PACA: A Distributed File System Cache for Parallel Machines. Performance under *Unix-like* workload *

Toni Cortes Sergi Girona Jesús Labarta

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Technical Report UPC-DAC-RR-95/20
or UPC-CEPBA-RR-95/13
toni@ac.upc.es

Abstract

Parallel computers have a great computing power while I/O performance is far behind and it is, very often, the machine bottleneck. In this paper we tackle this problem presenting PACA, a cache directed file system aimed at parallel machines which run both Unix-like and parallel applications. PACA is a specific cooperative caching mechanism that defines a parallel global cache from all nodes local caches. Simplicity, scalability and performance are the main objectives in the design. We also present performance figures under Unix-like workload and compare them with other parallel/distributed caches already implemented.

1 Introduction

Parallel computers are becoming a common sight in scientific and academic environments. Much effort has been laid on increasing the computing power these machines offer, but only, in the last few years, some work has been concentrated on file system performance. This interest in file systems has arose because I/O operations are becoming a significant bottleneck in parallel applications. Most of the solutions given to the problem are usually aimed at parallel applications and in few general cases to *Unix-like* ones, but not both.

In order to increase the disk access performance there are several proposals for new interfaces, run-time libraries, and file systems. Our work tries to design an efficient parallel file system where the cache is the main module. It is well known that traditional caching greatly improves *Unix-like* applications as they, quite often, reuse already accessed data [SMIT85]. We believe that parallel applications can also take advantage of caching and specially of aggressive prefetching.

We think that nodes in a parallel machine, or network of workstations, should cooperate in order to increase their performance. In our work, ways to allow cooperation in the file system caching area are studied. If nodes could not only take advantages of the blocks they have in their cache, but also use the blocks kept in remote caches, a higher read/write bandwidth could be obtained. It is very important for this cooperation to be simple, scalable and should increase the overall file system performance.

In this paper we present PACA, a specific cooperative caching mechanism that defines a single parallel global cache. This main cache is build from the union of all the small local caches on the different nodes. On top of this mechanism we have implemented some caching and prefetching policies. From all the implemented policies, special interest will be placed in LRU-Interleaved and Full-File-On-Open algorithms.

Section 2 introduces some related work. Section 3 describes the target environment and it is followed by the design of PACA, LRU-Interleaved caching policy and some prefetching mechanisms that have been implemented.

*This report has been supported by the Spanish Ministry of Education (CICYT) under contract TIC-94-537.

The next section explains the N-Chance Forwarding algorithm that will be used to compare the results obtained by LRU-Interleaved. In section 6, a generic comparison of LRU-Interleaved and N-Chance Forwarding algorithms is presented. The simulation methodology is detailed in section 7 and it is followed by section 8 where performance results are given. Finally, sections 9 and 10 propose some future work and the conclusions.

2 Related Work

As parallel machines are becoming more popular every day, several parallel/distributed file systems have been implemented. Some of these file systems are RAMA [MILL95], Vesta [CORB94], PIOUS [MOYE94], CM5 sfs [LOVE93], Paragon PFS [BERR94], Mach [ROY93] and Zebra [HART92]. Although they are parallel/distributed file systems, none of them places the special interest in caching we present in this paper.

A different approach has been the design of specialized interfaces to support collective I/O [BORD93] [CORB93]. With a collective I/O interface, all processes make a single joint request to the file system, rather than numerous independent requests. D. Kotz has taken the idea a little further introducing a file system support to the collective I/O resulting in the Disk-directed I/O technique [KOTZ94b] [KOTZ95].

In this work, a cache that is distributed all over the parallel machine is presented. Most of the work along this line has not been done on parallel machines but on networks of workstations. Leff et al. studied the impact that distributing cached objects over the network could have [LEFF93]. A more practical work was done by Dahlin et al. [DAHL94]. They proposed several cooperative caching algorithms and simulated their performance. In their work, interest was only placed in read operations while, in this paper, we increase both read and write performance. They also identified the N-Chance Forwarding algorithm as the best choice. This algorithm, adapted to our environment, will be the reference point in the figures presented later.

Similar approaches have been taken in data base implementation [FRAN92], scalable distributed shared memory [LENO90] and remote memory paging [MARK95].

3 Target Environment

The parallel machine this file system is aimed at is made of several nodes with a local memory. Full connectivity is offered by the interconnection network and nodes can have one or more links to the net. These links can be used in parallel to improve the transference bandwidth to or from a single node. The machine may have several disks (possibly more than one per node) where the data is distributed.

This work was initially aimed at a tightly coupled parallel machine as it relies on high connectivity and communication bandwidth. The viability of these ideas for workstations connected by ATM or similar is also proven in this paper.

We believe that parallel machines should not only be used by parallel application but also by *Unix-like* ones in a time sharing manner. This would help to spread the popularity of these architectures.

This work started as a file system cache prototype for the PAROS operating system microkernel [LABA92][LABA94]. This target platform defined the environment we work with. In order to implement our distributed cache, the underlying microkernel should offer the following abstractions and functionalities.

First, we need the possibility of having multithreaded applications where their threads can run in different nodes. This will let us implement a file server with several threads that will work on different requests in parallel.

A mechanism, like ports, to communicate different applications will be used to send requests to the file server. The server will send a notification to the user once the request is finished using the same mechanism. No data transfers will be done using ports.

The *memory_copy* is the last mechanisms needed. This mechanism will be used to transfer data between the cache and the user memory. Our Assumption is that any processor can set up a data transfer between any other two processors. The processor that invokes the mechanism is charged with all the overhead.

In figure 1, an example of how requests are served is shown. The service starts when a client sends a request to the file server through a port. The server, if the block is not cached,

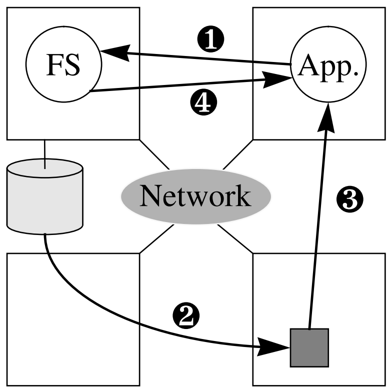

Figure 1: Steps done by the File Server in order to fulfill a request from an application. The displayed read request is not yet cached and thus has to be fetched from disk.

reads it from disk and copies it to the cache. Once cached, the block is copied to the user buffer and the user is notified using another port. All memory copies are done using the *memory_copy* mechanism.

4 PACA Design

4.1 Cache

We want to design a cache that scales at the same rate as the number of nodes in the parallel machine. This could be done by increasing the memory the file server has each time a node is added. This solution ends up with a server node that needs a huge in board memory becoming too expensive. A different, and cheaper, solution is to allow each node to cache its data regardless of what other nodes have. This mechanism scales the cache at the same rate as nodes are added, but it leads to an unbalanced and inefficient use of the allocated memory. Lets suppose that some of the nodes run very intensive I/O applications while others are idle or running very intensive CPU bound processes. In this scene, all the caches in the not I/O bound processors are not used while they could be utilized to increase the other nodes performance.

PACA is a specific cooperative caching mechanism that defines a single parallel global cache. This cache is built from the union of all local caches from the different nodes. This global management can lead to high performance through a high global hit ratio and good adaptability to the changing needs of the nodes and system.

Special importance has been laid on simplicity. If the algorithms are simple enough, good performance, scalability and robustness can be easily obtained in many cases. A first step towards simplicity used in the caching algorithm implemented in PACA is to avoid replication and the associated coherence keeping mechanisms. This lack of replication will also allow a very good cache utilization.

In order to achieve our goal a set associative mapping has been implemented. 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.

Lets discus the algorithm that places blocks in a set and how are they managed within a set. When the server finishes fetching a block from disk, it applies a hash function to the file name and block number. The result of this function indicates which node will hold the block. Next, a place in the local cache of that node has to be found. In order to find which block should be replaced a LRU replacement algorithm is used. The least recently used local block will leave the cache in behalf of the new one. If the hash function is good enough, one of the oldests blocks in the cache will be replaced and the behavior will be an approximation to a global LRU replacement algorithm. If the server wants to check if a block is already cached it only has to apply the hash function and search the block in the resulting node. This last search can be optimized using a hash table of cached blocks per node. Placing and searching a block in the global cache takes minimal CPU time (computing two hash functions) making the file server computing time nearly irrelevant. The above algorithm will be referenced, from now on, as LRU-Interleaved.

As a first step, we have implemented a centralized single server. We have started with a single server as we believe that 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 (less that 50 nodes) the single server is not a bottleneck. The reasons behind this reasonable scalability are little server overhead and distributed data transfers. The file server has very little work to do. It only has to find the block and program its transference. As data movements are done directly from the cache to the client or vice-versa, the server network performance is not penalized. Regardless these results, work on the distribution of PACA for larger systems is ongoing.

We have chosen a delayed write back policy. There is a single thread that cleans all dirty blocks from the whole cache once every 30 seconds as it is done in Unix.

As performance is a very important issue in the design of PACA we have taken several steps in order to increase it as much as possible. The first step towards it was done making algorithms simple and eliminating all cache coherence mechanisms. A second step was done using the potential communication parallelism these architectures have. 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 will decrease the overhead produced by bringing the data from a remote node increasing the performance. In our current version, up to five blocks can be delivered in parallel. Of course, the advantages we may obtain with this parallelism depend on the number of links nodes have and the interconnection network bandwidth.

4.2 Prefetching

Although there are some very big parallel applications that may reuse the file system blocks [AST95], most of them will access files using a unknown pattern and will not reuse the data already accessed [KOTZ94a] [PURA94]. These applications may not benefit from a huge cache as they will not reuse the cached blocks. In order to increase their performance, several methods like Disk-Directed I/O [KOTZ94b] [KOTZ95] are being proposed.

We believe that huge caches can be used to aggressively prefetch the blocks the application may need. It is well known that prefetching it is not always a good idea as it may end up delaying the application if many mispredictions are made [SMIT85] [KOTZ91]. Nevertheless, if the cache is big enough these mispredictions will not affect too much the overall cache performance. Extending this idea, the bigger the cache is, the more aggressive the prefetching policies can be.

The first prefetching algorithm we have implemented on PACA is the typical One-Block-Ahead. When this prefetching policy is active, each read or write operation queues the next block to the prefetching queue. This block will be brought into the cache once the disk becomes idle. If the block is requested before the prefetching was made, it will be extracted from the queue.

Besides the One-Block-Ahead, we have studied some aggressive prefetching policies and good results with this kind of algorithms have been obtained. The best policy we have implemented consists on 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 once the application asks for them. From now on we will refer to this algorithm as Full-File-On-Open. A similar kind of aggressive prefetching was originally implemented in distributed systems as Coda [SATY90]. In this file systems the files were cached in the local disk once they were opened as accessing to the local disk was faster than accessing to a remote disk. In PACA, the file is prefetched to the large in memory cache instead of a local disk. In section 8 some figures will be shown in order to see the benefits that may be obtained when using this policy.

Better prefetching policies could be achieved if the algorithm could extract information from the application in order to correctly predict which blocks to read. We believe that mechanisms like Disk-Directed I/O could be used to send this information to the file server cache and thus improve the prefetching performance.

5 N-Chance Forwarding Algorithm

We want to test how good our algorithm is comparing it with one of the latest ones presented in the bibliography. This algorithm was developed by Dahlin et al. [DAHL94] as part of the NOW project.

This algorithm divides the cache a workstation has in 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.

As our target machine is a parallel machine running a microkernel and not on a network of workstations with a full *Unix-like* operating system on top, some changes have been made to adapt N-Chance Forwarding as we describe in the next paragraphs.

N-Chance Forwarding allows each node to cache the blocks its applications request. The difference with some kind of isolated self caching algorithms is that it attempts to avoid discarding unreplicated blocks (singlets) from client memory. When a client discards a block, the server checks if that block is the last copy cached by any client. If the block is a singlet, rather than throw the block away, the client sets the *recirculation_count* to n and forwards the data to a random peer. The peer that receives the data adds the block to its LRU list as if the block had been recently referenced. If a recirculating block reaches the end of the LRU list, its *recirculation_count* is decremented and it is forwarded again unless the count becomes 0, in which case it is simply discarded. If a client references a singlet, it resets the block's *recirculation_count* and caches the data normally. If a client has a remote hit, the block is replicated from the remote cached and kept in the local cache of the requesting client.

The parameter n indicates how many times a singlet should be allowed to recirculate through different clients' LRU lists without being referenced before finally being discarded. We have used two values for n. The first one, n=0, implements a cache where remote hits are possible but no coordination between nodes is done. From now on, this variant of the N-Chance Forwarding will be called Greedy in order to use the same terminology used by Dahlin et al. in their paper. The second value, n=2, is used because it was described as the best choice in the above paper.

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 simplifications has to be taken in account as it may speed write operations. It is important to notice that no bottleneck has been identified in this single server version of the algorithm under the Sprite workload used in this paper.

6 Algorithms Comparison

Before getting into performance details it would be very useful to compare both LRU-Interleaved and N-Chance Forwarding algorithms. Both of them try to implement some kind of cooperative caching where all nodes in a net share their file system caching space. The main difference between them is the way the cached data is distributed among nodes.

In LRU-Interleaved a very simple with no replication algorithm is used. Much more interest is placed in obtaining a full utilization of the cache achieving a higher global hits ratio than minimizing the amount of data transfer between nodes. LRU-Interleaved relies heavily on high communication bandwidth environments which are starting to be widely spread. On the other hand, the N-Chance Forwarding algorithm places special interest in local hits in order to minimize the number of block transferences between nodes.

Another difference between both algorithms is the way the cache coherence problem is solved. In the distribution mechanisms presented in this paper no replication is allowed and thus no cache coherence problems appear while in N-chance Forwarding some kind of cache coherence algorithm has to be implemented. This difference can become very important once the distributed version of PACA is implemented. Currently, with a single server, the lack of cache coherence algorithm only simplifies the LRU-Interleaved code.

There are several important issues that have to be examined while studying the results. First, we have to see if the overhead introduced by the remote hits can be overweighted by a better cache utilization. It will also be useful to investigate the impact that encouraging local caching has on the number of forwarded and dirty blocks. Finally, the ability of parallelizing data transfers should de studied.

7 Simulation Methodology

7.1 Simulator

The file system and cache simulator used in this work is part of a bigger one that reproduces the behavior of a distributed memory parallel machine [LABA95]. This software not only simulates the machine and disk access but different short term scheduling policies. The whole simulator is a trace-driven one where traces contain CPU, communication and I/O demand sequences of every process instead of the absolute time for each event.

An important issue to understand the results presented later is the communication model implemented in the simulator. All communications are divided in 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 is proportional to the size of the data sent and the intercommunication network bandwidth.

The file system module can simulate the following caching algorithms: no cache (all requests access to the disk directly), N-Chance Forwarding (algorithm to which we want to compare our design) and LRU-Interleaved (the caching policy we present in this paper). As to prefetching policies the following have also been implemented: No-Prefetching, Full-File-On-Open (presented earlier) and One-Block-Ahead (typical one that prefetchs the next block after each operation).

7.2 Simulation Parameters

We assume a disk as described by Ruemmler and Wilkes [RUEM92]. We simulate 8 Kbytes disks blocks which equals the size of the cache block. Reading a disk block takes 17.4 milliseconds while writing it takes 18.3 milliseconds. All measures in this paper are taken with only one disk connected to one of the nodes.

This work tries to test the file system cache performance in two environments: the actual platform where PAROS runs and the network of workstations considered in NOW [DAHL94]. In order to achieve both goals we have used two sets of network parameters. The first one tries to simulate the behavior of the multiprocessor where PAROS is running. In this environment the interconnection network bandwidth is 1.36 Mbytes/s while we can transfer 13.6 Mbytes/s within local memory. The second set of parameters has a 19.4 Mbytes/s interconnection network and a 52 Mbytes/s in memory transfer rate. In order to reference both sets of parameters we will call PAROS to the first one and ATM to the second one. In both configurations we assumed a port startup of 1 millisecond and a memory copy startup of 100 microseconds. Both startups have been obtained from the PAROS operating system performance. Different startups have also been studied and no significant differences where found. In table 1 we can see the time it takes to copy a 8 Kbytes block to a local or remote memory.

Simulation Parameters	Local Memory	Remote Memory
PAROS	674	5844
ATM	250	503

Table 1: Time (in microseconds) that takes to copy a 8 Kbytes block (startup included) from the cache to the client using the PAROS parameters and the ones used in ATM.

In all simulations, a 1 Mbyte per node cache is used unless something different is said.

A final parameter of the simulator is the number of links to the network nodes have. In order to see the impact the number of links have when trying to transfer several blocks in parallel to or from the same node 1, 2 and 4 links have been simulated.

7.3 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. Most of them are optimizations implemented in some, or all, caching policies. In this subsection we explain such optimizations.

The file server is a multithreaded application running on a single node sharing the CPU with some other applications. In order to avoid bottlenecks, the file server has a higher priority than the rest of applications running on the same node. As the server consumes very little CPU time, it will not affect too much the rest of applications in the same node.

It is important to note that cleaning or forwarding a block is done asynchronously and only a local copy has to be done in the critical path of the operation. It is also important that all writes to disk are done also asynchronously by the syncer or on a delayed clean.

Another optimization implemented on the LRU-Interleaved algorithm consists on sending the block requested by a read operation to the user before it is cached. When a client reads a file block, the block is fetched from the disk and sent directly to the user. Once the user has been notified, a copy to the cache is done. This takes a memory copy out of the critical path of a read miss. This optimization was also implemented on the N-Chance Forwarding with no significant increase in performance. For that reason, we decided not to use it as we want to stick to the original algorithm as much as possible. This lack of improvement is because only a local copy is moved from the critical path and implies an additional remote copy once the user has been notified.

Besides caching algorithms, we have also implemented several prefetching policies as explained in the above subsection. As we want to optimize the critical path of the read and write operations we give them higher priority over prefetching operations. A block prefetching will only be started if the disk is idle, otherwise no blocks will be prefetched. This will avoid read or write operations to wait for a prefetch block that may never be used.

8 Performance

8.1 Sprite Workload

In order to get the results in this paper we have used some parts of the Sprite workload, described in detail by Baker et al. [BAKE91]. The Sprite user community included about 30 full time and 40 part time users of the system. These traces list the activity of 42 client machines and some server over two day period measured under the Sprite operating system. We have not used the trace on its full length in order to reduce the simulation time. A 10 hours period has been used. This time interval starts at 9:00 am and lasts until 7:00 pm. We believe that this period is significant enough as it takes all office hours allowing a mixture of highly loaded time intervals and not so busy ones.

All executions started with an empty cache making some more disk operations than the ones we would get with a warm cache. As the simulation time is big enough, this should not make a big difference compared to a warm cache start.

We used these traces as we believe that the *Unix-like* work parallel computers do is very similar to the one done in a net of workstations.

8.2 Performance Results

In this section we will compare our cache with several other algorithms. The first one will be a regular file system with no cache. This one is very important as it helps us to scale the results and see if a given increase or decrease in performance is significant enough. We will also compare with N-Chance Forwarding as one of the latest algorithms in the bibliography. Finally, we will also show the performance of the Greedy algorithm as it will help us to explain some gains or loses in performance compared to N-Chance Forwarding.

All figures show the average read/write time measured from the moment the client starts a port send until it receives the reply. All policies are evaluated for 1,2 and 4 links architectures.

When comparing the read and write graphs we have to take in account that the average size of both operations is not the same and no direct comparison can be made. While the first one is 5041 bytes, the second one is 8140 bytes.

It is usual to compare the cache performance using the One-Block-Ahead prefetching algorithm. We have simulated it and similar, or even worst, results than without prefetching were obtained as many blocks are being prefetched while requested. As no significant difference is obtained we will only show the results of the executions without prefetching. This will ease the task of interpreting the results obtained using the different algorithms.

In order to simplify the study of the figures presented in this paper we will start explaining the results obtained when using the ATM parameters, 1 link and no prefetching at all. Once we are done with this configuration we will go through the rest of graphs describing the differences we may find. In order to group the interpretations in a more comprehensive way we will focus on the read operation first and then the write operations will follow.

In figure 2, the average read time with the above parameters is represented. The first important thing to notice is that no matter which caching algorithm is used, a significant increase in performance is obtained if com-

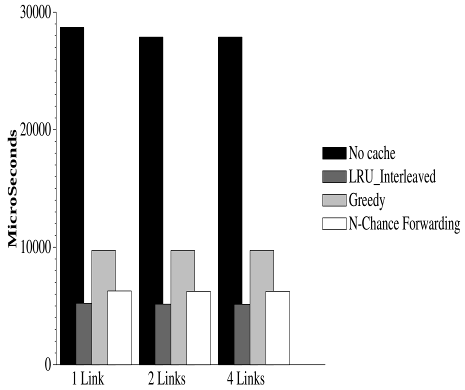

Figure 2: Average READ time under the ATM parameters. These times were taken with NO-PREFETCHING. The average read size was 5041 bytes.

Prefetching Policy	Local Hit %	Remote Hit %	Total Hit %
LRU NP	1.7	85.0	86.7
LRU FFOO	1.7	90.9	92.6
N-CHA NP	59.5	24.3	83.8
N-CHA FFOO	61.3	22.5	84.5

Table 2: Local, remote and global READ HIT ratios obtained by LRU-Interleaved and N-Chance Forwarding with No-Prefetching (NP) and Full-File-On-Open (FFOO). Only real hits are included, semi hits are not shown in this table. These ratios have been taken from the ATM parameters and 1 link runs. There is no significative difference if other parameters are used.

pared to no caching. This proves, once more, the importance of caching on a *Unix-like* environment.

We can also see that the average read time using the LRU-Interleaved caching distribution is faster that the one obtained by N-Chance Forwarding (around a 16.5% better). This may seem strange as most of the cache hits obtained by the second one are local while the ones obtained with the first algorithm are remote. This would lead to a better N-Chance Forwarding performance if similar global hit ratios were obtained but, as shown in table 2, no such thing happens. The distribution algorithm we propose accomplishes a relatively higher hit ratio. As a miss on a read block is much more expensive than a local hit (around 40 times slower) a few misses can increase the overall read time increasing the application execution time. The reason behind this difference in the number of hits is due to the better cache utilization made by our algorithm. While LRU-Interleaved uses the full cache as no replication is allowed, N-Chance Forwarding may waste some of the caching space with replication. Lets explain what happens using a small and simplified example. Suppose we have four machines where the first two have no replicated blocks while the other ones do have some replication among them. If the first machine has to forward a block to the second one a non replicated block will have to leave the cache. This unreplicated block should not have left the cache as other replicated blocks are in the cache. This scenario explains the difference in global read hit ratio that allows LRU-Interleaved to implement faster reads than N-Chance Forwarding.

The above scenario becomes much more important when the Full-File-On-Open prefetching algorithm is used. If some nodes are very active in I/O they will have many prefetching blocks coming to their portion of the cache. These new blocks will forward other local blocks in order to make space for themselves. As the forwarding of blocks increases, the chances of falling on a processor with no duplicated blocks is also incremented. Thus, the probability of the above scenario to happen will also be much higher than with no prefetching. In table 2 we can observe that the hit ratio difference between N-Chance Forwarding and LRU-Interleaved becomes much more important than when no prefetching was done. This, of course, leads to a much better performance of the read operations when our caching algorithm is used (around 32% faster reads). This difference can be seen in figure 3.

All measures presented by Dahlin et al. in their paper were measured with 16 Mbytes local caches instead of 1 Mbyte ones. As we wanted to check if the same problem appeared with bigger local caches we run some simulations modifying the size of the local caches (figure 4). In the results we obtained the above situation decreased quite a bit as less blocks were forwarded. In any case, the problem still exists and gave our algorithm around a 5% gain. Another interesting result obtained when different local cache sizes were used is that LRU-Interleaved can achieve similar hit

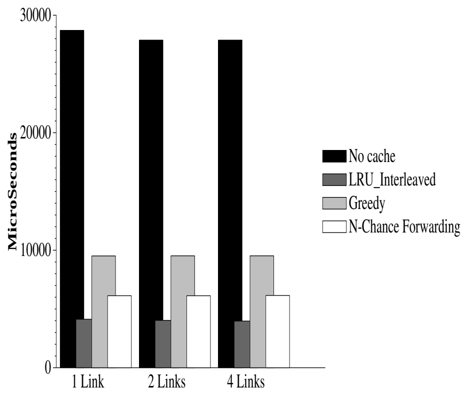

Figure 3: Average READ time under the ATM parameters. These times were taken with FULL-FILE-ON-OPEN prefetching policy. The average read size was 5041 bytes.

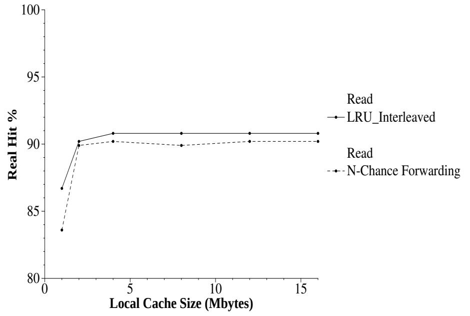

Figure 4: Influence that the size of local caches has on the overall READ HIT RATIO. These measures have been taken under ATM, 1 link and no prefetching parameters.

rations than N-Chance Forwarding using less caching memory.

If we reexamine figures 2 and 3 we can take a look at the influence the number of links has on the performance. We observe that no significant difference appears. This means that with such a fast interconnection network, the possibility of concurrent transfers does not result in better performance. Although we have simulated 4 links configurations with the ATM parameters, we should keep in mind that it is an unreal configuration. As the local memory is only twice faster than the interconnection network, we could never have four remote transferences done in parallel. The simulation was done in order to study the benefits we could obtain if such configuration was possible.

The simulation results of read operations using both algorithms under the PAROS parameters (figures 5 and 6) are very similar

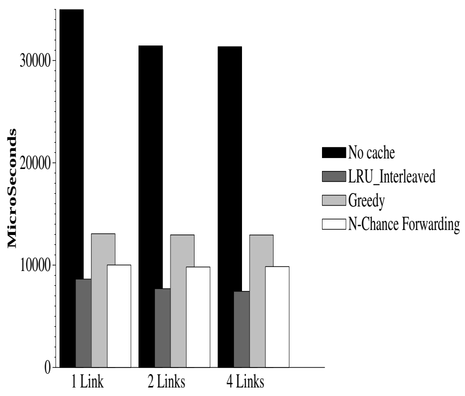

Figure 5: Average READ time under the PAROS parameters. These times were taken with NO-PREFETCHING. The average read size was 5041 bytes.

to the ones under the ATM. The main difference is that under this configuration all reads are slower as the interconnection network is also slower. We can also observe that the number of links becomes important to the LRU-Interleaved algorithm. As a remote copy is much more expensive, the parallelism we achieve starts to be significant allowing better read times. As most read operations are small in size we hardly take advantage of the 4 links and only 2 are really used.

Lets now start to examine the graphs where the average write time is displayed. In this part we will proceed in the same manner as we did with the read operations. In all the graphs where average write time is shown, a very high improvement compared to no caching is observed. This behavior is very normal as writing a block will, in most cases, only mean one or two memory copies. A disk access will only happen when a portion of a not cached block has to be modified. This does not occur very often as most blocks are new on the write operation or the block has already been cached.

In figure 7 we can see that once more the LRU-Interleaved algorithm behaves better than the N-Chance Forwarding. In order to explain this, somehow, unexpected result we should also explain why the Greedy algorithm is also better than N-Chance Forwarding. The difference between them is that no block forwardings are done by Greedy. If no forwardings are done write operations are faster because the overhead produced by forwarding blocks is avoided and because a smaller remote

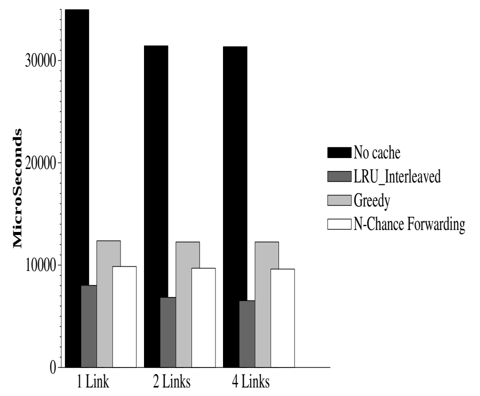

Figure 6: Average READ time under the PAROS parameters. These times were taken with FULL-FILE-ON-OPEN prefetching policy. The average read size was 5041 bytes.

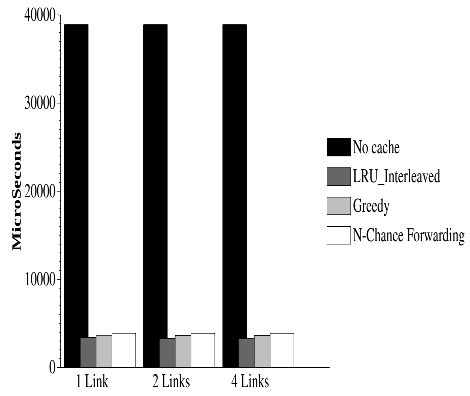

Figure 8: Average WRITE time under the ATM parameters. These times were taken with FULL-FILE-ON-OPEN prefetching policy. The average read size was 8140 bytes.

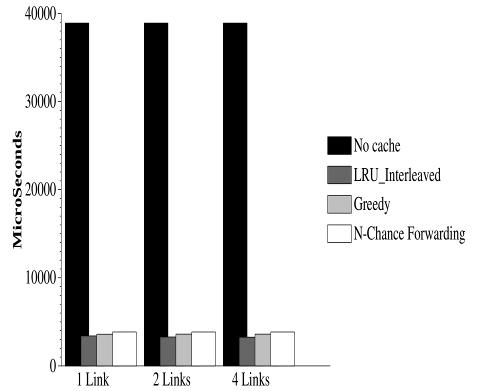

Figure 7: Average WRITE time under the ATM parameters. These times were taken with NO-PREFETCHING. The average read size was 8140 bytes.

hit ratio is obtained. A miss on clean is faster than a remote hit because only a local memory copy has to be done while on a remote hit we have to bring the block from the remote cache first. Besides, on a remote hit we have to invalidate the remote block. This extra overhead produced by forwarding blocks on remote hits overweights the benefits obtained by all the local operations when compared to LRU-Interleaved.

The difference between our proposal and the Greedy algorithm is due to the dirty block cleaning policy. When a dirty block has to be forwarded it is cleaned before being sent to a remote peer [DAHL95]. Although both operations are done asynchronously some overhead still exists. Nodes with many write operations may tend to forward lots of dirty blocks before the syncer gets a chance to clean them. This will add some overhead to the critical path of the operation that finds the dirty block. In LRU-Interleaved, blocks will have time to be cleaned by the syncer before the are needed again. This happens because it behaves nearly as a global LRU. The overhead produced by a dirty block makes our algorithm around a 10% faster than N-Chance Forwarding.

When using any kind of prefetching no real improvements are obtained in either algorithms as far as writes are concerned (figure 8). This happens because there is no difference whether the block is placed on a clean cache block or on a previously local cached portion. This is true because most files open to be written are either new or truncated. This avoids having to read the block from disk before modifying it and thus makes prefetching quite useless.

As can be deduced from figures 7 and 8, the number of links does not make a significant difference in either algorithm. The reason is the same as in the read operations.

On the executions run with the PAROS parameters (figures 9 and 10), N-Chance forwarding behaves better than LRU-Interleaved. This is because the overhead introduced by forwardings is not enough to overweight the expenses of a remote hit. Remote hits are much more expensive as the interconnection network is 10 times slower than a local memory copy.

A way to solve this problem is to parallelize

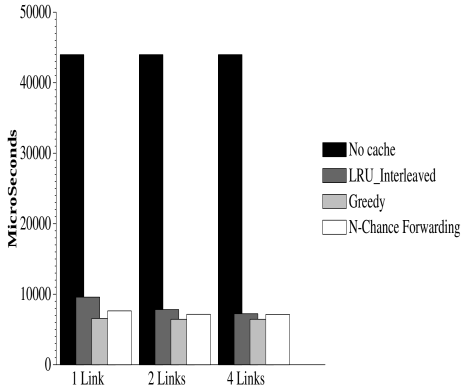

Figure 9: Average WRITE time under the PAROS parameters. These times were taken with NO-PREFETCHING. The average read size was 8140 bytes.

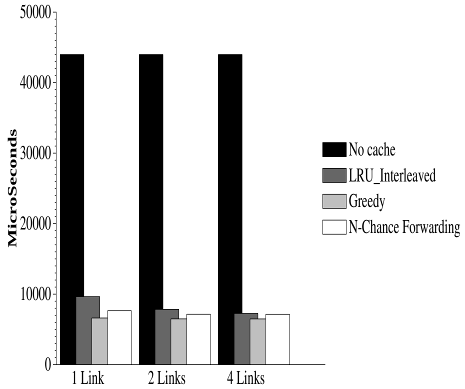

Figure 10: Average WRITE time under the PAROS parameters. These times were taken with FULL-FILE-ON-OPEN prefetching policy. The average read size was 8140 bytes.

the data transferences from the user to the cache. If enough memory copies are done in parallel we can hide the difference between a local and a remote copy. This can be observed as the number of links increases. In our case, 4 links would be enough to have the same average write time. When using the PAROS parameters we can have 4 links as the local memory is faster than four times the interconnection network.

This parallel transferences are not implemented in N-Chance Forwarding as they would make no sense. All writes are done in the local cache and we cannot make parallel copies within a local memory.

The last important issue is to prove that this centralized control is no bottleneck on either

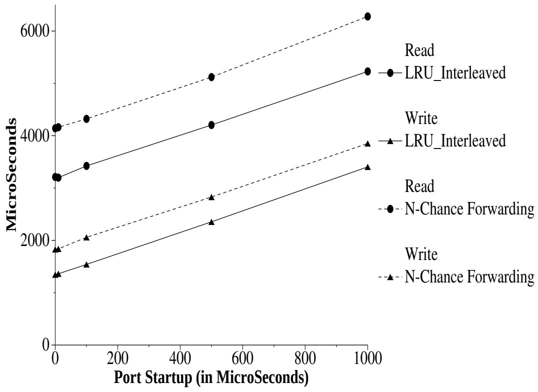

Figure 11: Influence the PORT STARTUP has on read and write operations. All runs were executed under ATM, 1 link and No-Prefetching configuration.

algorithm. First of all, we measured the average time for a local hit, a remote hit, a miss, and so on. We also calculated the theoretical times it should take a block operation if no other operations were being processed. After comparing these times no significant difference was found. This implies that no bottleneck was found either in the disk, centralized control nor interconnection network. This was true for most configurations but the PAROS parameters and 1 link. In this last case the interconnection network became an appreciable bottleneck. This results apply to both LRU-Interleaved and N-Chance Forwarding algorithms.

8.3 Startup Influence

As we chosed the startups measured in the PAROS microkernel operating system they could have not been the appropriate ones. In order to check the validity of those startups, we studied the influence they have on the file system operations. All measures in this subsection were taken under the ATM, 1 link and No-Prefetch configuration. This parameters were chosen as the executions under them will be most affected by the startup time. This happens because data transferences are very fast and no parallelism can be obtained, thus the startup overhead becomes more significant. We will only concentrate on the LRU-Interleaved and N-Chance Forwarding algorithms as they are the two main algorithms compared in this paper.

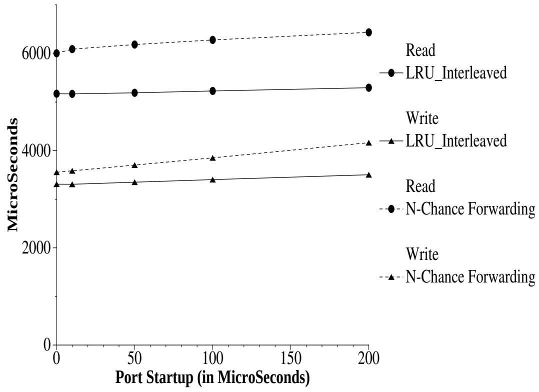

Figure 12: Influence the *MEMORY_COPY* STARTUP has on read and write operations. All runs were executed under ATM, 1 link and No-Prefetching configuration.

First, we will examine the influence port startups have on both read and write operations (figure 11). A linear increase in the average operation time can be observed. This is because each file system operation has two port sends (request and reply). If both communications are increased the final operation time will also be increased the same amount of time.

This, of course, will not be so in the distributed version. As many more communication through ports will be done, the port startup will have a higher influence.

A second startup we should take care of, is the one done in a memory copy operation. In figure 12, we can see the influence this startup has on read and write operations. As can be observed, this startup does not really affect the read operations as they are not bound to the memory copy time. On the other hand, the write operation is much more affected as most operations are just one or two memory copies. A higher influence can be observed in the N-Chance Forwarding algorithm as most copies are local and a modification in the startup affect more a fast copy that a slower one.

A conclusion we can obtain from this subsection is that if different startups had been chosen no big differences would have been obtained. Thus, the results presented in this paper are still valid even if the startups taken might have been a bit too high.

9 Future Work

As the results are encouraging enough, we believe that this file server should become fully scalable and thus we are studying distributing it. We are working on ways to do it without loosing the simplicity we have achieved and trying to avoid as much communication between servers as possible.

A second step, which may be somehow related to the first one, is to try to merge the RAMA file system designed by Miller and Katz [MILL95] with our proposed cache. We believe that both mechanisms, being very similar, should merge well together ending up with a very fast and scalable file system.

More measures have to be taken in the parallel applications area. We will test how the CHARISMA [KOTZ94a] [PURA94] applications work. This benchmark set is made of applications that rarely reuse already cached data. We will also test a *Europort* parallel out-of-core implementation of a finite element solver [AST95].

We also plan on working on prefetching policies that may increase the applications' performance. A first step will be to study how applications behave and try to find ways for the prefetcher to make better guesses. We will also try to obtain semantic information from mechanisms already implemented as Disk-Directed I/O [KOTZ94b] [KOTZ95]. This information should be able to hint the prefetcher in some way.

10 Conclusions

In this paper we have presented a distributed cache oriented file 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 before a fully scalable file system is obtained. Scalability up to 50 nodes has been demonstrated and work is being done in order to beyond that number.

We have shown that a very simple algorithm obtains better read bandwidth and a similar write performance than more complex ones. Once the distributed version is implemented, we may obtain much better comparative write performance as no cache coherence algorithm has to be implemented. This will simplify the algorithm and will also avoid many invalidating messages.

The figures have shown some important aspects we should take in account when designing a distributed cache. First, if a good global cache utilization is done by no replicating blocks better performance can be obtained with smaller local caches. Second, remote write hits do not increase write performance. Another very significative aspect in the resulting performance is the moment where dirty blocks are cleaned. Finally, the the potential communication parallelism parallel machines have can be used to hide some of the overhead introduced by remote hits.

One last achievement was the Full-File-On-Open prefetching algorithm we propose. We have seen that this kind of aggressive prefetching may increase the hit ratio and thus decrease the average read time. We have also shown that prefetching very rarely increase the write bandwidth.

11 Acknowledgments

We owe special thanks to Mike D. Dahlin for answering all our questions about the way N-Chances Forwarding works. We are grateful to the people at Berkeley who gathered the Sprite traces that helped us to feed our simulator and get the results we present in this paper. We would also like to thank E. Markatos whose comments improved the contents of this paper.

References

[AST95] M.Ast, J. Labarta, H. Manz, et al. "A General Approach for an Automatic Parallelization Applied to the Finite Element Code PERMAS" HPCN 1995

[BAKE91] M.G. Baker, J.H. Hartman, M.D. Kupfer et al. "Measurements of a distributed File System" In Proc. Of the 13th SOSP, 1991, pp. 198-212

[BERR94] R. Berrendorf, H.C. Burg, et al. "Intel Paragon XP/S - Architecture, Software Environment, and Performace" Forschungsezentrum Julich GmbH, Zentralinstitut fur Angewandte Mathematik, KFA-ZAM-IB-9409

[BORD93] R.R. Bordawekar, J.M. del Rosario and Alok N. Choudhary "Design and Evaluation of Primitives for Parallel I/O" Proceedings of Supercomputing'93, pp. 452-461

[CORB93] P. Corbett, D. Feitelson and J. Prost and S. Johnson "Parallel access to files in the Vesta file system" Proceedings of Supercomputing '93, pp. 472-481

[CORB94] P. Corbett and D. Feitelson "Design and implementation of the Vesta parallel file system" Proceedings of the Scalable High-Performance Commputing Conference 1994, pp. 63-70

[DAHL94] 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

[DAHL95] Michael D. Dahlin Personal communication

[FRAN92] M.J. Franklin, M.J. Carey and M. Livny. "Global Memory Management in Cliente-Server DBMS Architectures" International Conference on Very Large Dara Bases, 1992. pp. 596-609

[HART92] J.H. Hartman and J.K. Ousterhout "Zebra: A Striped Network File System" USENIX File System Workshop 1992, pp. 71-78

[KOTZ91] D. Kotz "Prefetching and Caching Techniques in File Systems for MIMD Multiprocessors" PhD Thesis from Duke University, Department of Computer Science, 1991

[KOTZ94a] D. Kotz and N. Nieuwejaar "Dynamic File-Access Characteristics of a Production Parallel Scientific Workload" Supercomputing '94

[KOTZ94b] D. Kotz "Disk-directed I/O for MIMD Multiprocessors" OSDI'94, pp. 61-74

[KOTZ95] D. Kotz "Expanding the Potential for Disk-Directed I/O" Dartmouth Computer Science PCS-TR95-254, 1995

[LABA92] J. Labarta, J. Gimenez, C. Pujol, T. Jove and J.I. Navarro "PAROS: Operating System Kernel for Distributed Memory

Parallel" Parallel Computing and Transputer Applications. IOS Pres/ CIMNE, Barcelona 1992

[LABA94] J. Labarta, S. Girona, T. Cortes et al. "The Paros Operating System Microkernel" CEPBA Technical Report CEPBA/UPC Research Report No.RR-94/05

[LABA95] J. Labarta, S. Girona, V. Pillet, T. Cortes and J.M. Cela "A Parallel Program Development Environment" CEPBA Technical Report UPC-CEPBA-RR-95/02

[LEFF93] A. Leff, J.L. Wolf and P.S. Yu "Replication Algorithms in a Remote Caching Architecture" IEEE Transactions on Parallel and Distributed Systems, vol. 4, No. 11, 1993, pp. 1185-1204

[LENO90] D. Lenonsky, J. Laudon, K. Gharachorloo, A. Gupta and J. Hennesy. " The Directory-Based Cache Coherence Protocol for the DASM Multiprocessor" International Symposium on Computer Architecture, 1990, pp. 148-159

[LOVE93] S.J. Lo Verso, M. Isman, A. Nanopoulos et al. "sfs: A Parallel File System for the CM-5" USENIX Technical Conference, 1993

[MARK95] E.P. Markatos, G. Dramitinos and K. Papachristos "Implementation and Evaluation of a Remote Memory Pager" FORTH-ICS Technical Report TR-129, 1995

[MILL95] E.L. Miller and R.H. Katz "RAMA: Easy Access to High-Bandwidth Massively Parallel File System" USENIX Technical Conference, 1995

[MOYE94] S.A. Moyer and V.S. Sunderam "PIOUS: A Scalable Parallel I/O System for Distributed Computing Environment" SH-PCC, 1994, pp.71-79

[PURA94] A. Purakayastha, C.S. Ellis, D. Kotz et al. "Characterizing Parallel File-Access Patterns on a Large-Scale Multiprocessor" Duke University Technical Report CS-1994-33

[ROY93] P.J. Roy "UNIX File Access and Caching in a Multicomputer Environment" OSF RI Workshop, Grenoble Juny 1993

[RUEM92] C. Ruemmler and J. Wilkes "UNIX Disk Access Patterns" HP Laboratories Technical Report, HPL-92-152, 1992

[SATY90] M. Satyanarayanan, J.J Kistler, P. Kumar, et al. "Coda: A Highly Available File System for a Distributed Workstation Environment" IEEE Transactions on Computers, vol. 39, No. 4, 1990, pp. 447-459

[SMIT85] A.J. Smith "Disk cache - Miss Ration Analysis and Design Considerations" ACM Transactions on Computer Systems, vol. 3, No. 3, 1985, pp. 161-203