that a write operation with LRU-Interleaved is 14% faster than with N-Chance Forwarding. In order to explain this difference we should first explain why Greedy is also faster than N-Chance Forwarding. The main difference between these two algorithms is that the first one does not forward blocks while the second one does. The overhead due to not delayed forwardings increases the write time a lot. A not delayed forwarding implies an extra remote memory copy in the write operation. As write operations are very fast, this extra time affects the overall write time significantly.

LRU-Interleaved is also faster than Greedy because of the dirty blocks. As N-Chance Forwarding and Greedy algorithms clean a dirty block just before forwarding it [5], the syncer does not have enough time to clean all blocks before being forwarded. This does not happen in LRU-Interleaved as blocks are cleaned once they are completely discarded. The overhead of block cleaning is higher than the time lost because of *remote hits*.

Finally, another important issue is the different cost *remote hits* have in LRU-Interleaved compared to N-Chance Forwarding and Greedy. The same explanation as in the read operations applies.

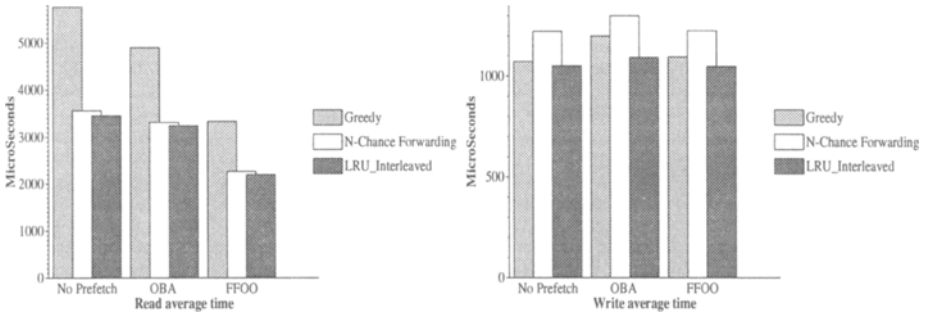

Fig. 2. Average read and write times due to the PREFETCHING policy.

7.4 Prefetching Influence on Read and Write Operations

The traditional prefetching policy One-Block-Ahead (OBA) decreases the read time a little bit (Fig. 2). With this algorithm not much gain is obtained. This is because starting to prefetch a block just after accessing the previous one is not early enough. When the user needs the prefetching block, it is still on its way from the disk.

Another problem this algorithm has is that random accesses see very little gain. If blocks are accessed in a random way, the block prefetched are not the ones the application needs.

These problems found in One-Block-Ahead have lead us to implement the Full-File-On-Open policy (FFOO). If we start prefetching the file once it is open, the probability to have finished the prefetching of a block before it is needed increases. Besides, if a random access is performed, most blocks in the first part of the file will have been prefetched before requested.

The Greedy algorithm improves much more than any other as its hit ratio is very low when no prefetching is done.

If we move to the write operations, the first thing we notice is that no real gain is obtained with either algorithm. This is because increasing the hit ratio does not increase the write performance. A *miss on clean* takes the same or even less amount of time than a cache hit. For instance, *remote hits* on N-Chance Forwarding are even more expensive than *misses on clean* [3].

Another problem appears when the block to be written is being prefetched. This write will take some of the disk read time.

On the other hand, if a block is prefetched very few *misses on dirty* will happen increasing the write-operation performance. One thing outweighs the other and not much difference is seen on write operations due to prefetching.

7.5 Interconnection Network Bandwidth Influence

In this section, we study the influence the local-remote bandwidth ratio has on the already shown results. Figure 3 presents the percentage gain obtained by LRU-Interleaved over N-Chance Forwarding when the local-remote bandwidth

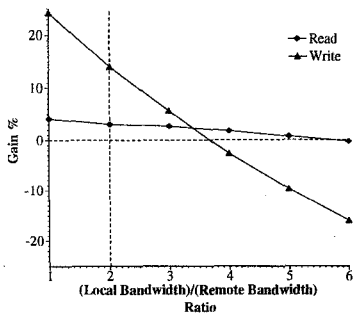

Fig. 3. Influence of the local/remote transference bandwidth.

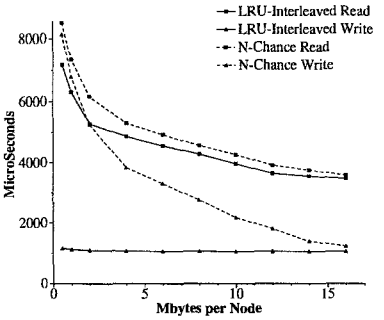

Fig. 4. Read and write performance using different local CACHE SIZES.

ratio is modified. The most favorable case for our algorithm is where there is no difference between a local and a remote transfer. From this point we decrease the remote bandwidth until N-Chance Forwarding reads and writes become faster.

This figure shows that the results presented in this paper are valid even with lower interconnection bandwidths. It also shows that after a certain point N-Chance Forwarding is the way to go.

7.6 Cache Size Influence

In this work we were also interested in studying the effect local cache sizes had on both algorithms (Fig. 4). We can observe that LRU-Interleaved works much better than N-Chance Forwarding when small local caches are used. This is because a higher global hit ratio is obtained by our algorithm due to a better cache utilization. As no replication is allowed, the whole cache contains useful blocks. Dahlin's algorithm loses part of the cache with replicated blocks and it behaves as a smaller one. The benefit of the *local hits* cannot outweight the higher hit ratio obtained by our algorithm.

It is also important to examine the behavior of the write operations. When N-Chance Forwarding is used with small caches the probability to forward a dirty block is very high. This dirty block has to be cleaned before sending it to the new node increasing the write operation time.

8 Conclusions

In this paper we have presented a distributed-cache-oriented file system designed to work on a parallel machine running a microkernel-based operating system. Simplicity, scalability and performance have been the three main objectives in the design. While simplicity and performance have been clearly achieved, more work has to be done if a fully scalable file system is to be obtained.

We have shown that a very simple algorithm obtains similar read bandwidth and a better write performance than more complex ones.

We have also seen that in our environment, as long as the local bandwidth is less than 5 times the interconnection network one, LRU-Interleaved obtains very good results. From this point on N-Chance Forwarding is the way to go.

The Figures have shown some important aspects we should take in account when designing a distributed cache. First, if a relatively fast interconnection network is available, the importance of remote and local hits can be outweighted by other factors like avoiding block forwarding and reducing the number of cleans. Second, remote write hits do not increase write performance and may even decrease it.

We have seen that aggressive prefetching in large cooperative caches may increase the hit ratio and thus decrease the average read time. We have also shown that prefetching very rarely increases the write bandwidth.

More information may be found in the longer version of this paper [3].

Acknowledgments

We owe special thanks to Michael D. Dahlin, E. Markatos and Maite Ortega for their help and useful comments. We are also grateful to the people at Berkeley who gathered the Sprite traces used in this work.

References

1. T.E. Anderson, M.D. Dahlin, J.M Neefe, D.A Patterson et al. "Serverless Network File Systems," *15th SOSP*, December 1995, pp. 109-126
2. M.G. Baker, J.H. Hartman, M.D. Kupfer et al. "Measurements of a distributed File System," *Proc. Of the 13th SOSP*, 1991, pp. 198-212
3. T. Cortes, S. Girona and J. Labarta "PACA: A Cooperative File System Cache for Parallel Machines," *UPC-CEPBA Technical Report RR-UPC-CEPBA-1996-07*
4. M.D. Dahlin, R.Y Wang, T.E. Anderson and D.A Patterson "Cooperative Caching: Using Remote Client Memory to Improve File System Performance," *OSDI'94*, pp. 267-280
5. M.D. Dahlin *Personal communication - 1994*
6. M.J. Feeley, W.E. Morgan, F.H. Pighin et al. "Implementing Global Memory Management in a Workstation Cluster," *15th SOSP*, December 1995
7. M.J. Franklin, M.J. Carey and M. Livny. "Global Memory Management in Cliente-Server DBMS Architectures," *ICVLDB*, 1992. pp. 596-609
8. D. Kotz "Prefetching and Caching Techniques in File Systems for MIMD Multiprocessors," *PhD Thesis from Duke University, Dept. of Computer Science*, 1991
9. J. Labarta, J. Gimenez, C. Pujol, T. Jove and J.I. Navarro "PAROS: Operating System Kernel for Distributed Memory Parallel," *PACTA*, Barcelona 1992
10. J. Labarta, S. Girona, V. Pillet, T. Cortes, L. Gregoris "DiP : A Parallel Program Development," *Euro-Par'96, Lyon, August 1996*
11. A. Leff, J.L. Wolf and P.S. Yu "Replication Algorithms in a Remote Caching Architecture," *IEEE Trans. on Parallel & Distributed Systems, vol. 4, No. 11*, 1993, pp. 1185-1204
12. A. Leff, J.L. Wolf and P.S. Yu "Efficient LRU-Based Buffering ina LAN Remmote Caching Architecture," *IEEE Trans. on Parallel & Distributed Systems, vol. 7, No. 2*, 1996, pp. 191-206
13. E.P. Markatos, G. Dramitinos and K. Papachristos "Implementation and Evaluation of a Remote Memory Pager," *FORTH-ICS Technical Report TR-129*, 1995
14. M. Ortega, T. Cortes and J. Labarta "Implementation of a Cooperative File System Cache on PAROS," *UPC-DAC Technical Report UPC-DAC-96/16*, 1996
15. C. Ruemmler and J. Wilkes "UNIX Disk Access Patterns," *HP Laboratories Technical Report, HPL-92-152*, 1992
16. A.J. Smith "Disk cache - Miss Ration Analysis and Design Considerations," *ACM Transactions on Computer Systems, vol. 3, No. 3*, 1985, pp. 161-203