Design Issues of a Cooperative Cache with no Coherence Problems *

Toni Cortes Sergi Girona Jesús Labarta
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya - Barcelona
{toni, sergi, jesus}@ac.upc.es
http://www.ac.upc.es/hpc

Abstract

In this paper, we examine some of the important problems observed in the design of cooperative caches. Solutions to the coherence, load-balancing and fault-tolerance problems are presented. These solutions have been implemented as a part of PAFS, a parallel/distributed file system, and its performance has been compared to the one achieved by xFS. Using the comparison results, we have observed that the proposed ideas not only solve the main problems of cooperative caches, but also increase the overall system performance. Although the solutions presented in this paper were targeted to a parallel machine, reasonable good results have also been obtained for networks of workstations.

1 Introduction

There is a general trend to use inter-node cooperation for improving performance of parallel and distributed file systems. Cooperative caching is a good example of this concept [9]. In a cooperative cache, all nodes work together to build a global cache. This cooperation increases the cache size and the global-hit ratio, thus improving the file-system performance.

This cooperation between nodes raises some design problems that need to be addressed. The first problem is the cache-coherence. We have to find a mechanism to allow good cooperation and still keep the data coherent. Second, we also need to find a way to balance the load between the nodes if an efficient cooperative cache is to be obtained. Finally, a fault-tolerance mechanism is also needed. In general, a single node failure should not result in the whole system failure.

In this paper, we study different approaches to solve the afore-mentioned problems. We propose a cache that avoids the cache-coherence problem by avoiding replication. We also present several ways to distribute the cache buffers among the servers to make them able to adapt to load variations. And finally, we propose a fault-tolerance mechanism based on parity buffers. Furthermore, we will also propose three tolerance levels so that the most appropriate one can be used in each configuration.

All the performance results presented in this paper were measured through simulation. Simulations were used so that a wide range of environments and architectures may be studied. These results were compared against the ones obtained by the algorithms proposed in xFS, the file system designed as part of the NOW project [2, 1, 9]. We ran all the simulations using the CHARISMA [21, 14] and Sprite [3] trace files.

This work has been developed as a part of PAFS [6, 4], a parallel/distributed file system specially designed to work on a parallel machine.

This paper is structured into 7 sections. After this introduction, Section 2 describes the main aspects of PAFS and Section 3 explains the three design issues presented in this paper: cache-coherence, load-balancing and fault-tolerance mechanisms. Section 4 discusses the most important differences between PAFS and xFS, the two file systems compared in this paper. In Section 5, we describe the simulator and the trace files used to get the performance results presented in Section 6. Finally, we present the conclusions in Section 7.

1.1 Target Environment

PAFS is a parallel/distributed file system designed to run on a parallel machine. We assume that this parallel machine has a large number of nodes connected through a fast interconnection network. We also assume that each node may have none, one, or even several disks connected to it. Although PAFS was designed to run on a parallel machine (PM), we also show that it can perform reasonably well on a network of workstations (NOW).

Each node runs a micro-kernel based operating system instead of a monolithic one. All functions not offered by the kernel itself are implemented by out-of-kernel servers. This is also the case for the file-system operations. Besides the typical micro-kernel abstractions, a *memory_copy* operation is desirable in order to implement PAFS efficiently.

The *memory_copy* is used to transfer data between the cache and the user. Our assumption is that any processor can set up a data transfer between any other two processors. We will refer to this operation as *remote copy* when

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

IOPADS 97 San Jose CA USA

the data is copied from one node to a different one. On the other hand, a *local copy* occurs when the information is moved within the same node. Similar remote-memory-access mechanisms are supported in a variety of distributed-memory systems [7, 15, 25]. Furthermore, most current parallel machines offer very fast mechanisms to copy memory blocks from one node to another [12, 13].

1.2 Terminology

In this subsection, we introduce basic concepts and present terminology used in this paper.

As in traditional caches, requesting a file block means ending up with either a cache hit or a cache miss. The difference, in a cooperative-cache environment, is that a cache hit may be either local or remote. If the requested file block is found in the same node as the requesting client is running we have a *local hit*. On the other hand, if the requested file block is found in the cache of a different node, we have a *remote hit*. We also use the term *global hit* to identify both kinds of hits.

It is also important to differentiate between the possible situations that may be found on a cache miss. The first possibility is that the file block that has to be replaced has been modified but written to disk yet (i.e., it is dirty). We call to this situation a *miss on dirty*. On the other hand, we will refer as a *miss on clean* to those *misses* in which the replaced block does not need to be written to disk (i.e., it is clean).

1.3 Related Work

A large number of parallel and distributed file systems have been proposed in the research community. Among all of them, xFS [2, 1, 9] is of special interest to our work as it was the first one to include the idea of cooperative caching. Furthermore, we have used it to compare the performance obtained by the cooperative cache built in PAFS. The most important differences between both file systems will be presented, in detail, in a later section (§4).

Sarkar and Hartman have proposed a way to decrease the load of the managers in cooperative caching [24]. In their work, clients are allowed to make local decisions about the location of a block based on hints. These local decisions decrease the number of times clients request information from managers.

Leff et al. have theoretically analyzed the effects of remote memory on cache activities [17, 18]. In their latest work, they also propose a mechanism which avoids the cache-coherence problem. The main difference with our proposal is that our servers do not need to communicate among themselves to know the placement of a given block. Furthermore, only a cooperative-replacement algorithm was presented in their work, while we present a complete file system with a fault-tolerance mechanism. Finally, in our work all measurements are done using real workloads.

There has also been interesting research in cooperative memory usage [10, 19, 20]. This kind of cooperation has also been studied in the database field by Franklin et al. [11].

A centralized version of this work was also developed by this research group [5, 22].

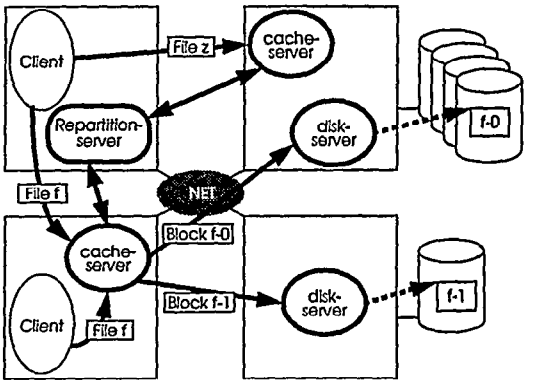

Figure 1: PAFS architecture.

2 PAFS overview

2.1 File-system Architecture

When designing a file system, there are two main issues that have to be taken into account: how the data is distributed among the disks and how this data and meta-data are managed by the servers before they get to the clients. In this work, we only examine the second point as the ideas we present are valid no matter how data is stored on the disks.

To run PAFS we need, at least, three sets of servers: disk-servers, cache-servers and a repartition-server (Figure 1). A fourth set of servers, the parity-servers, is only needed whenever the fault-tolerance mechanism is activated.

Cache-servers are in charge of serving the clients requests. They manage the cache and meta-data information. If the data needed by a cache-server is not in memory, and has to be fetched from disk, this information is requested from a disk-server. These disk-servers are processes responsible for physically reading and writing the disk blocks.

To implement a highly scalable system, the inter-server communication has to be as low as possible. For this reason, we propose a load distribution that does not need any communication between cache-servers. Each cache-server is responsible for a set of files. It keeps all the information needed to find a file block in the cache, or the disk, without the need of any other cache-server. Clients know which servers are in charge of which files computing a hash function using the file name (or file-ID).

The entire global cache is distributed among the cache-servers. All the buffers assigned to a cache-server build this server's partition. When a buffer is assigned to a given partition, the cache-server which owns this partition is the only one allowed to use it. As the load of all the cache-servers may not always be balanced, there has to be a mechanism to dynamically reassign buffers among them. The repartition-server is in charge of such redistribution. Different policies to distribute buffers are presented later in this paper (§3.2).

Finally, another important issue is to offer a fault-tolerance mechanism. The fault-tolerance mechanism built on PAFS is based on parity buffers in a similar way to the one found in a RAID level 5 [23]. The parity-servers are the processes responsible for such mechanism. More information on the way this works is also presented in a later section (§3.3).

A very important issue in the design of PAFS was the lack of dedicated nodes. All servers may run in any node and share the node with user applications without interfer-

ing too much in their performance. The only exception are the disk-servers that have to run on the nodes where disks are connected, but they can also share the CPU with other applications.

As this file system was designed to study the behavior of a cooperative cache, we have not placed much effort in designing a placement algorithms to distribute file blocks among the disks. We have used a round-robin algorithm. This may not be the best way to distribute blocks in the disks, but it was good enough for our purposes. Furthermore, should any other distribution mechanisms have been used, the same basic conclusions could have been made.

To increase the performance of the write operations, PAFS uses a delayed-write policy. Each cache-server has a thread named syncer that wakes up every 30 seconds. Once awaken, it searches for all dirty file blocks and updates them in the disk.

2.2 Cooperative Cache

The cooperative cache is the most important part of this file system. We have designed a cache that has the advantages of cooperation and avoids the problems derived from the coherence mechanism.

Each cache-server is in charge of a set of files and a set of cache buffers distributed among all the nodes. When a new block enters the cache, the cache-server in charge of this block has to discard another block from its partition. It has to be from its partition as we want to avoid communication between servers as much as possible to achieve high scalability. In order to choose a buffer to place this block into, the PG-LRU (Pseudo-Global LRU) replacement algorithm is used. This algorithm is very similar to the well-known LRU but modified to achieve some data locality. To implement PG-LRU we have to define the concept of queue-tip. A partition queue-tip is a set of buffers made by $X\%$ of the least-recently-used ones in that partition. Instead of replacing the least-recently-used block, one of the blocks in the queue-tip is discarded using the following algorithm. If there is a buffer in the queue-tip located in the same node as the client, this buffer is replaced. Otherwise, the least-recently-used buffer is chosen regardless of the node into which it is located.

A study on the impact the queue-tip size has on the overall performance was presented in an earlier work [6]. From that work, we learned that queue-tips larger than 5% of the partition size do not increase the overall system performance. This study fixed the size of the queue-tip used in this paper to 5% of the partition size. Anyway, this parameter should be tuned by the system administrator to adapt to the needs of the systems.

3 Design Issues

3.1 Cache Coherence

The cache-coherence problem arises when replication is allowed and the system maintains several copies of the same data. This replication is mainly done to increase the *local-hit* ratio of the global cache.

One of the main ideas presented in this paper is that exploiting the local-hit ratio is not the only way to achieve a high-performance cooperative cache. This leads us to believe that we can avoid replication as its main objective is not a key issue in our design. If a block is already found in the global cache, it is sent to the user address space, but no replication on the client's cache is performed. If there is no replication, no cache-coherence problems may appear and we can get rid of all the coherence mechanisms. This simplifies the cache and file-system design very much. In the performance section (§6), we will show that avoiding the coherence problem not only simplifies the design, but also increases the file-system performance.

3.2 Repartition Policies

In this section, we describe the mechanism proposed to distribute the buffers among the cache-servers in order to adapt to the changing needs of the system. Furthermore, we also present some policies that can be used with this mechanism and which will be evaluated in the performance section(§6).

The mechanism we propose consists of a periodic redistribution of the buffers among the cache-servers. The repartition-server, which is in charge of such redistribution, periodically asks for the working-set from each cache-server. We define the working-set of a server as the number of different file blocks that have been accessed during a redistribution interval. Using this information, and the current size of each cache-server partition, the repartition-server redistributes the buffers proportionally to each server's working-set. Once the repartition-server has decided which blocks belong to which partitions, it sends a message to all the cache-servers informing about the composition of their new partition.

It is important to clarify that changing the owner of a buffer (i.e., changing the partition it belongs to) does not mean that the buffer is moved or copied to another node. It only means that a different cache-server will be able to use it. As we have already said, a process may program copies of memory blocks from any node to any other node using the remote-memory copy mechanism. Only the owner of a buffer will be allowed to copy to/from the buffers in its partition.

It is also important to notice that once a buffer changes the partition it belongs to, it also loses all the information kept in it. This happens because the new cache-server does not know how to handle the files of any other cache-server. This also means that if the buffer is dirty, it will have to be sent to disk before the re-assignation is done.

The repartition policies presented in this paper will be based on a combination of the following concepts. First, the number of buffers that change ownership may, or may not, be limited in order to avoid drastic changes in the partitions size. And second, the reasignation of buffers may be done eagerly or lazily.

Drastic changes in the size of the partitions may lead to bad performance results. To avoid these dramatic changes, we propose to limit the number of buffers a cache-server may gain or loose in each redistribution. Limiting the buffers a cache-serve may lose should be a function of the number of buffers it has in its partition. On the other hand, the limitation on the buffers a cache-server may gain should be a function of the number of buffers it will be able to use. We propose that the maximum number of buffers that a cache-server may loose in each repartition should equal a predefined percentage of its partition size. This percentage may be tuned by the system administrator. In all the simulations presented in this work, we only allowed cache-servers to lose 10% of their blocks. This percentage was shown to be an appropriate one in a previous work [6]. On the other

hand, we also have to find a number that gives an idea of the number of new buffers that can be used between repartitions. We believe that the number of blocks that can be physically read in a repartition interval gives this bound. For this reason, limiting the number of gained buffers to the maximum number of blocks that can be physically read in a repartition interval seems an interesting idea.

The instant when buffers change their ownership may also have a significant impact on the effectiveness of the repartition algorithm. One possibility is to implement an eager algorithm that changes the ownership of the buffers as soon as all nodes have been notified. This eager algorithm has the problem that many reassigned buffers may be used in the last portion of the interval. It may even happen that they are never used by the new cache-server. During all this time of inactivity, they might have been used by the old server increasing the system performance. To solve this problem, a lazy algorithm may be implemented. The repartition-algorithm may send the number of buffers from each node that a given cache-server has gained instead of the identifier of these buffers. Then, when the cache-server needs a new buffer, it requests a buffer from one of the servers in its list. This allows the buffers to be used by their old owner until they are really needed. It also allows the old owner to decide which buffer to discard at the moment the buffer is really to be discarded. To implement this request out of the critical path of the read/write operations, a deferred mechanism is implemented. Once a cache-server gets the list, it requests the first buffer in the list. Once this buffer is used, another one is requested on the spot. This continues until no more buffers can be requested. With this mechanisms, at most one buffer changes ownership without being used.

The first repartition policy we propose has been named *Not-limited*. In this policy, buffers change ownership using the eager mechanism and only the number of buffers a cache server may lose is limited.

Limited is the second proposal for the repartition policy. This policy is quite similar the previous one. The only difference is that the number buffers a cache-server may gain is limited by the number of new blocs that can be physically read from disk.

Finally, *Lazy-and-limited* changes the eager repartition mechanism used in the *Limited* algorithm and replaces it by the lazy one.

In order to see whether the dynamic redistributions is really needed, we will also evaluate a *Fixed-partition* policy where the buffers are assigned at boot time.

3.3 Fault Tolerance

As the cached information is scattered among all the nodes, we need to provide the system with some kind of fault-tolerance. The whole system should be able to continue even if a node fails.

We propose a fault-tolerance mechanism based on parity lines and parity buffers quite similar to the one used in a RAID level 5 [23]. We define a parity line as a set of cache buffers where none of them is located in the same node. All the buffers in the line are used to keep file blocks but one, the parity buffer. The idea is to keep the XOR of all the dirty buffers from a line in its parity buffer. Thus not all buffers in the line affect the parity buffer. Only those buffers that have been modified and still have not been updated in the disk are kept in the parity buffer. Should one of these dirty blocks be lost, the information kept in its parity buffer could be used to restore the last version of the lost buffer.

The parity-servers are the processes responsible for all the fault tolerance mechanism and cooperate with the cache-server to keep the parity information up to date. For performance reasons, we have one of these servers in each node where a parity-buffer is located. This means that a single parity-server can handle more than one line. Anyway, if for scalability reasons, each line has to have a dedicated parity-server, this could also be done with no modifications in the system.

The protocol used between cache-servers and parity-servers is described as follows. In a write operation, and before the buffer is modified, the cache-server sends a message to the parity-server informing that the buffer will be modified. Afterwards, the parity-server copies the unmodified buffer to a local buffer and informs the cache-server that the buffer can be modified. Then, the cache-server modifies the buffer as requested by the client. At the same time, the parity-server copies the modifications from the client in order to keep the parity buffer up to date. Once the modifications are included in the parity buffer, the parity-server informs the cache-server. Finally, the cache-server can inform the user about the end of the write operation. Once a buffer is sent to the disk, the parity-server is also informed and removes the information of this buffer from the parity buffer. All the messages that go to and from the parity-server are sent though ports while all the data copies are done using the memory-copy mechanism.

When a node fails, the parity-servers can rebuild the modified file blocks that have been lost and can write them to disk. Once all the lost file blocks are in the disk, the file system continues working normally but for a smaller global cache. If the parity-server is the one that fails, we may rebuild the parity buffers from the information kept by all the cache-servers.

An important issue is that not all systems need the same level of fault tolerance. While in some environments fault tolerance is crucial, there are other environments where performance is much more important than fault tolerance. If PAFS is to be useful in most environments, a set of fault-tolerance levels has to be offered.

As there are environments were there is no need of fault-tolerance mechanisms, or where speed is more important than fault tolerance, PAFS offers as the first level, or level A, a non fault-tolerant level.

The second level, or level B, is the one proposed by Anderson et al. in xFS [1, 9]. The assumption in this fault-tolerance level is that a file block, when cached in the same node as the client which requested it, may be lost if the caching node fails. This comes from the assumption that if this node fails, all its applications will also fail and no one will care for the last actualization of this block. Thus only blocks placed in a remote cache are guaranteed to be in the disk (or are recoverable through the parity buffer).

The previous level might not be sufficient in an environment with a lot of file sharing between applications. It would also not be sufficient in an environment where a lot of communication between applications is done through files. Let's examine the following example. Suppose we have a parallel make which has to compile two files and link them afterwards. As we have a parallel environment, both compilations are run in parallel. Some of the blocks of the object files are kept in the cache where each compilation is being made and thus are not sent to the disk (or parity buffer).

If one of the nodes fails after the compilation is done, the linker will not be able to make the executable file. The linker will have no problem with one of the files, but the file that was compiled in the failed node will be corrupted as some blocks have been lost [8]. This may not be acceptable in some environments and a higher degree of fault tolerance has to be proposed. To solve this problem PAFS offers the last fault-tolerance level: level C. In this level, all modified blocks are sent to the disk (or parity-server) offering a higher degree of fault tolerance.

4 Algorithm Comparison

To test the performance obtained by the mechanisms presented in this paper, we compare PAFS (with these mechanisms included) with xFS [2, 1, 9]. This file system was chosen because it has one of the latest, and best, cooperative-cache algorithms found in the literature (N-Chance Forwarding). The most relevant differences between both systems are explained in this section.

The first significant difference is that xFS encourages *local hits* by placing new blocks in the client's local cache and making a local copy of the blocks found in a remote cache. PAFS, on the other hand, does not try to increase the number of *local hits*. It tries to speed up *remote hits* to make less significant the *local-hit* ratio.

A second difference is the way the coherence problem is approached. While PAFS avoids it by simply avoiding replications, xFS implements a token-based-on-a-per-block-basis cache consistency scheme.

In PAFS, once a block is placed in a node, it remains there until it is discarded. On the other hand, xFS keeps moving blocks from one node to another for two basic reasons. The first one is to increase the *local-hit* ratio. The second one is to increase the life-time of a block. Once a block reaches the last position in the LRU list, it is forwarded to another node. After N jumps without being accessed, the block is finally discarded.

Another difference between both file systems is the environment they are designed to run on. PAFS is designed to work on a parallel machine (or NOW) where each node runs a micro-kernel based operating system. All services, including the file system, have to be implemented by out-of-kernel servers instead of by the kernel itself. On the other hand, xFS works on a monolithic operating system. The file-system code is implemented in the kernel and some operations can be handled locally without sending or receiving any messages. To study the behavior of xFS in our environment, we placed all file system functions originally implemented in the kernel in an out-of-kernel server leaving all the rest untouched. We also placed one such server in each node to minimize the impact of sending a message for each request. These modifications have been proven to be nearly irrelevant in the overall system performance [6, 4].

Finally, only fault-tolerance level B is implemented by xFS while levels A, B and C are offered by PAFS.

5 Simulator and Trace Files

5.1 Simulator

The file-system and cache simulator used in this project is part of DIMEMAS[1] [16] which reproduces the behavior of a distributed-memory parallel machine. This software not only simulates the machine and disk access, but also different short-term process-scheduling policies.

The whole simulator is a trace-driven one where traces contain CPU, communication and I/O demand sequences for every process instead of the absolute time for each event.

The communication and disk models implemented in the simulator are very important to understand the results presented later. All communications are divided into two parts: a startup, or latency, and a data-transfer rate, or bandwidth. The startup is constant for each type of communication (port or *memory_copy*) and it is assumed to require CPU activity. This startup is different whether the communication is within a node or it crosses the interconnection network. The data-transfer time is proportional to the size of the data sent and the interconnection-network bandwidth. The disk is also modeled by a latency and a bandwidth where the latency depends on the kind of operation (read or write).

Using this simulator we have implemented all the algorithms proposed as part of PAFS. We simulate all the servers, their communication, their CPU consumption and the interaction with the clients. We have also implemented the algorithms presented in xFS in order to make the comparison.

5.2 Trace Files and Simulation Parameters

For our experiments, we used two sets of traces: CHARISMA [21, 14] and Sprite [3]. The first one characterizes the behavior of a parallel machine while the second one characterizes the workload that may be found in a NOW.

The CHARISMA traces were taken from the Intel iPSC/860 at NASA Ames' Numerical Aerodynamics Simulations (NAS). This multiprocessor has 128 computational nodes, each with 8 MBytes of memory.

The information kept in this trace file only contains the operations that went through the Concurrent File System. This means that any I/O which was done through standard input and output or to the host file system was not recorded. The complete trace is about 156 hours long and was collected over a period of 3 weeks. To reduce the simulation time, we only used part of this trace file. We used a two-day period (February 15th and 16th) which was the busiest period of the trace. Incidentally, the first 10 hours were used to warm the cache as we want to study the permanent-state behavior.

The Sprite user community included about 30 full-time and 40 part-time users of the system. These traces list the activity of 48 client machines and some servers over a two day period measured in the Sprite operating system.

Although the trace is two days long, all measurements presented in this paper are taken from the 15th hour to the 48th hour. This is done because we used the first fifteen hours to warm the cache.

Table 1 presents the number of requested operations, accessed blocks and average operation sizes performed during the simulated period. This table is meant to help the reader

[1] DIMEMAS is a performance-prediction simulator developed by CEPBA-UPC and is available as a PALLAS GmbH product.

	CHARISMA		
	operations (in thousands)	8KB blocks	Average size (in Bytes)
Read	812	3493	32475
Write	1987	4653	14734
Open	5	-	-
Close	5	-	-
Unlink	.5	-	-
Total	2809.5	8146	-
	Sprite		
	Operations (in thousands)	8KB blocks	Average size (in Bytes)
Read	519	650	4657
Write	192	305	8000
Open	407	-	-
Close	400	-	-
Unlink	47	-	-
Total	1565	955	-

Table 1: Trace file comparison.

	Parallel Machine	NOW
Nodes	128	50
Local-Cache Size	1 MB	16 MB
Buffer Size	8 KB	8 KB
Memory Bandwidth	500 MB/s	40 MB/s
Network Bandwidth	200 MB/s	19.4 MB/s
Local-Port Startup	2 μs	50 μs
Remote-Port Startup	10 μs	100 μs
Local *Memory_copy* Startup	1 μs	25 μs
Remote *Memory_copy* Startup	5 μs	50 μs
Number of Disks	16	8
Disk-Block Size	8 KB	8 KB
Disk Bandwidth	10 MB/s	10 MB/s
Disk Read Seek	10.5 ms	10.5 ms
Disk Write Seek	12.5 ms	12.5 ms

Table 2: Simulation parameters.

to understand the real load placed on the file system. It will also be useful to see the differences between both workloads.

As we have two different workloads we also simulated two basic architectures. While the CHARISMA trace file has been tested on a parallel machine, the Sprite trace file has been simulated on a NOW similar to the one used by Dahlin et al. [9]. The parameters used to simulate both architectures are shown in Table 2.

6 Performance

This section is divided into two subsections. The first one describes the behavior of the CHARISMA trace file simulated on a parallel machine. The second one describes the behavior of the Sprite trace file on a NOW.

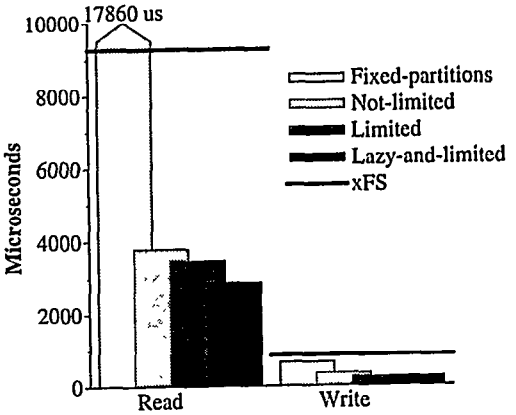

Figure 2: Read and write average times obtained with the reparation mechanisms (PM).

6.1 Parallel Machine (PM)

Repartition Mechanisms

Figure 2 shows the read and write average times obtained when using the proposed repartition mechanisms. The straight line represents the values obtained by xFS and the various versions of PAFS are represented by the solid bars.

The first thing that can be observed is that a *Fixed-partition* policy obtains a very poor read performance. This is because the workload is quite unbalanced at some points and *Fixed partitions* is not able to adapt to the changing needs of the system.

Another important result extracted from Figure 2 is that limiting the number of buffers that change ownership using a lazy algorithm is a good idea. When no limit is set, many buffers change ownership but are not used by their new owner as it does not have time to use them. If the block had remained with its original owner, its information might have been used to serve a client request.

We can also see that PAFS runs better than xFS independently of the dynamic repartition mechanism used. The reasons behind this behavior are explained in the following paragraphs.

First, the number of *global hits* is higher in PAFS (95%) than in xFS (92%). This difference is basically due to the effect of replication. As xFS wastes some of its buffers keeping replicated blocks, the cache can keep a fewer number of different blocks. Thus, it behaves as a smaller cache.

Second, *remote hits* take longer in xFS than in PAFS. A *remote hit* in PAFS only means a remote copy of the requested bytes from the cache to the user address space (1).

$$PAFS\_R\_hit = R\_copy(request\_size) \quad (1)$$

On the other hand, a *remote hit* in xFS has more things to do. As it wants to increase the *local-hit* ratio, the requested block is copied to the local cache and then is sent to the user address space (2). We should notice that the information that goes through the network in PAFS is the information requested by the user and in xFS is the whole block. Every *remote hit* also needs to contact the manager of the block to locate the remote copy (3). This communication will be done for each remote hit found in the request while in PAFS only one communication to a remote server is done no matter how many remote hits are found in the request. It

is also quite frequent to find that the buffer needed to place the requested block has to be forwarded to another node in order to increase its lifetime. The influence this fact has on the average *remote-hit* time can be observed in equation (4) where the time of forwarding a block is weighted by the probability of this happening (pf). Similarly, if the buffer to be used is dirty we will need to send it to disk before using it. The operations needed are the same as in a forwarding (5) but the probability is different (pd).

$$basic = R\_copy(block\_size) + L\_copy(request\_size) \quad (2)$$

$$manager = 2 * R\_port \quad (3)$$

$$forward = [2 * R\_port + R\_copy(block\_size)] * pf \quad (4)$$

$$dirty = [2 * R\_port + R\_copy(block\_size)] * pd \quad (5)$$

$$xFS\_R\_hit = basic + manager + forward + dirty \quad (6)$$

If we calculate the time spent in xFS (6) and in PAFS (1) using the probabilities found in the simulations (pf=.75 and pd=.51) we find that xFS takes around 5 times longer than PAFS in performing a *remote hit*. However, these calculations do not take into account the CPU consumed, nor the contention of the network, nor the time spent waiting for a server to become ready. If we take this time into account (as was done in the simulations), the ratio (i.e., 5) becomes even more. On the other hand, PAFS only has 3.3 *remote hits* for each one found in xFS. This means that the effort needed to increase the *local-hit* ratio is much higher than the benefits obtained.

Third, misses are also more time consuming in xFS than in PAFS. This also happens due to the forwarding of blocks and the cleaning of dirty blocks in the critical path of the operation.

When comparing the write performance obtained by PAFS and xFS we have to take an extra issue into account. In xFS, most of the write operations need to ask the ownership of the block to the manager. Furthermore, if more than one copy is kept in the cache, the manager has to invalidate them before grating the ownership to the requesting server. This extra work is not done in PAFS as no coherence mechanism is needed.

All these results prove that avoiding the coherence problem by avoiding replication not only simplifies the design but also improves the file-system performance.

Network-Bandwidth Influence

As PAFS does not exploit locality as much as xFS, it is very important to study the influence that the interconnection-network bandwidth has on the results presented earlier.

This study has been done varying the bandwidth of the interconnection network. Figure 3 presents the influence this variation has on the read (solid lines) and write (dashed lines) average times. On the X axis we represent the ratio between the local-memory bandwidth (L-BW) and the interconnection-network bandwidth (R-BW). We should notice that PAFS with *Fixed partitions* is not represented as it behaves much worse than xFS and thus it is of no interest to us anymore.

We can observe that the difference between the distribution algorithms is not affected by the interconnection network speed. It can also be observed that the difference between PAFS and xFS in the read operation remains the same independently of the interconnection-network bandwidth. This happens because the time gained by xFS due to its higher *local-hit* ratio is lost due to of its longer *remote-hit* duration.

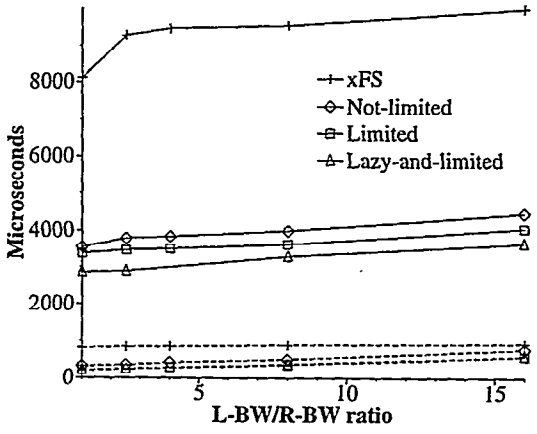

Figure 3: Network-bandwidth influence (PM).

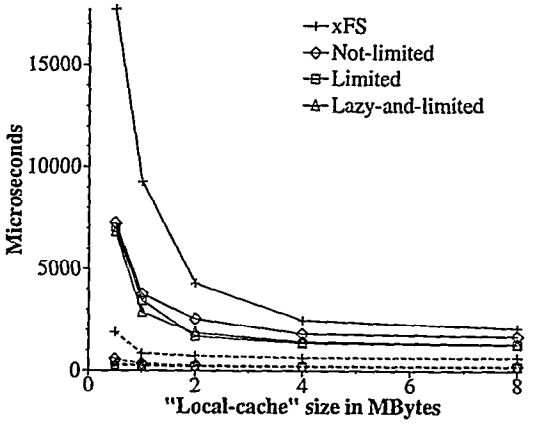

Figure 4: Local-cache size influence (PM).

Write operations behave in a different way. A write miss in PAFS is, nearly always, placed in a remote cache while it is always placed in the local cache by xFS. This means that a slow interconnection network has more influence on PAFS than on xFS. Thus, the slower the interconnection network is, the smaller the difference between both algorithms becomes.

Cache-Size Influence

Another important parameter that influences the cache performance is the size of each local cache. Figure 4 shows the read (solid lines) and write (dashed lines) average times obtained while varying the size of the local cache.

The first thing we observe is that *Limited* and *Lazy-and-limited* tend to obtain very similar results with large caches. This is because the reassigned buffers that are not used become of little importance.

We also observe that the smaller the cache is, the worse xFS behaves if compared to PAFS. The reason, as was explained earlier, is that all the replicated blocks use buffers that could keep different blocks. If the local caches are very

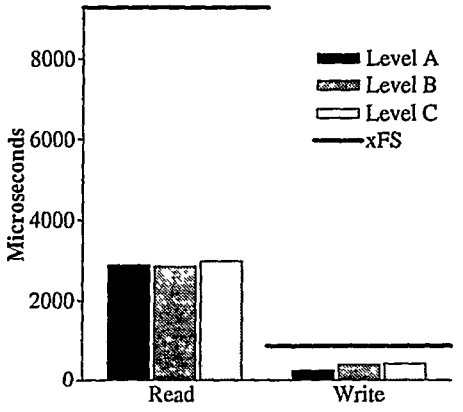

Figure 5: Fault-tolerance performance (PM).

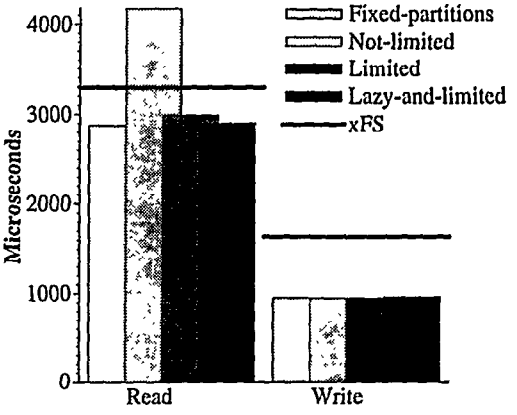

Figure 6: Read and write average times obtained with the repartition mechanisms (NOW).

small, this becomes a serious problem of xFS (as observed in Figure 4 when 512KByte and 1MByte local caches are used).

The final observation is that the cache size has little influence on the write performance, because a cache hit or a miss takes the same amount of of time as long as there is no need to read the block first. In both cases we need a copy from the user address space to the cache.

Fault-Tolerance Influence

Besides the coherence and repartition mechanisms, we also propose a fault-tolerance mechanism based on parity buffers. In Figure 5 we present the results obtained by this mechanism using the three different levels. This performance is compared to xFS which only implements level B. All three levels were tested with the *Lazy-and-limited* repartition algorithm as it has been shown to be the best one.

The most important information extracted from the above figure is that the cooperative algorithm implemented in PAFS behaves better than xFS even with a higher fault-tolerance level.

If we compare the performance of the three different levels we can see that levels B and C take longer performing the write operations than Level A. This happens because some more *memory_copy* operations have to be done per write in order to send the modifications to the parity-server. The little difference between levels B and C is due to the low *local-hit* ratio achieved by PAFS.

Read operations are only slightly affected as they do not take part in the parity mechanism. The only influence may be a higher resource utilization.

6.2 Network of Workstations (NOW)

Repartition Mechanisms

In this subsection, we present the read and write average times obtained by the different repartition mechanisms in a NOW (Figure 6). As can be seen, the behavior is quite similar to the one obtained in a a parallel machine but for *Fixed partitions* and *Not-limited*.

As the Sprite workload is quite balanced, the *Fixed-partition* policy behaves quite well. There is no need to redistribute buffers among the cache-servers. Actually, if we do so with no limit, we end up obtaining a quite bad performance. This happens because many blocks change ownership when this is not really needed and their information is lost for no good reason.

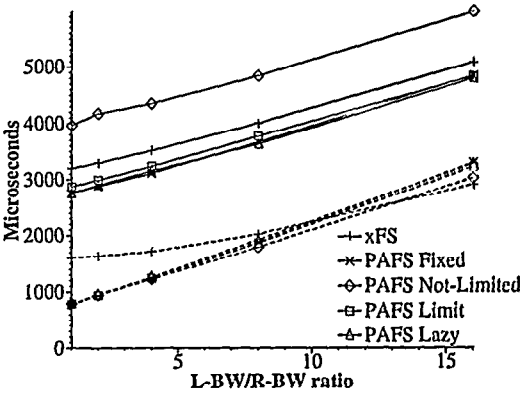

Figure 7: Network-bandwidth influence (NOW).

Network-Bandwidth Influence

The influence of the interconnection-network bandwidth (Figure 7) is also quite similar to the one found in a parallel machine. The most important difference is that with this environment and the Sprite workload we found the point where a write operation in PAFS is slower than in xFS. This happens when a remote copy is 10 times slower than a local one.

This intersection point has been found when a remote copy is 10 times slower than a local one due to several reasons. First, as the local-cache size used in this experiment is larger than the one used in the parallel machine, xFS has less *remote hits* and thus their extra overhead has less impact. Second, there is much more sharing in the CHARISMA trace file than in the Sprite one and the coherence mechanism has more work to do. Finally, the size of the write operations in the Sprite trace file are smaller than in the CHARISMA one.

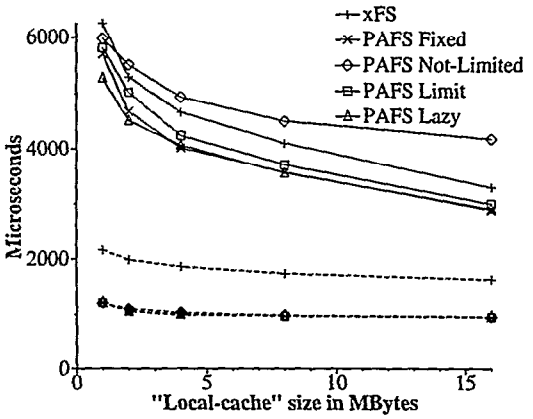

Figure 8: Local-cache size influence (NOW).

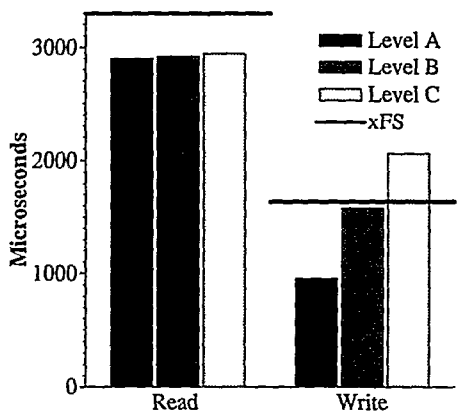

Figure 9: Fault-tolerance performance (NOW).

Cache-Size Influence

In Figure 8, we observe that the influence of the local-cache size has the same effect as in a parallel machine. The only difference is in the *Fixed partitions* and *Not-limited* mechanisms as was explained earlier (§6.2).

Fault-Tolerance Influence

Finally, we present the performance of the fault-tolerance mechanism in a NOW (Figure 9). We can see that the difference between levels B and C is much larger than in a parallel machine. This difference is because the size of the local cache is much larger in this experiment. As many more blocks can be written in the local cache, the parity mechanism in level B has much less work than in level C.

This same reason can be applied when comparing levels B and C in PAFS against xFS with level B. The larger the local cache is, the less work the fault-tolerance mechanism has to do.

7 Conclusions

This paper has presented high-performance solutions to three of the most important problems found when designing a cooperative cache: coherence, load balancing and fault tolerance.

We have shown that avoiding the coherence problem by avoiding replication not only simplifies the file-system design but also increases the performance significantly.

We have also found that the redistribution policies should neither be too aggressive nor too conservative. We have found that *Lazy-and-limited* has both characteristics and obtains the best results.

A parity-based fault-tolerance mechanism which may obtain a high level of fault-tolerance and still allow a high-performance file system has also been presented. Furthermore, three different levels of fault tolerance have been proposed.

We have also shown that achieving high *local-hit* ratio is not the only way to design high-performance cooperative caches. Taking most of the overhead away from remote hits is also a good way to achieve high-performance cooperative caches.

Finally, although the solutions presented in this paper were designed for a parallel machine, reasonable good results have also been obtained in NOWs.

Acknowledgments

We owe special thanks to Michael D. Dahlin for answering all our questions about the way xFS and N-Chances Forwarding work. We are grateful to the people of the CHARISMA project who gathered the CHARISMA traces and 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 M. Huget, Prof. E. Markatos and Prof. Pedro de Miguel whose comments improved the contents of this paper. We are also grateful to M. Ortega for implementing the first prototype. And last, but not least, we want to thank Rajesh R. Bordawekar for his interesting comments during the shepherding process.

References

[1] ANDERSON, T. E., DAHLIN, M. D., NEEFE, J. M., PATTERSON, D. A., ROSELLI, D. S., AND WANG, R. Y. Serverless network file systems. In *15th International Symposium on Operating System Principles* (1995), pp. 109–126.

[2] ANDERSON, T. E., DAHLIN, M. D., NEEFE, J. M., AND THE NOW TEAM. A case for NOW (Networks of Workstations). *IEEE MICRO* (February 1995), pp. 54–64.

[3] BAKER, M. G., HARTMAN, J. H., KUPFER, M. D., SHIRRIFF, K., AND OUSTERHOUT, J. Measurements of a distributed file system. In *13th th International Symposium on Operating System Principles* (1991), pp. 198–212.

[4] CORTES, T., GIRONA, S., AND LABARTA, J. I/O performance of scientific-parallel applications under PAFS. Tech. Rep. CEPBA-RR-1996-23, Centre Europeu de Paralelisme (UPC), 1996.

[5] CORTES, T., GIRONA, S., AND LABARTA, J. PACA: a cooperative file-system cache for parallel machines. In *Proceedings of the 2nd International Euro-Par Conference* (August 1996), pp. I:477–486.

[6] CORTES, T., GIRONA, S., AND LABARTA, J. Avoiding the cache-coherence problem in a parallel/distributed file system. In *Proccedings of the High-Performace Computing and Networking* (April 1997), pp. 860–869.

[7] CULLER, D. E., DRUSSEAU, A., COPEN, S., KRISHNAMURTHY, A., LUMETTA, S., VON EIKEN, T., AND YELICK, K. Parallel programming in Split-C. In *Proceedings of Supercomputing'93* (1993), pp. 262–283.

[8] DAHLIN, M. D., 1996. Private Communication.

[9] DAHLIN, M. D., WANG, R. Y., ANDERSON, T. E., AND PATTERSON, D. A. Cooperative caching: Using remote client memory to improve file-system performance. In *1st International Symposium on Operating System Design and Implementation* (1994), pp. 267–280.

[10] FEELEY, M. J., MORGAN, W. E., PIGHIN, F. H., KARLIN, A. R., AND LEVY, H. M. Implementing global memory management in a workstation cluster. In *15th International Symposium on Operating System Principles* (1995).

[11] FRANKLIN, M. J., CAREY, M. J., AND LIVNY, M. Global memory management in client-server DBMS architectures. In *Very Large Data Bases* (1992), pp. 596–609.

[12] GILLET, R. B. Memory channel network for PCI. *IEEE MICRO* (February 1996).

[13] INC., S. G. Origin servers. technical overview of the origin family, 1996. http://www.sgi.com/Products/ hardware/servers/technology/overview.html.

[14] KOTZ, D., AND NIEUWEJAAR, N. Dynamic file-access characteristics of a production parallel scientific workload. In *Proceedings of Supercomputing '94* (November 1994), IEEE Computer Society Press, pp. 640–649.

[15] LABARTA, J., GIMEMEZ, J., PUJOL, C., JOVÉ, T., AND NAVARRO, J. I. PAROS: Operating-system kernel for distributed-memory parallel machines. In *Parallel Computing and Transputer Applications* (September 1992), pp. 673–682.

[16] LABARTA, J., GIRONA, S., PILLET, V., CORTES, T., AND GREGORIS, L. DiP: a parallel program development environment. In *Proceedings of the 2nd International Euro-Par Conference* (August 1996), pp. II:665–674.

[17] LEFF, A., WOLF, J. L., AND YU, P. S. Replication algorithms in a remote caching architecture. *IEEE Transactions on Parallel and Distributed Systems 4*, 11 (February 1993), pp. 1185–1204.

[18] LEFF, A., WOLF, J. L., AND YU, P. S. Efficient LRU-based buffering in a LAN remote caching architecture. *IEEE Transactions on Parallel and Distributed Systems* 7, 2 (February 1996), pp. 191–206.

[19] LENONSKY, D., LAUDON, J., GHARACHORLOO, K., GUPTA, A., AND HENNESY, J. The directory-based cache coherence protocol for the DASM multiprocessor. In *International Symposium on Computer Architecture* (1990), pp. 148–159.

[20] MARKATOS, E. P., AND DRAMITINOS, G. Implementation of a reliable remote-memory pager. In *USENIX 1996 Annual Technical Conference* (1996).

[21] NIEUWEJAAR, N., KOTZ, D., PURAKAYASTHA, A., ELLIS, C. S., AND BEST, M. L. File-access characteristics of parallel-scientific workloads. *IEEE Transactions on Parallel and Distributed Systems 7*, 10 (October 1994), pp. 1075–1089.

[22] ORTEGA, M., CORTES, T., AND LABARTA, J. Implementation of a cooperative file-system cache on PAROS. Tech. Rep. CEPBA-RR-1996-8, Centre Europeu de Paralelisme (UPC), 1996.

[23] PATTERSON, D., GIBSON, G., AND KATZ, R. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (June 1988), ACM Press, pp. 109–116.

[24] SARKAR, P., AND HARTMAN, J. Efficient cooperative caching using hints. In *2nd International Symposium on Operating System Design and Implementation* (1996), pp. 35–46.

[25] WHEAT, S. R., MACCABE, A. B., AND RIESEN, R. PUMA: an operating system for massively parallel systems. In *27th Hawaii International Conference on System Science* (1994).