COOPERATIVE CACHING AND PREFETCHING IN PARALLEL/DISTRIBUTED FILE SYSTEMS

Antonio Cortés Rosselló

UPC. Universitat Politècnica de Catalunya

Departament d'Arquitectura de Computadors

Barcelona (España)

■

Jesús Labarta Mancho

Advisor

A THESIS SUBMITTED IN FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE
Doctor en Informàtica

5

COOPERATIVE CACHING AND DISTRIBUTED CONTROL

5.1 MOTIVATION

After the results obtained by the centralized version of our cooperative cache, we are ready to present the distributed version. This new design will solve most of the problems found in the centralized version but while still maintaining the main ideas presented in the previous chapter.

Scalability is the first problem we want to solve with this distributed version. As it was already explained in the previous chapter, a centralized cooperative cache can obtain quite good performance on machines with tenths of nodes, but if larger machines are used a distributed version is needed.

In the previous version, we wanted to prove that exploiting physical locality was not mandatory to achieve a high-performance cache. Once this idea has been proved, there

is no need to be so strict and ignore all physical locality. In this new version, we will try to achieve some more locality as long as it does not add overhead to the remote hits.

Another important objective of this distributed version consists of proving that avoiding replication to avoid coherence problems still works when the control is distributed. The results presented in the previous chapter give the feeling that this is a good solution to the coherence problem. Anyway, it has to be proven in a system where the coherence problem really exists.

Finally, we want to compare this distributed version with xFS in its original form. This will help us to finally prove the ideas presented in this work.

5.2 FILE-SYSTEM ARCHITECTURE

In this section, we will describe the architecture of PAFS, the parallel/distributed file system where our cooperative cache with a distributed control is located. This system is based on several servers that take care of the different tasks that have to be done in a file system. To run PAFS we need, at least, three sets of servers: disk-servers, cache-servers and a redistribution-server (Figure 5.1). A fourth set of servers, the parity-servers, is only needed whenever the fault-tolerance mechanism is activated. In this chapter, we will focus on the three first sets and the fault-tolerance mechanism will be described in Chapter 6.

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 data 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

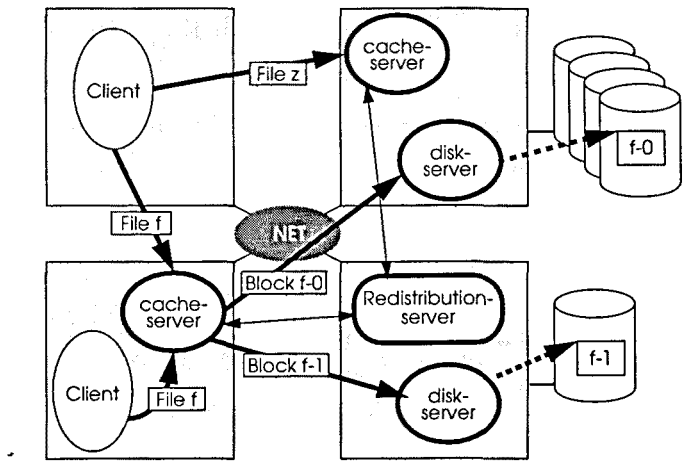

Figure 5.1 PAFS architecture.

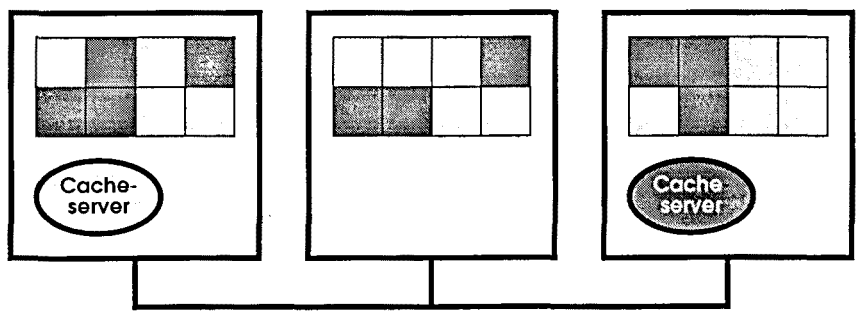

Figure 5.2 An example of two cache-server partitions in a three-node machine.

of files. It keeps all the information needed to find a file block in the cache, or 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 idea is very similar to the one presented in PACA but instead of computing the node to place the block, we compute the cache-server responsible for this block.

The whole global cache is distributed among the cache-servers. All the buffers assigned to a cache-server build this server's partition. In Figure 5.2 we present an example with two cache-servers and their partitions in a three node machine. We can see that the buffers assigned to each partition may come from any node indistinctly. It is also important to see that the size of the partitions may be different for each cache-server. 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, the system needs a mechanism to dynamically reassign buffers among the cache-servers. The **redistribution-server** is in charge of such redistribution. Different policies to distribute buffers are presented later in this chapter.

A very important issue in the design of PAFS was to enforce the lack of dedicated nodes. Servers may run in any node and share it with user applications without interfering too much in their performance. Disk-servers are the only exception as they have to run on the nodes where disks are connected, but they can still 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 algorithm to distribute file blocks among the disks. The disk of the first block is chosen using a hash function similar to the one used to bind files and cache-servers. The rest of the blocks are placed among the disks using an interleaved algorithm. This may not be the best way to distribute blocks among the disks, but it was good enough for our purposes. Furthermore, should any other distribution mechanisms have been used, the ideas presented in this work could have also been applied.

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 it wakes up, it searches for all dirty file blocks and updates them in the disk. This mechanism and interval have been chosen as the ones found in a *Unix* system [RT74].

5.3 BLOCK DISTRIBUTION AND REPLACEMENT ALGORITHM

As has already been explained while describing the file system architecture, PAFS has several cache-servers that manage the file blocks in the cooperative cache. This introduces a new level in the distribution of blocks when compared to the centralized version. In the previous version, we only needed one mapping function to decide in

which node a given file block was to be placed. Now, the distribution algorithm that places blocks into nodes has two mapping functions. The first one decides which cache-server is in charge of the block while the second one places the block in one of the nodes where the cache-server has buffers.

$$cache\_server = fh(file\_ID) = (rand(file))MOD\ cache\_servers \tag{5.1}$$

The first mapping is done on a per-file basis. This means that all the blocks from a file are always managed by the same cache-server. This simplifies many aspects of the design as each server behaves nearly as a centralized server for a given set of files. If only one server is in charge of a given file, it keeps all the information for this file and does not need to communicate with any other cache-server to fulfill any requests on this file. This distribution of files among cache-servers allows us to implement the POSIX semantic [Pre92] or even stricter ones as in a centralized system. The distribution of files among cache-servers is done in a similar way as blocks among nodes in the centralized version. We use a hash function that, from the file name (or file-ID), computes which cache-server will be in charge of all the blocks of that file (equation 5.1). If this hash function is good enough, the load of the cache-servers will also be balanced. Furthermore, this is a very simple mechanism for the clients to know which server to contact when they have to access data from a given file.

The second mapping consists of placing the file block in one of the nodes where the cache-server has buffers. This time we use a more flexible mapping than in the centralized version. A block is not assigned to a given node a priory. The replacement algorithm will be the one to decide which node is going to keep the new block. Actually, a block may be placed in a different node each time it enters the cache. The important thing is that we still maintain the restrictions of not allowing, neither replication, nor reallocation of blocks while they are in the cache. The only possible way of reallocating a block appears after the block has been discarded from the cache. Once it is requested again, as it is not in the cache (it was discarded) the cache-server can place it in any node. This is not really a reallocation as the block is not moved from one node to

another. It is as if it were the first time that this block entered the cache. This new location of the block will not be modified as long as the block remains in the cache.

The replacement algorithm used in this distributed version is a little bit different from the one proposed in the centralized version. The first difference relates to the choice of the node from which a block may be replaced. In the centralized version, the node from which a block was to be replaced was inherent to the new block. A given file block could only be placed on a given node and thus a block from that node had to be replaced. Now, the replacement algorithm does not have to take nodes into account, it can replace any block from any node. The only restriction is that the replaced block is owned by the same cache-server as the new one. This means a buffer from the partition of this cache-server will be used. The second difference is that we have relaxed a little bit the idea of not encouraging physical locality. We still defend that it should not be the key issue in the design, but if some locality can be achieved with no overhead, it would not be wise to ignore it. The replacement algorithm we propose is PG-LRU which stands for Pseudo-Global LRU. As can be extracted from its name, it is based on the well-known LRU but for a simple modification to increase physical locality.

To describe the modification added to increase locality, we first need to define the concept of **queue-tip**. In a LRU replacement algorithm, all the blocks are kept in a queue ordered by their last access time. We define the **queue-tip** as the last portion of this list where the least-recently-used blocks are kept. The size of the queue-tip is defined by the system administrator using a percentage of the list.

The replacement algorithm works as follows. Once a block has to be replaced, the cache-server examines the buffers in the queue-tip. If a buffer from the queue-tip is found in the same node as the client requesting the new block, this buffer will be used. This will increase the number of local hits without adding any overhead to the remote-hit operations. We should remember that this overhead in the remote hits was detected as a problem in the previous chapter. If no such buffer is found, the least-recently-used block will be replaced, no matter in which node it is located. It is important to notice that each cache-server will use this algorithm to replace the blocks from their own partitions and the blocks from other partitions will not be affected.

We can see now that there is no communication between cache-servers. As a file is only handled by a single cache-server, this server will have all the information about the file and no other server will know about it. Furthermore, the replacement algorithm only examines the blocks in the partition of the cache-server that needs to replace a block and does not need to cooperate with any other cache-server. This lack of communications between cache servers, will allow a high scalability of PAFS which was one of the objectives.

5.4 COHERENCE MECHANISM

In the same line as the centralized version, we avoid the coherence problem by avoiding replication. Both, the block distribution and replacement algorithms, have been designed to avoid replication. As we obtained a good performance in the centralized version, it is reasonable to suppose that the same idea can also be valid in the distributed version.

Avoiding replication to avoid the coherence problem is a very important difference if compared with the solutions presented by other file systems and caches. As was explained in Chapter 2, the mechanisms proposed to solve this problem while maintaining the *Unix* semantic, consist of asking permission before modifying a block. Before this permission is granted, it is necessary to invalidate the copies that have become obsolete. Another possibility consists of actualizing all the copies at the same time. Both solutions imply a significant overhead our file system is not willing to take.

5.5 BUFFER REDISTRIBUTION

It is quite straight forward to see that the load of the servers will not always be balanced. There may be some periods of time where a given cache-server needs more buffers than the others. On the other hand, some server may be idle while other are heavily loaded. For this reason, it would be a good idea to allow a dynamic redistribution of the cache buffers among the servers. A server that needs to keep more blocks should have more

buffers than another that hardly keeps any file blocks. In this section, we describe the mechanism proposed to distribute the buffers among the cache-servers in order to adapt to these changing needs of the system. Furthermore, we also present some policies that can be used with this mechanism and that will be evaluated in the performance section(§5.8).

The mechanism we propose consists of a periodic redistribution of the buffers among the cache-servers. The redistribution-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 since the last redistribution. From now on, we will refer to this time between two redistributions as the redistribution interval. Using this information, and the current size of each cache-server partition, the redistribution-server redistributes the buffers proportionally to each server's working-set. Once the redistribution-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 it.

It is also important to notice that once a buffer changes the partition it belongs to, it 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. For this reason, it is important to limit the number of buffers moved from one partition to another. Moving a buffer to a partition where it will not be used may degrade the hit ratio as a cached block was discarded for no good reason. Another important effect of changing the owner of a block is that, if it is dirty, it has to be sent to disk before the new cache-server can use it.

The redistribution policies presented in this thesis 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. Second, this movement of buffers between partitions 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 set a limit to both, the number of buffers a cache-server may gain and the number of buffers that can be lost by a server in each redistribution. Limiting the buffers a cache-server may lose should be a function of the number of buffers it has in its partition. Using an absolute number makes no sense as it may be too high for servers with very few blocks while it could be too small for servers that own many buffers. We propose that the maximum number of buffers that a cache-server may lose in each redistribution should equal a predefined percentage of its partition size. This percentage should be tuned by the system administrator. 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. It makes no sense to let a cache-server to gain more buffers than the ones it can use. For this reason, we have to find an absolute number that gives an idea of the number of new buffers that can be used between redistributions. Our simulations show that the number of blocks that can be physically read in a redistribution interval gives this bound. For this reason, limiting the number of gained buffers to the number of new blocks that can be read in a redistribution interval seems an interesting idea.

The instant when buffers change their ownership may also have a significant impact on the effectiveness of the redistribution algorithm. One possibility is to implement an eager algorithm that changes the ownership of a buffer as soon as all the nodes have been notified. This eager algorithm has the problem that many reassigned buffers may not be used until 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 redistribution-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 mechanism, at most one buffer changes ownership without being used.

The first redistribution 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 redistribution policy. This policy is quite similar the previous one. The only difference is that the number of buffers a cache-server may gain is limited by the number of new blocks that can be physically read from disk.

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

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

5.6 ADAPTABILITY TO THE VARIATION OF SERVERS, NODES AND VIRTUAL MEMORY

As we already explained in Chapter 4, it is very important to allow the cache to adapt to the needs of the whole system. This means that we should allow nodes to enter and leave the cache. We should also allow the nodes to decide the number of buffers that want to share with the global cache. In this section, we briefly describe the mechanisms proposed to allow this adaptability.

Whenever a given node decides to start taking part in the cooperative cache it only has to send a message to the redistribution-server. From that point, the redistribution-server will know that those buffers are ready to be used in the next redistribution. On the other hand, if a given node decides to leave the cooperative cache, it also has to send a message to the redistribution-server. This server will inform all cache-servers that all the buffers from that node have to be freed. If any of those buffers is dirty, the cache-server in charge of the dirty block will have to send it to the disk before freeing it.

In a similar way, our mechanism allows a given node to change the size of its local cache in favor of the virtual-memory system as proposed in Sprite [NWO88]. As speed may be important in this case, the redistribution-server may decide to start a redistribution before the predefined interval is over.

A different thing occurs when a new cache-server is added or taken away from the system. This modification in the number of cache-servers implies a modification in the hash function and the files have to be redistributed among the cache-servers according to the new hash function. To do this redistribution, all the cache-servers will have to communicate to pass their information about the files they owned to their new owners. Furthermore, the buffers with already cached blocks will be moved to the partition of the new responsible for the file blocks kept in the buffer. In the case that the cache-server leaves the cache, all its empty buffers will belong to nobody until the next redistribution. This movement of responsibilities has a significant overhead, but we believe that it is not important as this situation should not be very frequent.

5.7 TUNING PAFS

In this section, we evaluate the values we need to set in the parameters of the redistribution algorithm. We need to set the queue-tip size, the redistribution interval and the percentage of blocks that a cache-server may lose in each redistribution. Once we have set these values we can try to evaluate the proposed redistribution policies in order to choose the most appropriate one.

It is quite straight forward to see that all these parameters are correlated among them. For example, the limitation on the percentage of buffers that can be lost by a cache-server will depend on how often this distribution is done. To find the best values for these parameters, we should design a full factorial experiment [Jai91]. The problem with this type of experiments is the large number of tests that have to be run and this is specially problematic if the simulations take as long as ours do. As we only want a reasonable good set of values (not necessarily the best ones), we have decided to study each parameter by itself keeping the rest of parameters untouched. This is a simplistic study, but it is good enough for our needs.

It is important to notice that the queue-tip size, the redistribution interval and the percentage of lost buffer are parameters that should be tuned by the system administrator. In this section, we only propose some values that should give a good performance in most environments. Furthermore, we needed to set these parameters to run the simulations presented in this work.

5.7.1 Queue-tip Size

The first parameter we need to set is the queue-tip size. This value should be large enough to increase the physical locality of the cache while at the same time should not imply a search of the whole LRU list. For this reason we have run several experiments varying this percentage from 2% to 20%. These executions have been run with the fixed partition policy as we still have not decided any of the other parameters. Figures 5.3 and 5.4 present the percentage of physical locality obtained by the different executions. This percentage has been calculated dividing the number of local hits by the total number of accessed blocks.

As we can see in the figures, it makes sense to search up to 5% of the queue to increase locality. Large queue-tips do not increase the locality in any significant way and thus are not needed. As a result of this experiment, all simulations presented from now on will have a queue-tip size of 5% of the LRU-queue.

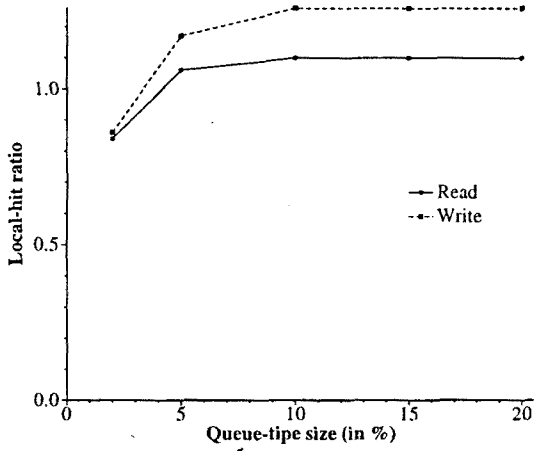

Figure 5.3 Influence of the queue-tip size on the local-hit ratio (PM/CHARISMA).

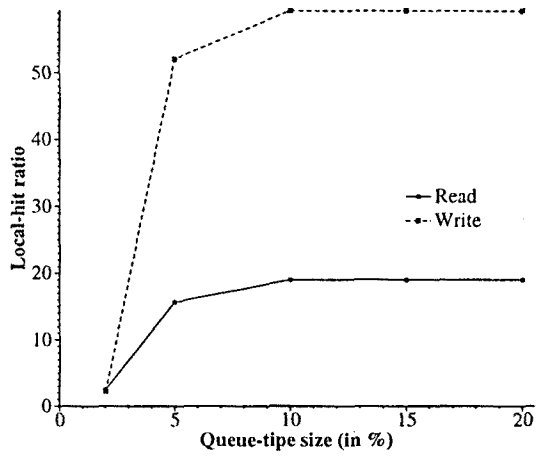

Figure 5.4 Influence of the queue-tip size on the local-hit ratio (NOW/Sprite).

5.7.2 Redistribution Interval

To study the impact the redistribution time has on the system performance we have used the *Not-limited* redistribution policy which uses an eager redistribution and only limits the number of buffers a cache-server may lose. The percentage of the buffers a cache-server may lose has been set to 10% and the queue-tip size was set to 5% of the LRU list. In Figures 5.5 and 5.6 we present the average read and write times obtained using different redistribution intervals. This study has been done using both workloads to find an interval appropriate for both of them.

As we can see, redistribution is not needed under the sprite workload but it is extremely necessary under the CHARISMA one. It is clear that the interval cannot be too large because large redistribution intervals obtain bad performance results under the CHARISMA workload. On the other hand, the Sprite workload is very balanced and redistributing buffers does not help to increase its performance. This means that very frequent redistributions have to be avoided. This leaves us the interval between 5 and 60 seconds. Any values in this interval should obtain quite good performance in both systems. As we have to chose one, we have picked a 5 second interval. We believe that when the other limitations are set, the more frequent the redistribution is done,

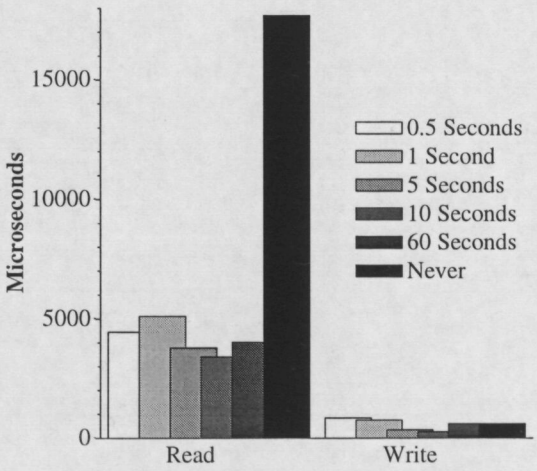

Figure 5.5 Influence of the interval between redistributions (PM/CHARISMA).

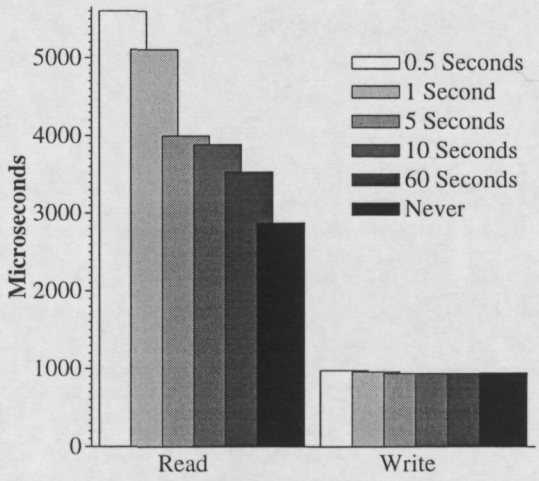

Figure 5.6 Influence of the interval between redistributions (NOW/Sprite).

the better the system will perform. Anyway, we have to remember that we are not in search of the best possible values, any reasonable value is good enough for us. The system administrator is the one that will have to decide the most appropriate values.

5.7.3 Limiting the Number of the Buffers that can be Lost

Following a similar approach as in the previous subsection, we have run simulations varying the percentage of buffers that can be lost. The simulations have been run using the *Not-limit* redistribution policy and the redistribution interval has been set to 5 seconds. Figures 5.7 and 5.8 present the average read and write times obtained.

We can observe that distributing many buffers in each redistribution does not lead to a good system performance. We can also see that, if no buffers are redistributed and the load is not balanced, as in the CHARISMA trace file, the performance obtained is also very poor. In conclusion, allowing a cache-server to lose 10% of the buffers from its partition in each redistribution seems a reasonable limit.

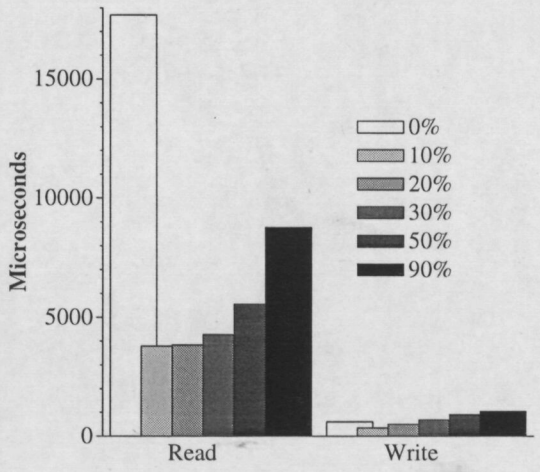

Figure 5.7 Influence of the limitation of lost buffers (PM/CHARISMA).

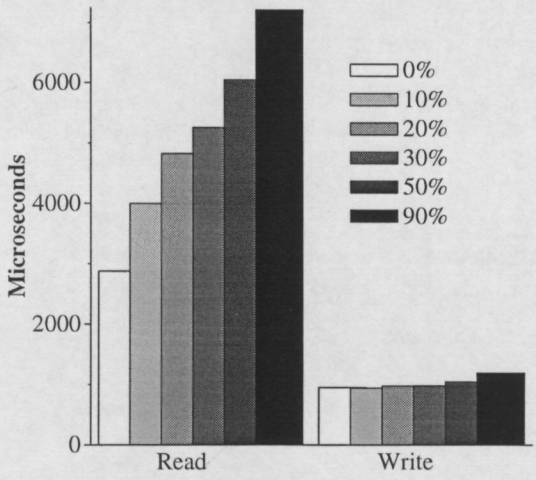

Figure 5.8 Influence of the limitation of lost buffers (NOW/Sprite).

5.7.4 Redistribution Algorithms

Now, we only have to choose one of the redistribution algorithms proposed in Section 5.5. To make this comparison, we have simulated all these policies under both environments. The redistribution parameters used are a queue-tip size of the 5% of the LRU-list, a redistribution interval of 5 seconds and only 10% of the buffers can be lost in a single redistribution. Figures 5.9 and 5.10 present the average read and write operation times obtained in these simulations.

We observe that neither redistributing buffers with no control (*No-limit*) nor *fixed partitions* are good algorithms. The first one obtains a very poor performance if the load is very balanced. This happens, because if a given server has a small period with little work, once it gets its usual work again, it will have no buffers to work with. *Fixed partition* is not a good algorithm either as the system may get unbalanced during some periods of time. If the partitions are not able to adapt to this changes, the system does not perform as well as it could.

If we compare the two remaining policies, we observe that both of them achieve a quite similar performance under both workloads. The only difference is that *Limited* uses an eager redistribution policy while *Lazy-and-limited* uses a lazy one. The lazy

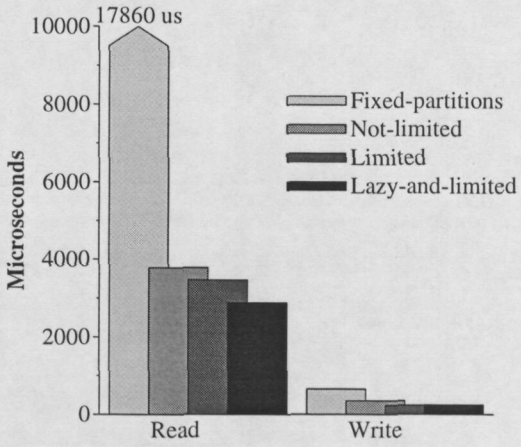

Figure 5.9 Influence of the limitation of gained buffers (PM/CHARISMA).

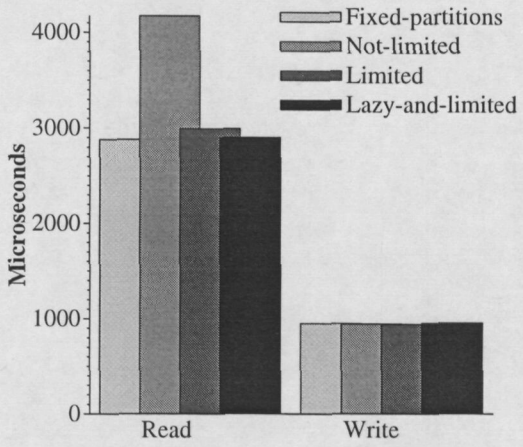

Figure 5.10 Influence of the limitation of gained buffers (NOW/Sprite).

policy allows the system to obtain a better performance but its implementation is more difficult and adds more communication between the cache-servers. We believe that any of the two policies could be used.

In the rest of this work, we will use the *Lazy-and-limited* redistribution policy as it obtains a better performance than the rest of studied policies.

5.8 EXPERIMENTAL RESULTS

Once we have chosen the redistribution policy and all the parameters needed to set PAFS running, we should compare it against xFS. This time, the version of xFS is the original distributed version.

5.8.1 Algorithm Comparison

Parallel Machine and CHARISMA

In this subsection, we compare the read and write performance obtained by both file systems. Table 5.1 presents the average read and write operation times and the *hit*

	PAFS	xFS
Average read time	2886.77 μs	9273.16 μs
Average write time	245.03 μs	855.45 μs
Local read *hit* ratio	0.79%	63.99%
Remote read *hit* ratio	94.18%	28.01%
Global read *hit* ratio	94.97%	92.00%
Local write *hit* ratio	0.86%	86.32%
Remote write *hit* ratio	97.87%	11.95%
Global write *hit* ratio	98.73%	98.27%

Table 5.1 Average read/write times and *hit* ratios obtained by both file systems (PM/CHARISMA).

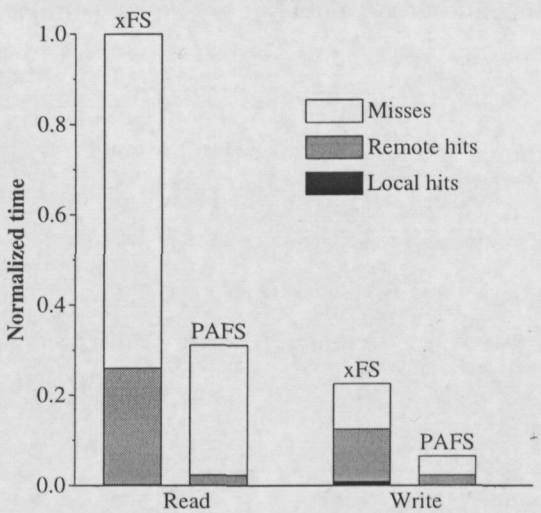

Figure 5.11 Total normalized time spent by PAFS and xFS performing read and write operations (PM/CHARISMA).

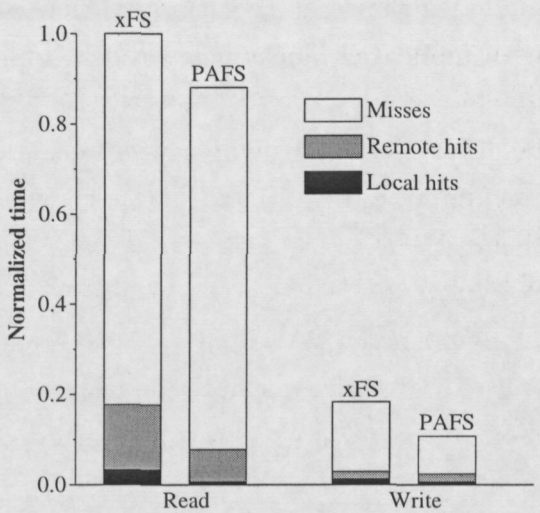

Figure 5.12 Total normalized time spent by PAFS and xFS performing read and write operations (NOW/Sprite).

ratios obtained. We observe that both read and write operations are faster with PAFS than with xFS. To explain this improvement in performance, we will first focus on the read operations and a dissertation on write operations will follow.

Figure 5.11 shows the total time spent performing read and write operations. These times have been normalized to the larger one (reads on xFS) to make the graph easier

to understand. Each bar shows the time spent working on *misses*, *remote hits* and *local hits* for each operation in both file systems. The time spent performing *global hits* can be easily obtained by adding the time spent on both *local* and *remote hits*.

The first thing we observe is that the time spent by PAFS working on read *global hits* is much lower that the one spent by xFS. This happens because PAFS obtains a higher *global-hit* ratio than xFS and because *remote hits* in xFS take longer than in PAFS.

The percentage of *global hits* obtained by PAFS (94.97%) is higher than the one obtained by xFS (92.0%) because of the better cache utilization made by PAFS. As replication is allowed in xFS, quite a few buffers keep replicated blocks while this space is used by PAFS to hold other file blocks that might also be important. This replication increases the *local-hit* ratio but decreases the number of *global-hits*. Thus, the cache with replicated blocks behaves as a smaller one.

In order to explain why *remote hits* take longer in xFS than in PAFS we will first calculate the theoretical time spent by each file system and afterwards we will compare them.

A *remote hit* in PAFS only means a remote copy of the requested bytes from the cache to the user address space (equation 5.2).

$$PAFS\_R\_hit = R\_memory\_copy(request\_size) \tag{5.2}$$

As xFS places a great interest on obtaining a high *local-hit* ratio, a *remote hit* always copies the block into the local cache. This means that two copies are needed, one from the remote cache to the local one and another from the local cache to the user address space (equation 5.3). We also have to take into account that the whole block is copied through the network in xFS while only the requested bytes are sent through the network by PAFS. Furthermore, each *remote-hit* implies a communication with the manager that has the information about the location of the remote block. In this communication, a request and a notification are sent through ports (equation 5.4). The

simulation results show that it is quite frequent to find that the buffer needed to place the requested block has to be forwarded to another node to increase its lifetime. The influence this has on the average *remote-hit* time can be seen in equation 5.5 where the time of forwarding a block is weighted by the probability of this happening (*pf*). In a similar way, if the buffer to be used is dirty, the system will need to send it to disk before using it. The operations needed are the same as in a forwarding (equation 5.6) but the probability (*pd*) is different. Summarizing, if we add all these components together, we will have the theoretical formula of the time that takes to serve an average *remote-hit* in xFS (equation 5.7).

$$basic = R\_memory\_copy(block\_size) + L\_memory\_copy(request\_size) \quad (5.3)$$

$$manager = 2 * R\_port \quad (5.4)$$

$$forward = [2 * R\_port + R\_memory\_copy(block\_size)] * pf \quad (5.5)$$

$$dirty = [2 * R\_port + R\_memory\_copy(block\_size)] * pd \quad (5.6)$$

$$xFS\_R\_hit = basic + manager + forward + dirty \quad (5.7)$$

To be able to calculate these two equations, we first need to know the values of *pf*, *pd* and the average *request_size*. These probabilities are taken from the simulations and have the following values: *pf*=.75, *pd*=.51 and *request_size* = 6741 bytes. The probability of forwarding a block is quite high because most blocks are forwarded at least twice in their live. The number of times a block is forwarded can be even higher if the block is accessed after it has been forwarded and before it is finally discarded as its forwarding counter is reset to 0. The probability of finding a dirty block may also seem quite high. This happens because of the small local-cache size. As all writes are placed in the local cache, and write operations are very fast, the cache is filled quite easily. Once the cache is full of dirty buffers, the following accesses will have to send the block to the server. As the cache is filled so easily, the `syncer` does not have time to clean the blocks before they are used.

If we calculate both, the time spent in xFS and PAFS (shown in equations 5.2 and 5.7) we find than xFS takes around 5 times longer than PAFS in performing a *remote hit.*

We have to notice that this calculations did not take into account the CPU consumed, nor the contention of the network, nor the time spent waiting for a server to become ready, etc. If all this time is taken into account (as was done in the simulations) this 5 becomes even larger. On the other hand, PAFS only has 3.3 *remote hits* for each one found in xFS. This means that the overhead paid to increase the *local-hit* ratio is much higher than the benefits obtained.

Finally, it can also be observed that the time spent working on *misses* is also larger in xFS than in PAFS. This is partially because of the higher *miss* ratio explained in the above paragraph. The larger amount of *misses on dirty* that occur in xFS (as explained earlier) also has some impact on the time spent working on *misses*.

Let's now explain the gain obtained in write operations. The main reason for the performance difference is the overhead produced by the coherence algorithm. Most writes have to ask for the block ownership and quite a few of them also have to invalidate the copies kept in other nodes. All these operations are quite time consuming and increase the average write time in xFS. Besides, block forwarding and the extra copies needed to bring the block to the "local-cache" before modification also have a significant impact on the write performance.

Another reason that explains the poor write performance obtained in xFS is the higher number of *misses on dirty* found when compared with PAFS. The same explanation given for read operations can be applied in this case.

Network of Workstations and Sprite

Now it is time to compare both file systems in a network of workstations. In Table 5.2 we present the average read and write operation times plus the *hit* ratios obtained in such environment. The first thing we observe is that PAFS is still faster than xFS but the difference is much less important. As we did in the above section we will first explain the behavior of the read operations and the write operations will follow.

	PAFS	xFS
Average read time	2903.80 μs	3294.77 μs
Average write time	957.34 μs	1635.77 μs
Local read *hit* ratio	12.22%	65.51%
Remote read *hit* ratio	73.15%	20.31%
Global read *hit* ratio	85.37%	85.82%
Local write *hit* ratio	24.71%	43.52%
Remote write *hit* ratio	25.21%	6.35%
Global write *hit* ratio	49.92%	49.87%

Table 5.2 Average read/write times and *hit* ratios obtained by both file systems (NOW/Sprite).

In Figure 5.12 we observe that the difference between the time spent performing read *global hits* is not as big as it was in the parallel machine. This happens because now, the difference between the time spent to perform a *remote hit* in PAFS and in xFS is not as big. As the "local-caches" are bigger the probability of forwarding a block and the probability of finding a dirty buffer are much lower. Furthermore, the Sprite workload has much less file sharing than the CHARISMA one. This also takes away a lot of the overhead added by the coherence mechanism.

We can also observe that the time spent working on read *misses* is nearly the same in both file systems. This is because of the similar *global-hit* ratio achieved by both of them (Table 5.2). Similar *hit* ratios are obtained as the cache is big enough to hide the problem produced by the replication done in xFS.

The difference between write-operation average times obtained by both file systems has also decreased in this environment if compared to the one obtained in a parallel machine. This can be explained with the same reasoning used with the read operations. There is much less overhead due to the coherence mechanism and the number of *misses on dirty* is also much lower than the one observed in xFS.

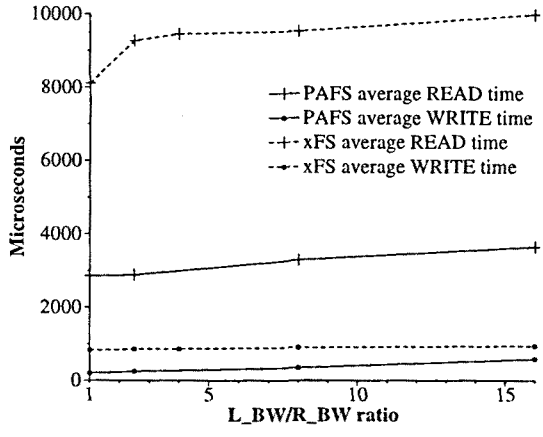

Figure 5.13 Interconnection-network bandwidth influence in the average read and write operation times (PM/CHARISMA).

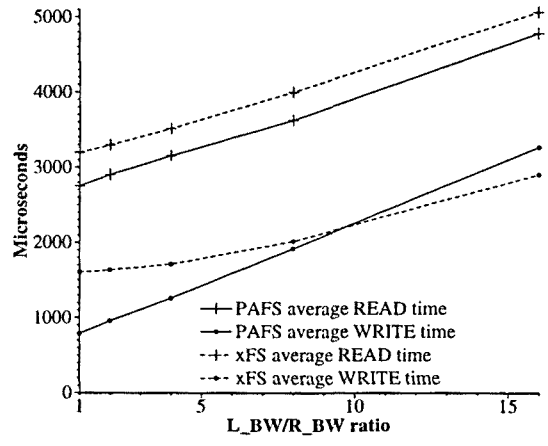

Figure 5.14 Interconnection-network bandwidth influence in the average read and write operation times (NOW/Sprite).

5.8.2 Interconnection Network Influence

Parallel Machine and CHARISMA

An important part of this work is to study the influence the communication speed has on the results presented in the previous subsection. To perform this study, we ran several simulations varying the network bandwidth and leaving the local one unmodified. In Figure 5.13, we can examine this variation. The X axis shows the ratio between the local memory-copy bandwidth (L_BW) and the interconnection network one (R_BW). The interval used starts at 1 where both bandwidths are equal and ends where the remote bandwidth is 16 times slower than the local one. This interval should include most parallel machines.

We observe that the bandwidth ratio affects the read operation in a similar way to both file systems. This means that the time gained by xFS due to its higher *local-hit* ratio is lost because of its *remote hits*. As all communications needed to serve a *remote hit* go through the interconnection network, the xFS *remote hits* are highly penalized.

Write operations behave in a different manner. This difference basically resides in the way write *misses* are treated by both file systems. In PAFS, a *miss* nearly always means a remote copy as the new block will probably replace a block in a remote node. In consequence, a slow down in the interconnection-network speed means a slow down in the write operation. On the other hand, a *miss* under xFS is always handled locally. It replaces a local block and no remote copies have to be done unless a forwarding is needed. These extra remote copies performed by PAFS cannot be outweighed by the extra work needed to maintain the cache coherent. Thus, the difference between the average write time in both file systems decreases as the network slows down.

Network of Workstations and Sprite

The influence of the interconnection-network bandwidth (Figure 5.14) 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 becomes slower than in xFS.

The intersection point is 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 is not as significant. Second, there is much more file sharing in the CHARISMA trace file than in the Sprite ones. This means that the coherence mechanism has less work to do under the Sprite workload. Finally, the average size of the write operations in the Sprite trace file is smaller than in the CHARISMA one. If user requests are large, fewer messages have to be sent to the server as only one for request is needed. As this request is always handled locally by xFS and remotely by PAFS, slowing down the network has a greater influence on PAFS than in xFS.

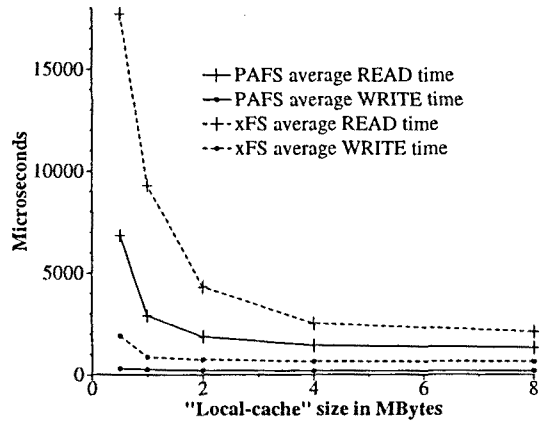

Figure 5.15 "Local-cache" size influence in the average read and write operation times (PM/CHARISMA).

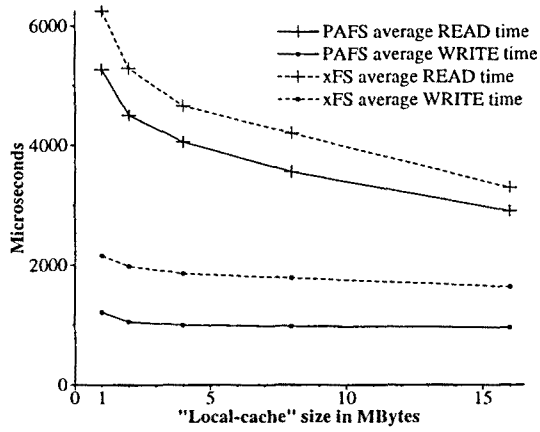

Figure 5.16 "Local-cache" size influence in the average read and write operation times (NOW/Sprite).

5.8.3 "Local-Cache" Size Influence

Parallel Machine and CHARISMA

Another important aspect that was also studied is the influence the "local-cache" size has on the file-system performance. To fulfill this study we simulated "local-cache" sizes from 512KBytes up to 8MBytes. It is quite clear that 512KBytes "local-caches" are very small but it seemed interesting to observe the behavior of both system when very small caches are used.

We observed, as expected, that decreasing the cache size, increases the read average time (Figure 5.15). As was shown in Figure 5.11, more than 75% of the total time spent on read operations was used to satisfy less than 8% of the blocks (*misses*). This means that the file-system performance is led by the *miss* ratio. As can be seen in Table 5.3, smaller caches have a much lower *global-hit* ratio. Thus, read operations are also much slower.

In Table 5.3, we can also observe that PAFS always obtains a better *global-hit* ratio. This is because our algorithm tries to use the global cache better by avoiding replication. A replicated block takes the place of a block which might also be important for the

	0.5 MB	1 MB	2 MB	4 MB	8 MB
PAFS	87.69%	94.98%	96.97%	97.78%	98.02%
xFS	81.85%	92.00%	95.93%	97.73%	97.78%

Table 5.3 "Local-cache" size influence on the read *global-hit* (PM/CHARISMA) ratio.

applications running on the system. This is important as long as the caches are small. The bigger the cache is the less important this replication side effect is.

In Figure 5.15, we can see that the read performance obtained by PAFS is much higher than the one obtained by xFS when small "local-caches" are used. The reason behind this behavior is the *global-hit* ratio obtained by both algorithms.

Another conclusion that can be obtained from Figure 5.15 is that write operations are nearly unaffected by the "local-cache" size. This is basically because the *global-hit* ratio is not very important on the performance of the write operations. A *hit* or a *miss* take, more or less, the same amount of time. Both, *hits* and *misses*, mean a copy of the requested bytes from the user area to the cache. This is only true if writing the block does not need to read the previous information from the disk, but we have observed that this does not happen too often.

Another possible interpretation of the results presented in this subsection is that PAFS, when using 1MByte "local-caches" has a similar behavior than xFS with a 8MByte "local-caches". This can be used to decrease the size of the cache in order to give more memory to the applications running on top and still have the same file-system performance as the one obtained by xFS.

Network of Workstations and Sprite

In Figure 5.16, we can also observe that the influence of the "local-cache" is similar to the one observed in a parallel machine environment. The same reasoning can also be used.

5.9 CONCLUSIONS

In this chapter, we have presented PAFS, a parallel/distributed file system that has a cooperative cache with no coherence problems. We have shown that avoiding replication to avoid the coherence problems not only simplifies the design, but also increases the performance of the system.

We have also proven that obtaining a high local-hit ratio is not the only way to design a high-performance cooperative cache. We have not exploited locality and we have still obtained a very fast file system.

A mechanism to allow a redistribution of the cache buffers among the cache servers has also been presented. Furthermore, we have also proposed a set of policies that could be used to redistribute these buffers. Finally, the *lazy-and-limited* redistribution policy has been detected to obtain the best performance. This happens because it allows a fast redistribution of buffers when it is really needed, but it behaves much more conservatively when this redistribution is not so needed.

We have proposed a generalization of the LRU replacement algorithm, PG-LRU, that increases the locality of the cooperative cache implemented in PAFS.

Finally, we have also shown that PAFS not only works well in a parallel machine, but it also obtains a better performance than xFS in a network of workstations.

More information about this distributed version, its characteristics and its performance can be found in earlier papers [CGL97a, CGL97b, CGL96a].

6

FAULT TOLERANCE

6.1 MOTIVATION

In a cooperative cache, file blocks are scattered among all the nodes in the network. Furthermore, these blocks are not sent to disk as soon as they are modified for performance reasons. This means that a given application may have several dirty blocks kept in the cache of a node where it is not running. In this situation, a node failure will mean that all modified blocks kept in this node will be lost and, thus, some applications that did not even know the existence of the failed node, will end up having some inconsistent files. There are environments where this situation is not acceptable and the file system should include a tolerance mechanism so that a node failure does not result in a lost of information.

Increasing the fault tolerance of the file system will add some overhead to the system as more work will have to be done. The challenge consists of making this overhead as

low as possible while still offering a reasonable degree of tolerance. In this chapter, we will propose a tolerance mechanism that adds very little overhead while still achieves a very high tolerance degree.

If we examine all possible environments where PAFS can run, we observe that their needs are not always the same. In some environments, there is no need of a fault-tolerance mechanism as a single node failure results in the whole system failure. There are other environments where speed is much more important than fault tolerance and one such mechanism is not desirable. On the other hand, there are environments where nodes fail quite easily (i.e. in a network of workstations, the owner of a node may disconnect it at any time). In this kind of environments, a tolerance mechanism is strictly necessary. For this reason, we will not only present a tolerance mechanism, but we will also propose three levels of tolerance that can be used depending on the needs of the system.

Finally, it is important to notice that we are only concerned about not losing information once a node fails. The mechanisms needed to maintain the dynamic information of the cache-servers, should any of them fail, are out of the scope of this work.

6.2 PARITY MECHANISM

To solve the fault-tolerance problem, we propose a mechanism based on parity lines and parity buffers quite similar to the one used in a RAID level 5 [PGK88]. 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.

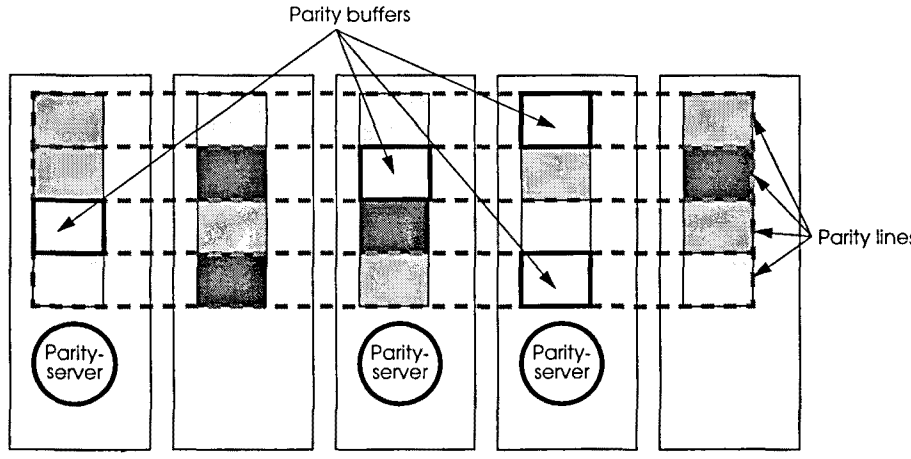

Figure 6.1 Distribution of the cache buffers in parity lines.

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 and these buffers are placed in an interleaved fashion over a predefined set of nodes. 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.

Figure 6.1 presents an example of a possible distribution of cache buffers in parity-lines on a five node machine. In this example, all parity lines have one block from each node no matter which partition the blocks belong to (each partition is represented by a different shade). We can also observe that there is a parity-server in each node where one or more parity buffers are placed. These parity-servers are in charge of all the parity buffers located in their node. It is also important to notice that parity-buffers do not belong to any partition as they hold the parity information of blocks from possibly different partitions.

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 using the information kept by all the cache-servers.

6.3 TOLERANCE LEVELS

As we have already mentioned, 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. In this section, we present the three tolerance levels offered in PAFS.

No Tolerance or Level A

There are parallel machines where a single node failure results in the whole system failure. In these machines either the hardware, or the operating system, have not been designed to handle a node failure. In this kind of machines, it would be ridiculous to assume the cost of a tolerance mechanism.

In other environments, speed is much more important than fault tolerance. The users of these machines want their applications to run as fast as possible and it is not a problem to restart the whole system if the hardware, or the operating system, fails, as

long as it is not very frequent. In this kind of environments, it is not recommended to have a tolerance mechanism as it will add an overhead that it is neither wanted nor needed.

For these kind of environments, we propose that the file system should be able to deactivate the tolerance mechanism and achieve a higher performance. From now on, we will refer to this lack of fault tolerance as **level A**.

Level B

There are some environments where we can make the assumption that a file block cached in the same node as the client which requested it may be lost when the caching node fails. In this kind of environments, it is assumed that when a node fails, its applications will also fail and no one will care for the last modifications they did. This assumption was taken in the design of the fault-tolerance mechanism implemented in xFS [ADN+95, DWAP94, Dah95].

If the afore-mentioned assumption is taken, only blocks placed in a remote cache should be guaranteed to be in the disk (or to be recoverable through the parity buffer). The degree of tolerance obtained and the overhead added will depend on the proportion of blocks that are kept in a remote cache. From now on, we will refer as **level B** to this second level of tolerance.

Level C

Level B might be enough in some environments but it would not be enough in a 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 the file system.

For instance, suppose we have a parallel `make` which has to compile two files (`f1.c` and `f2.c`) and link-edit them afterwards to generate the resulting executable file (Fig-

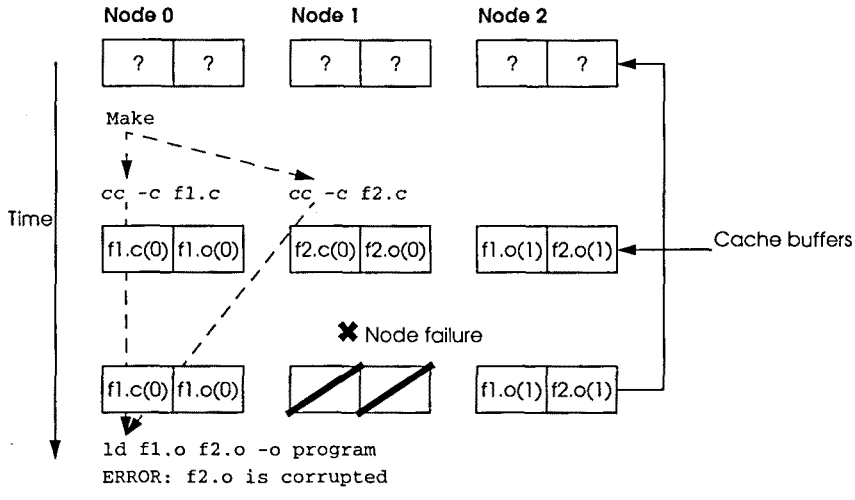

Figure 6.2 Example of a parallel make that needs a high tolerance level. The contents of each cache block is represented by the file name and the block number in parenthesis (i.e. name(#block)).

ure 6.2). As we have a parallel environment, both compilations are run in parallel in nodes 0 and 1. One of the blocks of the object file is kept in the cache where each compilation is being made and thus is not sent to the disk (or parity buffer). The other block of the object file is kept in a different node as there is no room for it in the local cache. If node 1 fails after the compilation is done, the linker will not be able to make the executable file. It will find f1.o with no problem but the first block of `f2.o` will be lost as it was kept in the failed node and the system did not send it to the disk [Dah96]. This may not be acceptable in some environments and a higher degree of fault tolerance has to be proposed.

To solve this problem we propose a third fault-tolerance level: **level C**. In this level, all modified blocks are sent to the disk (or parity-server) even if they are placed in the same node as the client using them. This offers a higher degree of fault tolerance. This third level will probably have to assume a higher performance penalty but a higher fault-tolerance level will also be achieved.

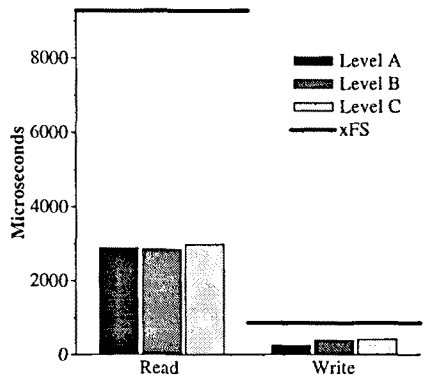

Figure 6.3 Performance of the fault-tolerance levels (PM/CHARISMA).

6.4 EXPERIMENTAL RESULTS

In this section, we will study the impact that these fault-tolerance mechanisms have on the overall system performance. We will compare the performance obtained by PAFS using tolerance levels A, B and C. These results will also be compared to the ones obtained by xFS which only implements level B.

As we have already done in the previous chapter, we will first focus on the performance obtained in a parallel machine and afterwards we will study the results obtained in a network of workstations.

Parallel Machine and CHARISMA

Let's study the impact the fault-tolerance mechanisms have on the file-system performance running on a parallel machine. Figure 6.3 presents the results obtained by PAFS using the parity mechanism with the three proposed tolerance levels (solid bars). This performance is compared to xFS which only implements level B (horizontal line).

The most important result obtained is that the cooperative algorithm implemented in PAFS still behaves better than xFS even with a higher fault-tolerance level. This proves that the parity mechanism can offer a high fault-tolerance level with very little

overhead. As communications are very fast, keeping the parity information up-to-date does not add too much overhead to the system.

If we compare the performance of the three different levels, we can see that levels B and C take somewhat longer performing write operations than level A. This is because some extra communication and *memory_copy* operations have to be done to keep the parity mechanism working. Cache-servers have to contact parity-servers to inform that a block is to be modified, the parity-server has to copy the current version of the block and the modifications from the user, the parity buffer has to be updated, and finally the cache-server has to be informed that the new version is in the parity-buffer.

Another important result is that there is no significant difference between levels B and C in PAFS. This is because the number of blocks that are kept in the local cache of the client is very small (Table 5.1). Thus, the number of blocks that have to be kept in the parity buffer is very similar. Should we stress locality, this difference would also grow.

Finally, read operations are only slightly affected as they do not take part in the parity mechanism. The only influence they may notice is a higher resource utilization.

Network of Workstations and Sprite

We present now, the performance of PAFS with the fault-tolerance mechanism in a network of workstations (Figure 6.4). We can see that the difference between the tolerance levels is much more important than the one found in a parallel machine and that the performance of the system gets much closer to the one obtained by xFS.

The difference between the performance obtained by levels B and C is due to the size of the local cache used in this experiment. When simulating a parallel machine, the local cache used was 1MByte while in this network of workstations we have 16MByte caches. As the cache size increases, the number of blocks that can be kept in the same node as the client requesting them also increases. This decreases the number of blocks

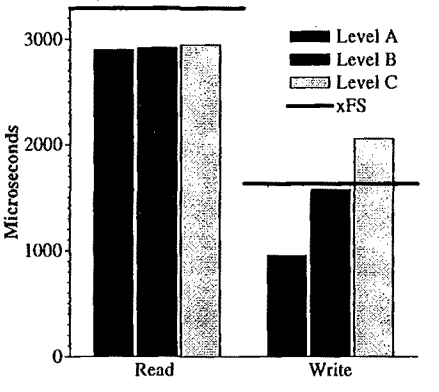

Figure 6.4 Performance of the fault-tolerance levels (NOW/Sprite).

that have to be sent to the parity-server in level B but it makes no difference in level C.

This same reasoning can be applied when comparing PAFS (levels B and C) against xFS (level B). The larger the local cache is the less work the fault-tolerance level B has to do, especially if the file system encourages locality as xFS does.

6.5 CONCLUSIONS

In this chapter, we have presented a parity mechanism that provides a high degree of fault tolerance to the parallel/distributed file systems that use it. Furthermore, this mechanism has been designed to add as little overhead as possible to the overall system. It is important to notice that this mechanism only avoids the lose of information. It does not guaranty that the system will still run should a cache-server fail. It only guarantees that no information will be lost.

We have also differentiated three tolerance levels that can be implemented using the parity mechanism. Each of these levels offers a different degree of tolerance along with a different overhead. The higher the tolerance degree, the higher the overhead is.

Finally, we have evaluated this parity mechanism with the three tolerance levels on PAFS and it has performed very well in both a parallel machine and in a network of workstations. This concludes that this parity mechanism is efficient enough to be placed in a parallel/distributed file system and still obtain a high-performance system.

More information about this fault-tolerance mechanisms, its characteristics and its performance can be found in an earlier paper [CGL97b].

7

LINEAR-AGGRESSIVE PREFETCHING

7.1 MOTIVATION

An interesting characteristic observed in cooperative caches is that the size of the global cache tends to be very big. For this reason, there are long periods of time where very old blocks are kept in the cache. There are many blocks that have not been referenced in several hours and are still kept in the cache.

The above situation brings back the idea of aggressive prefetching. These blocks kept in the cache for so long could be replaced by any other block and the system would not notice the difference. Nobody expects to find in the cache a block that was last accessed several hours earlier. The idea we propose is to do an aggressive prefetching so that these old blocks are replaced by other blocks that might be useful for the applications currently running. If many blocks are prefetched, the probability of bringing to the cache the blocks that applications really need increases. Furthermore, we can be quite

aggressive because a miss-prediction will replace a very old and probably not-needed block by another one that will also not be needed as it was miss-predicted. The important thing is that, most of the time, the useful blocks will not be replaced by these miss-predictions.

On the other hand, this idea of aggressiveness has to be controlled. For example, if the system had an extremely fast disk so that it could read the whole file between two requests, a very aggressive algorithm could be implemented. The system could prefetch the whole file after each request. This algorithm would not work well if files are larger than the cache as the same prefetch would discard its own prefetched blocks.

For these reasons, we present the linear-aggressive prefetching. This is a mechanism that builds aggressive algorithms from more conservative ones. These new algorithms will be quite aggressive while, at the same time, they will be smart enough to avoid the danger of over prefetching.

In this chapter, we will present two prefetching algorithms and their linear-aggressive version. We will evaluate their performance on top of two different file systems that implement a cooperative cache (i.e. PAFS and xFS).

7.2 PREFETCHING ALGORITHMS

In this section, we present two basic prefetching algorithms that will be transformed into aggressive ones in the next section. It is not important whether they are the best prefetching algorithms ever designed as we want to prove that linear-aggressive prefetching is a good way to improve the performance of already existent prefetching algorithms.

7.2.1 One-Block-Ahead (OBA)

The first algorithm we present is the well-known One-Block-Ahead (OBA). This algorithm is based on the idea that file operations very frequently have sequential locality

as it has been proved in may workload studies [KE90, Kot91, BHK+91, Smi78, Smi85]. Sequential locality appears when blocks of a file are accessed in a sequential manner (i.e. after accessing block n, block $n+1$ is requested).

The implementation of this prefetching algorithm is very simple. Following each read or write operation, the system prefetches the next sequential block after the ones requested by the user in the last operation.

The main problem this algorithm has is that not all files are accessed in a sequential way. There are many applications, specially parallel ones, where files are accessed using strided or even more complicated patterns [KN94, NKP+94]. In these situations, One-Block-Ahead is not able to correctly predict the next block to be accessed.

7.2.2 Interval-and-Size-Graph (ISG)

In order to present a more intelligent prefetching algorithm, we have developed Interval-and-Size-Graph (ISG). The idea behind this algorithm is that applications usually access files using a regular pattern [KN94, NKP+94]. For this reason, we have implemented an algorithm that tries to learn the patterns used by the application to be able to prefetch the blocks that are really going to be accessed.

This algorithm is a simplification of the algorithm based on data compression designed by Vitter et al. [VK96, CKV93]. In their algorithm, they try to build a tree of precedence that allows to decide which block is going to be accessed after a sequence of blocks. We have simplified it as our prediction will only depend on the last access and not the whole sequence. Furthermore, our algorithm does not base its decision on the blocks accessed but on the offset intervals as was proposed by Kotz [KE90, Kot91].

This algorithm keeps track of which offset interval comes after a given offset interval. Furthermore, it also remembers the size requested by the operation. With this information, it is quite simple to decide which blocks to prefetch. The system only has to keep the offset of the previous request and the offset of the current request. With these

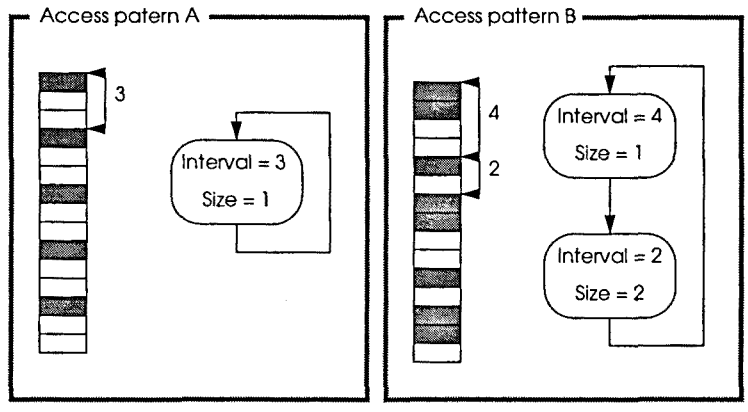

Figure 7.1 Example of a couple of access patters and their prefetch graphs.

two offsets, it can compute the interval and use it to predict the new position and the number of blocks that should be requested in the next operation.

Dynamic Graph Construction

The data structure needed to store all this information is a directed graph where the system keeps which offset interval followed another one and the number of blocks that were requested. This interval is always measured in blocks (not in bytes) because this is the basic access unit used by the file system. A node in the graph represents an offset interval that has been used when working with this file. It also represents the size of the requested information after the jump represented in the node. A link between two nodes means that after the interval kept by the first node, the user jumped the interval represented in the second node. In these links, the system may keep information such as the number of times this path has been followed, or a time stamp with the last time this link was followed.

Figure 7.1 presents an example of two access patterns and the graphs needed to be able to prefetch correctly the next requests. In access pattern "A" one of every three blocks is requested by each user operation. This pattern is very simple and a graph with only one node is sufficient to keep all the information about the pattern. Access pattern "B" is a little more complicated. After an interval of 2 blocks always comes a 4 block interval to once more jump two blocks. The accessed number of blocks also

varies depending on the interval. To keep all this information, the prefetching system needs a two-node graph as shown in the figure.

Besides the graph, the system also needs to maintain three additional variables. The first one keeps the offset of the last request. The second one keeps the number of blocks requested in the last operation. And finally, the third variable keeps the offset of the access previous to the last one.

To get a better understanding of the algorithm and its possibilities, it seems adequate to describe the way this graph is dynamically built. Once a file is open for the first time, the graph is completely empty and no prefetching can be done. The first two user operations are needed to fill the variables afore mentioned. Furthermore, the second operation has enough information to add the first node to the graph. This new node represents the first interval and the number of blocks requested after this interval. From this point on, the system enters the permanent state and its behavior is explained in the next paragraphs.

After each new access, the system should add a new node that contains the interval between the offset of the two last operations. This node should also contain the number of blocks requested in the last operation. If this node is not in the graph yet, it is inserted. Otherwise, there is no need to insert it as it already exists. Now, it is time to link this node to the node used in the previous operation. If a link already exists between these two nodes, the system only has to increment the number of times this links has been used or to place a new time stamp on it. If this link does not exist, it has to be added to the graph. This link should also be time stamped or the system should indicate that it has only been followed once.

To clarify this mechanism, we present an example that shows the creation of the graph needed for the access pattern "B" shown in Figure 7.1. Figure 7.2 presents the first five requests done by the user and the influence these operations have on the creation of the graph. In each step, the blocks marked in black are the blocks requested by the user in the current operation. Each node contains the interval it represents and the number of blocks requested after the interval. Each link also has the number of times

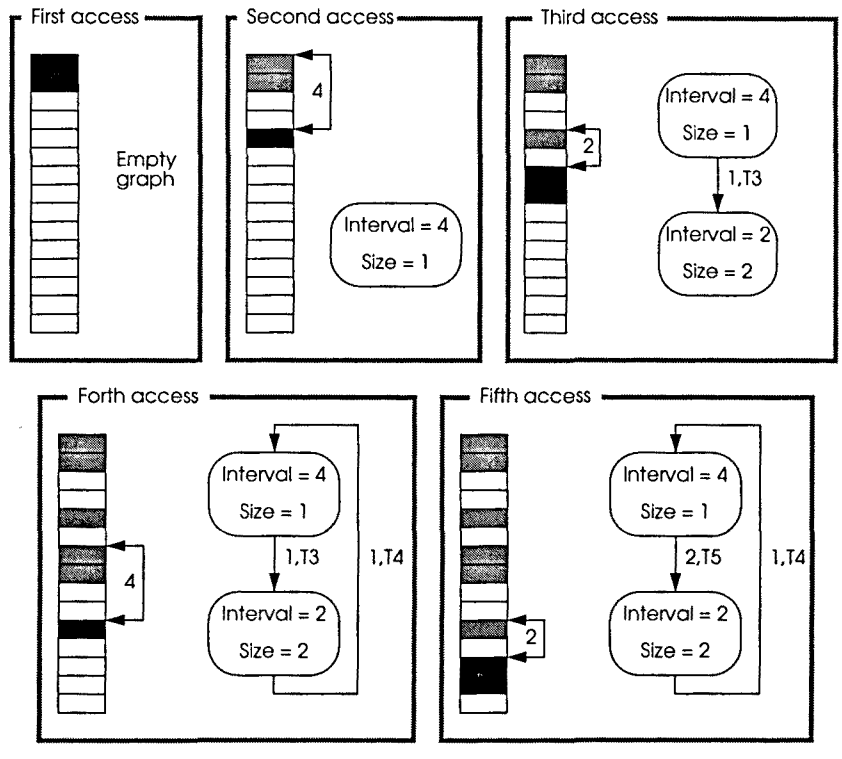

Figure 7.2 Example of a graph creation.

it has been followed and the time stamp (i.e., T1, T2, T3,...) of the last time this link was followed.

The only exception to this algorithm appears when two consecutive operations request bytes that start from the same block. As the requested block is the same as the one just requested, there is no need to prefetch it again (in most cases). For this reason, the system does not take this access into account and thus the graph is not modified in any way. This means that the graph will never have a node with an interval of 0 blocks. Furthermore, this variation in the algorithm has proved to avoid many loops that decreased the effectiveness of the prefetching mechanism.

Using the Graph to Prefetch

Before starting to describe the usage of this graph to prefetch blocks, it is necessary to define three terms used to describe the algorithm. The first one is the **last accessed**

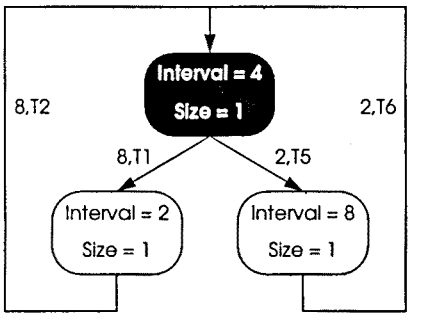

Figure 7.3 Example of a prefetch graph where nodes have more than one link that start from it.

node. This node is the node that represents the last interval requested by the user. Second, the **last prefetched node** is the last node used to prefetch blocks from the disk. Finally, the **prefetching path** is the path followed by the prefetching mechanism to go from the last accessed node to the last prefetched node.

The first prefetch starts when the graph builds the first loop. This means that a regular pattern has been detected (or at least this is the assumption). In this situation, the system gets the last accessed node and follows its link to get the next node in the graph. From this node, the prefetching mechanism extracts all the information needed to start the next prefetching. The system will prefetch as many blocks as the ones indicated in the node and the first block will be calculated by adding the interval kept in the node to the offset of the last user request. Now, the prefetching path is made by two nodes and one link.

After this first prefetch, each new user request will mean a new prefetch. Depending on the effect the request had on the graph, the system may have two different behaviors. If the last accessed node is not new and it is located in the prefetching path, it means that the algorithm is prefetching the right blocks. This means that the prefetching mechanism will use the node that follows the last prefetched block in order to prefetch the new blocks. Otherwise, it means that the prefetching path was wrong or that some changes have been done in the access pattern. In this situation, the system starts a new prefetching path starting from the new last accessed block.

So far, we have assumed that from every node only started one link. This will not always be the case, for example in Figure 7.3 we have a prefetching graph where the shaded node has two links that start from itself. This happens because after jumping four blocks, sometimes the application has jumped two blocks while other times the interval has been 8 blocks. In this situations, we have two possible alternatives. The system could follow the most used link or the most recently followed one. After some experiments we have seen that following the most used link achieves worst performance results than following the most recently used one. For this reason, the second option has been used in ISG. This means that in the example shown in Figure 7.3, after the shaded node the system would go to the node with and interval of 8 blocks as was used more recently than the other one. Following the most recently used link obtains better performance results because it adapts to variation in the access pattern much faster than the other proposal.

This algorithm is able to prefetch the correct blocks as long as the access pattern can be represented by the proposed graph. The problem appears during the first requests to a file. As the graph does not have information about the pattern that will be used, it is not able to make any decision of which blocks to prefetch. This initial state takes as many requests as different intervals are in the pattern plus 2. To solve this problem, we have made a small modification to the algorithm. If the current offset interval is not found in the graph, the system will behave as the One-Block-Ahead algorithm. We have to keep in mind that this algorithm will only be used during the time needed to build the graph. From that point on, blocks will be prefetched using the information from the graph.

The system keeps several prefetching graphs for each file, one for each process that is working with the file. As our simulations have shown that most graphs do not have more than two or three nodes, the system can keep many of these graphs without consuming too much memory. Furthermore, our simulations have also shown that if one graph is kept for each process, the prediction is much more effective than if only one graph is maintained for each file.

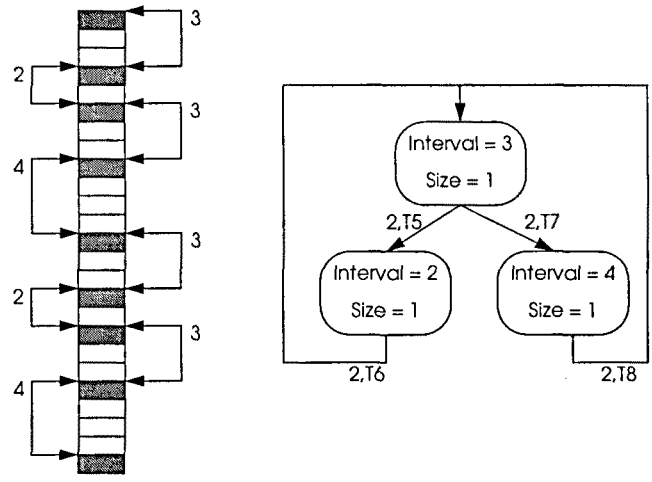

Figure 7.4 Example of a regular access pattern that cannot be prefetched by ISG.

Drawbacks of ISG

This algorithm has two major drawbacks. The first one has already been explained and solved. It consisted on the time needed to build the graph before the system is able to prefetch any block. The solution, as we already explained, consists of prefetching like a One-Block-Ahead as long as the graph does not have the current access pattern detected.

The second major drawback is that there are some regular access patterns that cannot be represented by the graph and thus cannot be correctly prefetched. For instance, the access pattern shown in Figure 7.4 is a clear example of a regular access patter that ISG would always fail to prefetch. In this example, after an interval of three blocks, it comes an interval of two, then an interval of three blocks again, then another interval of four blocks and the pattern starts again. This pattern, although it is regular, cannot be effectively represented by the proposed graph and would always follow the incorrect prefetching path leading to many miss-predictions. The solution to this problem consists of designing a better pattern representation, but this falls out of the scope of this work as we only want to present a simple but intelligent prefetching algorithm.

7.3 LINEAR-AGGRESSIVE PREFETCHING

In this section, we explain how to improve the performance of simple algorithms, such as the ones already presented, by making them aggressive. The idea is very simple, the prefetching mechanism will prefetch blocks continuously as long as it can predict blocks which are not in the cache yet. This mechanism was originally designed to work on a cooperative cache as the size of this kind of caches is quite big. Furthermore, the results we will present later in this chapter show that this mechanism can also be used in any kind of cache.

7.3.1 Aggressive One-Block-Ahead (Agr_OBA)

Making the aggressive version of One-Block-Ahead simply consists of continuously prefetch blocks in a sequential way starting from the last requested one. The system will keep prefetching the blocks sequentially until one of two things happens. If the end of file is reached, the prefetching mechanism will stop until a new request is done. If a new request is done before the prefetching algorithm reaches the end of file, and the requested block has not been prefetched yet, the system restarts the prefetching from the new position of the file pointer. If the requested blocks had already been prefetched, this means that the system prediction is correct and there is no need to modify the prefetching path. For this reason, the system continues bringing new blocks as if the user had not requested any block.

7.3.2 Aggressive Interval-and-Size-Graph (Agr_ISG)

The aggressive version of Interval-and-Size-Graph works in a very similar way. After a user request, the prefetching mechanism will decide which are the next blocks to prefetch as was done in the non-aggressive version. Once these blocks have been prefetched, it behaves as if the user had already requested the prefetched blocks and goes for the next node in the graph. This keeps going until the next block to prefetch is out of the file in which case the prefetching mechanism stops until a new request is performed. If the next user request is not the one predicted by the prefetching mecha-

nism, the algorithm modifies the graph and restarts the prefetching sequence from from the last requested block.

7.3.3 Limiting the Aggressiveness: Linear-aggressive Prefetching

As we have already said, there has to be a limit to the aggressiveness of the prefetching algorithms. To set this limitation we propose the linear-aggressive algorithms. The idea behind this limitation is to only allow the prefetching of a single block at a time for each file. We propose that a system with more than one disk should not use them to prefetch more than one block of the same file in parallel. Exploiting the parallelism offered by multiple disks can be achieved by prefetching blocks from different files. We may be prefetching as many blocks as disks but always from different files.

A more intuitive way of aggressive prefetching consists of using the potential parallelism offered by the disks to prefetch several blocks in parallel [KTP+96, PG94]. This way of aggressive prefetching achieves a very good disk utilization when only one file is being accessed. In the same situation, our proposal leaves all disks but one unused. On the other hand, we have the feeling that a parallel machine should be used to run many applications in parallel minimizing the time that only one file is being used. If as many files as disks are being accessed, our proposal achieves a similar disk utilization. Furthermore, the prefetching is more balanced as blocks from more than one file are being prefetched in parallel. We are not saying that exploiting the parallelism of several disks ends up in a bad system performance, we are only proposing another way to do an aggressive prefetching that limits the number of prefetched blocks by only prefetching one block per file at a time.

This limitation of only prefetching one block per file at a time may seem a little too restrictive. Actually, we could study increasing this limitation of parallelism to meet each site needs. Anyway, in this work we prove that having a single line of prefetching for each file is aggressive enough while, at the same time, does not flood the caches with prefetched information.

7.4 INCLUDING THE LINEAR-AGGRESSIVE ALGORITHMS IN PAFS AND XFS

Once designed the prefetching algorithms, we need to test them on top of a file system with a cooperative cache. We have done it on two different file systems. The first one is PAFS which is the file system presented in Chapter 5. The other one is xFS, the file system proposed by the NOW team. It is important to notice that these two file systems have been chosen because both implement a cooperative cache while their design issues are very different. In this chapter, we are not willing to compare both file systems but to evaluate the viability of the linear-aggressive prefetching algorithms we are presenting.

In PAFS, the management of a given file is handled by a single server. This kind of centralized management allows a simple implementation of the idea of a linear prefetching. The cache-server in charge of a file keeps all the prefetching information and can limit the number of simultaneously prefetched blocks to one. These servers have all the information about the access patterns as all requests have to go through them. All these factors favor the implementation of the proposed prefetching mechanism.

Implementing this kind of linear algorithms is not as simple in xFS if we want to maintain its original philosophy. In this system, each node is allowed to make its own decisions and only contacts a manager wherever an external help is needed. Following this idea, it makes sense that each node also makes its prefetching decisions. Furthermore, only the local node is able to have full information about the access patterns made by an application over a file. The manager only sees some of the operations, the ones that the local system cannot handle locally. This local prefetching presents a problem in order to implement a linear-aggressive prefetching. We can limit the number of blocks that are being prefetched by all the applications located in a single node, but there is no easy way to coordinate all nodes to avoid several prefetches of the same file being done in parallel. This problem could be solved using some external prefetcher, but this solution would modify the general philosophy of xFS.

What we have done consists of adding a linear-aggressive prefetching per node to xFS. This means that each node will only prefetch one block per file at a time but several blocks of the same file may be prefetched in parallel from different nodes. This is not a perfect implementation of our proposal but it is the closest design we have been able to find without modifying the original philosophy of xFS. Furthermore, it will help us to evaluate whether this linearity is a good idea or not.

There is still another implementation issue that should be explained before getting into the experimental results. It is clear that both, read and write operations are more important than prefetching a block. A user requested operation is always useful while a prefetched block might not be used if it was a miss-prediction. For this reason, we have given a lower priority to prefetching operations. Prefetching a block will never be done if other operations are waiting to be done on the same disk.

7.5 EXPERIMENTAL RESULTS

In this section, we present the results obtained while evaluating the proposed prefetching algorithms. We only present the results obtained by the read operations as the gains obtained are much more significant than the ones obtained by write operations. Write operations are not greatly affected by an increase on the number of cache hits as was already explained in Chapters 4 and 5.

The first thing we evaluate is the improvement in performance obtained by all the prefetching algorithms. We also compare the aggressive algorithms against the conservative ones. Then, we will present a study on the variation in disk traffic produced by these prefetching algorithms. It is interesting to evaluate the different loads placed on the disk by all the prefetching algorithms as it helps to evaluate the benefits compared to the amount of resources needed by each algorithm.

All these experiments were done using both, the CHARISMA and Sprite, trace files. All of them have been simulated using the parameters of a parallel machine and network of workstations as was explained in the methodology chapter (§3).

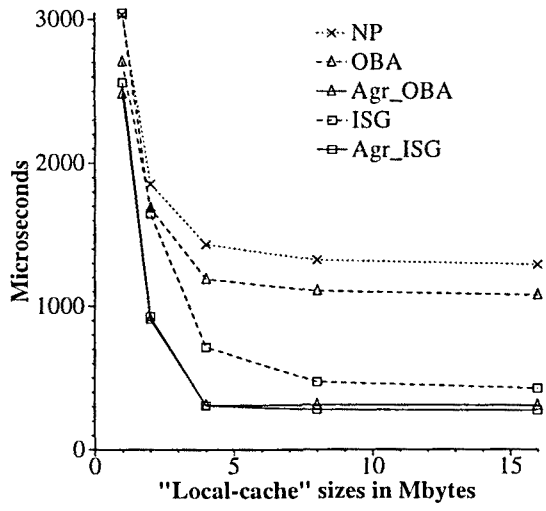

Figure 7.5 Average read operation time obtained by the presented prefetching algorithms on PAFS (PM/CHARISMA).

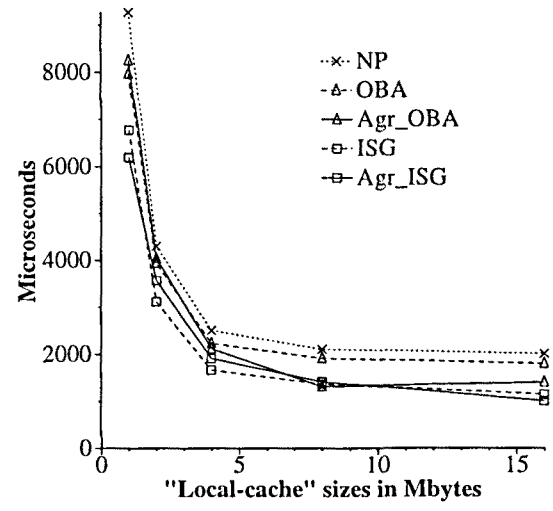

Figure 7.6 Average read operation time obtained by the presented prefetching algorithms on xFS (PM/CHARISMA).

7.5.1 Performance Improvement

Parallel Machine and CHARISMA

In this subsection, we present the average read time obtained by both, PAFS and xFS, while using the four prefetching algorithms presented in this chapter. These times have been compared to the ones obtained by the cooperative cache with no prefetching (Figures 7.5 and 7.6).

Let's first examine the results obtained in PAFS (Figure 7.5 and Table 7.1). The first thing we observe is that both linear-aggressive prefetching algorithms improve the read performance very much. This increase in performance is even bigger when large caches are used. Actually, we nearly have a 100% hit ratio. Such a high effectiveness has been obtained because there is a lot of reutilization of blocks and because the access patterns are mostly strided. Anyway, this is the kind of file-access that may be found in some parallel environment as the trace was taken from a real environment.

	PAFS					xFS				
	1MB	2MB	4MB	8MB	16MB	1MB	2MB	4MB	8MB	16MB
None	5.32	3.03	2.22	1.98	1.98	7.98	4.06	2.23	2.13	2.13
OBA	4.74	2.73	1.77	1.62	1.62	7.61	3.56	1.85	1.76	1.73
ISG	4.98	2.70	1.05	0.57	0.48	6.91	2.23	0.74	0.61	0.58
Agr_OBA	4.06	1.39	0.25	0.25	0.24	7.32	3.25	0.93	0.44	0.36
Agr_ISG	4.14	1.31	0.23	0.18	0.17	5.50	2.53	0.51	0.29	0.22

Table 7.1 Global miss ratio obtained by the presented prefetching algorithms (PM/CHARISMA).

We can also observe that ISG performs much better than OBA. This happens due to a couple of reasons. The first one is that ISG is more intelligent than OBA and thus knows how to predict blocks better. As we have already said, many of the accesses are strided ones and OBA is not able to predict this kind of patterns because it only prefetches the next sequential block and this is seldom the needed block. The second reason is that ISG is more aggressive than OBA. While OBA only prefetches a block after a read or write operation, ISG prefetches a whole set of blocks. If the user requests large amounts of blocks in each operation, ISG will also prefetch large amounts of blocks. As this is the case in some parts of the CHARISMA workload, ISG behaved as a quite aggressive algorithm and achieves a performance similar to the one obtained by Agr_OBA and Agr_ISG.

It is also interesting to see that Agr_OBA, although it is less sophisticated than Agr_ISG, achieves a very similar performance. The reason behind this behavior is, once more, the strided patterns found in the CHARISMA trace file. As Agr_OBA tries to prefetch further than just the next sequential block, the probability that the blocks of a strided pattern are prefetched is quite high. On the other hand, Agr_ISG prefetches strided patterns quite effectively as it is designed to recognize such kind of patterns. Furthermore, this similar performance results, show that the idea of making simple algorithms aggressive improves their performance quite a lot. They may even catch-up with more sophisticated ones as in this case.

Let's now compare the results obtained on xFS (Figure 7.6 and Table 7.1). With this file system we can also observe that, with large caches, linear-aggressive algorithms obtain better performance than their non-aggressive versions. The difference in performance between the aggressive and the conservative version is not as important as the one found in PAFS. This happens because the aggressive algorithms can prefetch more than one block from the same file in parallel. This makes these algorithms too aggressive and end up flooding the cache and decreasing the system performance.

We can also observe that ISG obtains a better performance than Agr_ISG with 2 and 4-MByte local caches. In the first case, this happens because the non-aggressive version obtains a higher hit ratio. With this small cache, the aggressive, and not-linear, version floods the cache discarding already useful blocks, this decreases the hit ratio as the discarded blocks will have to be brought from disk again when needed. In the second case (4-MByte caches), the performance of ISG is higher even with a lower hit ratio. This happens because so many blocks are prefetched that, during some periods of time, the managers and the disks get overloaded and cannot serve client requests as fast as they should. This also happens because the aggressive version was not the linear-aggressive one.

Network of Workstations and Sprite

When tested the prefetching algorithms on a network of workstations (Figures 7.7, 7.8 and table 7.2), the results obtained are not as impressive. Prefetching algorithms are not as effective because the files are much smaller than in the previous experiment. Small files are very difficult to prefetch as the system needs some requests to build their access pattern. Furthermore, with the presented algorithms, there is no way to prefetch the first access to a file and this can be a big percentage of the blocks accessed in a small file.

Anyway, we also observe that the aggressive versions obtain a better performance than the non-aggressive ones. This, again, proves that the idea of converting conservative algorithms in aggressive ones achieves a higher performance.

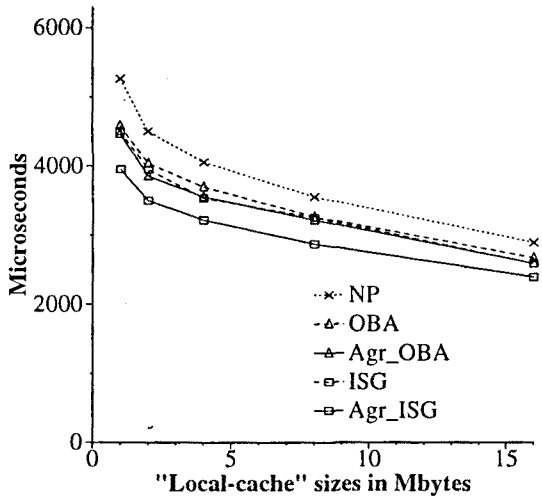

Figure 7.7 Average read operation time obtained by the presented prefetching algorithms on PAFS (NOW/Sprite).

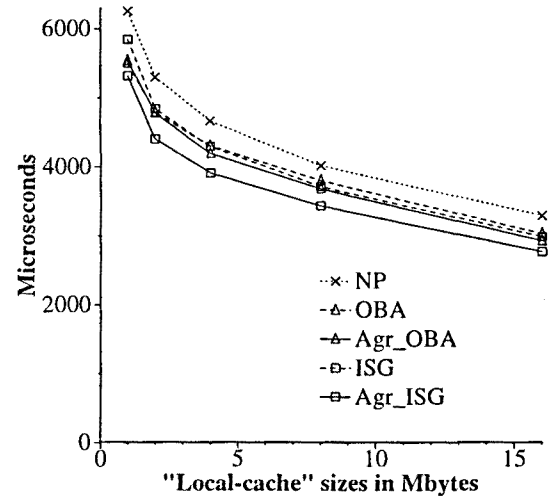

Figure 7.8 Average read operation time obtained by the presented prefetching algorithms on xFS (NOW/Sprite).

	PAFS					xFS				
	1MB	2MB	4MB	8MB	16MB	1MB	2MB	4MB	8MB	16MB
None	30.06	25.09	22.18	18.91	14.63	31.27	25.40	21.85	18.27	14.18
OBA	25.32	21.72	19.53	16.76	12.89	26.20	21.82	19.17	16.44	12.15
ISG	24.76	21.11	18.43	16.61	12.28	28.09	21.99	19.00	15.84	11.78
Agr_OBA	24.79	20.72	18.83	16.64	12.56	26.93	22.42	18.88	16.05	11.80
Agr_ISG	21.06	18.13	16.31	14.07	10.98	24.84	19.14	16.60	14.00	10.36

Table 7.2 Global miss ratio obtained by the presented prefetching algorithms (NOW/Sprite).

In this experiment, we also observe that when the aggressive algorithms are not implemented in a linear way, their performance is not as good as if they were implemented in a linear way. This is, once more, due to the cache floods and the excessive load placed on the managers and disk during some periods of time.

	PAFS					xFS				
	1MB	2MB	4MB	8MB	16MB	1MB	2MB	4MB	8MB	16MB
OBA	-1.38	-0.29	-1.47	4.68	1.42	1.44	-0.39	-0.24	0.38	0.31
ISG	22.98	12.21	6.79	4.77	-6.30	32.06	6.99	2.32	2.67	2.63
Agr_OBA	19.73	5.42	4.01	-5.58	-7.99	10.25	7.50	6.69	5.31	4.18
Agr_ISG	43.41	15.76	1.11	-3.99	-6.28	38.10	19.71	12.47	9.40	10.05

Table 7.3 Variation in disk traffic due to the presented prefetching algorithms (PM/CHARISMA).

7.5.2 Disk Traffic

After concluding that linear-aggressive prefetching increases the system performance, it is necessary to study the impact this aggressive prefetching has on the disk traffic. In this section, we present the disk-traffic variation observed when aggressive prefetching is done compared to the traffic observed when there is no prefetching.

Parallel Machine and CHARISMA

Table 7.3 presents the variation in the disk traffic produced by the different prefetching policies when the CHARISMA trace file was simulated. Each cell represents the percentage of traffic added by the prefetching policy when compared to the non-prefetching execution. If this number is positive, the prefetching algorithm increases the disk traffic while a negative percentage means that the disk traffic has decreased due to the prefetching algorithm. It is important to have in mind that the disk traffic is led by write operations and that doubling the number of reads does not have a great impact on the overall disk traffic. Only around a 10% of the disk traffic is done to fetch blocks from the disk in the version with no prefetching.

The first thing that can be observed is that OBA does not increase the disk traffic. This happens because this prefetching algorithm is very conservative and only prefetches one block for each user operation. As most of these prefetched blocks are used sooner or later, the disk traffic is not increased.

Prefetching algorithm	Physical writes per block
No Prefetching	5.959
OBA	5.930
ISG	3.419
Agr_OBA	3.514
Agr_ISG	3.263

Table 7.4 Average number of times a block is sent tot he disk during its life in the cache.

If we examine the increase of disk traffic made by ISG, we observe that it is nearly as high as the one found in the aggressive version. This happens because the number of blocks prefetched by this algorithm depends on the number of blocks requested by the user in each operation. As in the CHARISMA trace file there are many large requests, this algorithm prefetches many blocks and behaves as a quite aggressive algorithm and thus it has a similar amount of disk traffic as the aggressive algorithms.

We can also observe that the disk traffic produced by the aggressive algorithms is much higher in xFS than in PAFS when medium-size and large caches are used. This happens because of the linear-aggressive algorithms. As prefetching algorithms cannot be made linear in xFS, many more blocks are prefetched and the disk traffic increases. If we examine the results obtained with 1MBytes caches, it seems that this trend is not followed. But if we examine the absolute disk-traffic values observed in this case, we observe that the traffic is actually much higher in xFS than in PAFS. This miss-leading percentage appears because of the higher disk traffic observed in xFS when no prefetching is done. This higher traffic is due to the larger number of misses obtained by xFS when compared to the misses found under PAFS (explained in Chapter 5). As the reference point of the percentage is very different in xFS and in PAFS, these two columns cannot be compared.

Finally, the most surprising result appears when examining the variation in disk traffic made by the aggressive algorithms when PAFS has large caches. We can see that the disk traffic decreases when aggressive prefetching is done. This is completely anti-

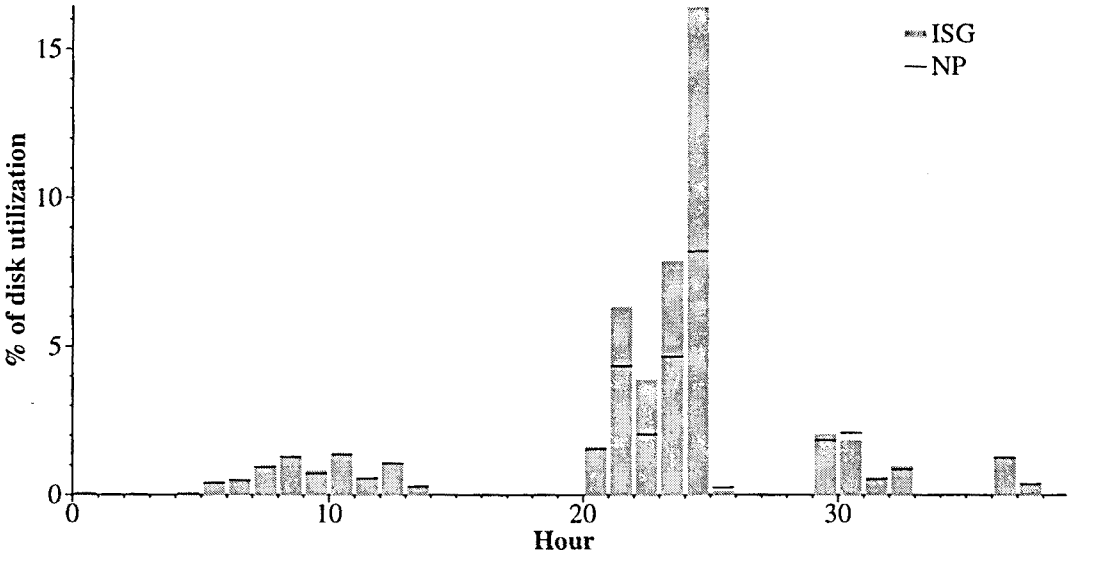

Figure 7.9 Load placed on the disks by ISG with a 1MByte local cache.

intuitive. As many more blocks are read, the number of disk blocks should not decrease but increase. Actually, the number of blocks fetched increases. What happens is that, in these cases, the number of disk writes decreases. This behavior can be easily explained. There are many blocks that are constantly being modified during their life in the cache. These blocks are sent to the disk periodically by the `syncer`. If all file-system operations between each modification are done faster, the period of time that this block is being modified decreases. This implies that the `syncer` has less opportunities to send this block to the disk and the number of physical writes also decreases. In order to support this idea, we present table 7.4 that shows the average number of times a block is sent to disk during its life time. We can observe that blocks are written to the disk less times when aggressive algorithms are used. This result does not only affect prefetching but any mechanism that speeds up the file system operations. The faster the operations are, the more probable the above situation will be and this will decrease the write traffic. This is not detected in xFS because many more blocks are prefetched increasing the disk traffic and because the operations are not as fast as in PAFS.

As the increase of disk traffic is quite significant when small caches are used, it seems interesting to study the real load placed on the disks. Figure 7.9 presents the percentage of disk utilization made by PAFS with ISG and 1MByte local caches (solid bars). This

	PAFS					xFS				
	1MB	2MB	4MB	8MB	16MB	1MB	2MB	4MB	8MB	16MB
OBA	-3.28	-4.36	-3.29	-4.10	-4.02	-2.59	-3.13	-3.18	0.61	-4.41
ISG	-1.59	-2.52	-3.02	-2.43	-3.61	2.99	0.00	-0.93	1.75	-3.37
Agr_OBA	3.81	1.77	0.85	-0.11	-1.04	2.97	4.71	4.18	6.04	-0.95
Agr_ISG	3.79	2.36	0.69	-0.47	-0.18	12.77	8.19	6.15	7.36	1.24

Table 7.5 Variation in disk traffic due to the presented prefetching algorithms (NOW/Sprite).

utilization is compared to the one obtained by the same system without prefetching (horizontal lines). We can see that during the most loaded hour, the disk traffic was only around a 16% of the possible with ISG and only 8% with no prefetching. This shows that linear-aggressive prefetching does not generally overload the disks while, at the same time, prefetches aggressively enough to greatly improve the system performance.

Network of Workstations and Sprite

If we perform the same study with the Sprite traces running on a network of workstations we obtain the results presented in Table 7.5.

The first thing we observe is that traffic is increased less than in the previous study. This happens because of the characteristics of the Sprite workload. In this workload, it is more difficult to prefetch blocks as was explained in Section 7.5.1. This fewer number of prefetched blocks leads to a smaller increase in the disk traffic if compared to the one observed in the CHARISMA study.

ISG behaves much more like OBA than in the previous study. This is because this algorithm is much less aggressive with the Sprite trace file than with the CHARISMA one. This happens because, in this study, the size of the user requests is smaller than in the previous one.

The rest of the results observed in this workload are very similar to the ones observed in the CHARISMA study and the same explanations can be applied.

7.6 CONCLUSIONS

The first conclusion that can be extracted from this work is that aggressive prefetching is a good idea. It increases the system performance specially when caches are big as it happens in cooperative caches.

In this chapter, we have also observed that prefetching with no limit may end up in a degradation of the system performance. For this reason, we proposed a way to limit the aggressiveness of the prefetching algorithms by only allowing a single prefetch for each file at a time. This has proven to be a good mechanism to limit the aggressiveness of prefetching and has achieved very good performance results. Anyway, this limitation in the parallelism of the prefetching might be too restrictive and a more flexible limitation should also be studied. There are environments where 2 or 3 lines of prefetch could obtain better performance results.

We have also presented a way to make linear-aggressive prefetching algorithms from simple and conservative ones. As this kind of prefetching algorithms obtained quite good performance results, it seems interesting to try to apply the same mechanism to more sophisticated algorithms to obtain a more intelligent aggressive prefetching.

Finally, we have shown that linear-aggressive prefetching does not necessarily overload the disk. As it is not too aggressive, the number of disk accesses is not excessive. Furthermore, it even decreases the disk traffic as it avoids some of the writes done by the `syncer`. Actually, this result is not only an effect of aggressive prefetching but should also appear with any mechanism that increases the file-system performance. It is important to notice that this effect could not have been detected using the old methodology as a speed up in the file system could not affect the distribution of the requests.

8

CONCLUSIONS AND FUTURE WORK

8.1 CONTRIBUTIONS OF THIS WORK

In this section, we describe the main contributions made in this thesis. These contributions have been achieved in three different fields. First, a novel approach to design cooperative caches has also been presented. Second, we have also contributed with a new mechanism that implements aggressive prefetching. The last contribution is the methodology used as a new way of simulation has been proposed.

8.1.1 Cooperative Caching

New Approach

The most important contribution made in this work is the new approach followed in the design of cooperative caches. So far, these kind of caches were designed following a common key issue. All of them relied on the physical locality of data to obtain high performance caches. The idea was to place the cached blocks in the local cache of the node that would request them. This was done because accessing a local cache is faster that accessing a remote one.

We propose that basing the performance on the physical locality is not the only way to achieve high-performance cooperative caches. Actually, we have designed a cooperative cache that does not exploit this locality and achieves a higher performance than traditional cooperative caches. The main problem found when exploiting physical locality is that keeping the blocks in the node that will need them introduces a significant overhead. This overhead is not outweighed by the benefits of this locality and ends up in a performance degradation.

This does not mean that physical locality is not a good idea. Of course that increasing it will increase the system performance as long as it does not introduce an additional overhead. For this reason, our system tries to achieve locality only when it does not add any overhead to the system.

The experimental results presented in this work show that this new approach obtains very efficient cooperative caches. Furthermore, it also simplifies the design and implementation allowing a more reliable system.

Cache Coherence

Replication has been traditionally used to increase the locality of the distributed systems. As the design approach used in this thesis does not stress locality, there is no need for replication. A very interesting side effect of the lack of replication is that all co-

herence problems disappear. If there is only one copy of the cached blocks, these blocks are always the up-to-date copy and there is no need to implement any cache-coherence mechanisms.

In our system, replication is not allowed and the cache-coherence problem is avoided. Avoiding this problem simplifies the cache design because we get rid of the coherence mechanisms which are quite complicated and time consuming. Furthermore, it increases the reliability of the file system because it is much simpler and easier to debug than traditional systems.

This way of solving the coherence problem does not have a performance penalty as we have already shown that not exploiting locality is a valid approach to design high-performance cooperative caches.

Fault-tolerance Mechanism

The last contribution done in the field of cooperative caches is the design of a parity-based fault-tolerance mechanism. This mechanism is able to handle the failure of a node with no loss of information. Once a node fails, the system can reconstruct all lost blocks using the parity information kept by the system. Once all blocks have been reconstructed and sent to the disk, the system continues its execution as if nothing had happened. This mechanism only cares about data. If a given file-server fails, the meta-data information kept by this server is lost and the system execution will have to be aborted.

In order to design a flexible system, we have defined three tolerance levels. Each level has different assumptions and offers a different degree tolerance. The higher the tolerance degree is the higher the performance penalty will be.

8.1.2 Prefetching

Linear Aggressive Prefetching

Finally, we have done some contributions in the field of file prefetching. In the previous study, we noticed that cooperative caches offer very large caches and that these caches usually keep some blocks that are very old. This led us to believe that aggressive prefetching could be used to increase the system performance quite easily. As very old blocks are kept in the cache, they can be replaced by the predicted ones and if the prediction was wrong no much difference should be noticed in the overall system performance.

As caches are very big, we can implement quite aggressive prefetching polices. On the other hand, if this prefetching is too aggressive the system may end up in a performance degradation. For this reason, we have designed the linear-aggressive prefetching. This mechanism converts a traditional conservative algorithm in a new one that is aggressive but intelligent enough to not flood the cache. It basically consists of prefetching blocks continuously but with the limitation that only one block can be fetched per file at a time. This mechanism has been tested with two simple prefetching algorithms and the experimental result show that the file system performance increases significantly.

In the experimental section, we have observed that this prefetching mechanism is not only valid for large cooperative caches. It can also be used with traditional caches of medium size and achieve quite good performance results. Anyway, the bigger the cache is, the better the algorithm will behave.

Finally, we have observed a very anti-intuitive behavior of the aggressive prefetching algorithms. We have observed that these kind of algorithms do not necessarily increase the disk traffic and that they may even decrease it. This happen because speeding up the file system operations decreases the number of write operations done by the `syncer`.

8.1.3 Methodology

So far, all research done in file-system simulation has been done using traces that keep the absolute time for each file-system event. This was done because it maintains the same load distribution as the one measured in the original system. Furthermore, the usage of this trace-file format eases the task of implementing the simulator as it does not need to simulate any process scheduling policy.

The problem with this methodology is that some side effects cannot be measured. For instance, if file-system operations are done faster, the applications will be able to make more requests to the system in the same period of time. This variation in the load distribution cannot be measured using the old methodology. Another problem is that the traces are not independent of the file systems they were taken from. The absolute time between events includes the CPU consumed by the application and the time needed by the file system to perform the operation. This time should not be present in the trace as the new file system will need a different amount of time for each operation. As there is no way to know the amount of time used by the system, it is not possible to take it away from the trace file and it adds some noise to the results obtained.

Our proposal consists of building the traces using relative times. The trace should only contain file-system events and the CPU time consumed by the applications between these events. It is important to notice that this CPU time only contains the time consumed by the application. This leaves out of the trace the time needed by the file system to perform the current event. This makes the traces more independent of the system they were taken from. It also allows a better simulation of the applications that make the I/O operations.

This new methodology adds some overhead to the simulations as a new resource has to be handled. Now, the sharing of the CPU has to be taken into account by the simulator. We believe that this overhead is not important if compared to the higher degree of accuracy achieved by the simulations.

8.2 FUTURE WORK

The work described in this dissertation has addressed many of the basic questions about building parallel/distributed file systems with a cooperative cache that has no coherence problems, but it leaves some questions unanswered and raises several new issues. This section describes some of the future work needed to address the unresolved problems and to increase the knowledge of the new issues.

- Detailed definition of a new trace format and simulation methodology. A wide set of traces with this new format should also be gathered.
- Study the benefits of integrating our cooperative cache with a log-structured file system.
- Modify the replacement algorithm using the lessons learned from Segmented-LRU.
- Relax the condition that the network and all the nodes are secure.
- Increase the scope of the fault-tolerance mechanism.
- Test the linear-aggressive prefetching on more intelligent algorithms.
- Study a possible increase in the number of blocks that can be prefetched in parallel from a single file.

New Trace Format and Simulation Methodology

After translating the traces to the new format, we reached the conclusion that new mechanisms to obtain traces should be defined. To be able to make better file-system simulations, the traces should have information on the application that made the requests. Knowing which node did which operations is not enough if a system that uses such information is to be designed and tested. This information has been traditionally taken away from the trace for privacy reasons and a way should be found so that this information is offered to the researchers while still maintain the privacy of the users. It is also important to extract the effects of the original system from the trace. The interval between two file system operations should not include the time needed by the

system to perform any of these events. Finally, we should collect a significant set of traces so that the new file-system researchers can use them to design better systems.

As the trace format is modified, a new simulation methodology is also needed. When using this trace format, the simulation will have to take into account the processor sharing between applications. It will also need to find a way to avoid race conditions or a way to measure their effect.

Integration with Log-structured File System

As cooperative caches offer such large caches, the disk traffic is led by write operations as most of the reads are handled by the cache. This is one of the basic assumptions made in the design of the log-structured file system. For this reason, it seems interesting to study the possibility of changing the *Unix*-like file system used in PAFS by a log-structured one.

Integration with Segmented-LRU

In this work, we have based our replacement algorithm on the well-known LRU because it is simple and easy to understand. Furthermore, it is the replacement algorithm most widely used in comertial file systems. Anyway, it seems interesting to create a new version of PG-LRU based on Segmented-LRU and try to take advantage of the benefits this algorithm has.

Security Issues

In the current design, all nodes are trusted ones. The system can cache file blocks in any of them knowing that the information will not be seen nor modified by any of the nodes. If the system is to be used in a very wide environment, this assumption may become too restrictive. We should find a way to allow remote caching even if a given node is not secure. This could be done by encrypting the file blocks and using a checksum for each block. A similar mechanism could also be used to solve the possible problems of an insecure network.

Increase the Scope of the Fault-tolerance Mechanism

Allowing the failure of file servers was out of the scope of this work as we only wanted to avoid losing information. This kind of failures should also be handled to offer a higher degree of fault tolerance. This could be solved by replicating the servers or by keeping a log with all the information needed to start a server to replace the failed one.

Linear-aggressive Prefetching

More work has to be done in the field of aggressive prefetching. We should test the linear-aggressive mechanism with more intelligent algorithms such as the ones based on Markov chains proposed by Vitter et al. [CKV93]. This should produce very intelligent aggressive algorithms that only prefetch the blocks needed by the applications.

We have also shown that one line of prefetching obtains good performance results and that many lines decrease the performance. We should study if more than one prefetching line can be used and which parameters affect this limitation.

A

xFS AND N-CHANCE FORWARDING

A.1 INTRODUCTION

xFS is the first example of a new paradigm for network file-system design, *serverless network file systems.* While traditional network file systems rely on a central server machine, a serverless system utilizes workstations cooperating as peers to provide all file system services. Any machine in the system can store, cache or control any block of data. This approach uses this location independence, in combination with fast local area networks, to provide better performance and scalability than traditional file systems. Furthermore, because any machine in the system can assume the responsibilities of a failed component, this serverless design also provides high availability via redundant data storage.

As the scope of both file systems (xFS and PAFS) is not equal, we will only describe the issues of xFS that are relevant to our work. We will explain the kind of servers and the

way they cooperate. We will also describe the cooperative caching mechanisms in detail. And finally, we will explain the most important differences in the approaches followed by both file systems and the modifications needed to implement xFS on DIMEMAS. Further information about xFS may be found in the literature [ADN+95, DWAP94, Dah95].

A.2 N-CHANCE FORWARDING: A COOPERATIVE CACHE

This cooperative cache is based on the idea that the cache of each node (or client if using the terminology used in xFS papers) is divided in two parts. The first part is locally managed by this client and it is the client that decides the information kept in it. The second part is used to build a globally coordinated cache. In this global cache the system keeps those block that are not specially needed by any node but that may increase the system effectiveness. This does not mean that a given node may not access the blocks kept in the locally coordinated cache of another node, it only means that it will not be able to decide the contents of this local part of the cache.

N-Chance Forwarding adjusts the fraction of each client's cache that is managed cooperatively depending on the client activity. N-Chance Forwarding recognizes that singlets are more globally valuable than duplicates, so it preferentially caches singlets. A singlet is defined as the only cached copy of a file block. N-Chance uses a randomized load balancing rather than a global knowledge to distribute singlets across the global cache.

A client in the N-Chance algorithm always discards the locally least-recently used object from its cache when it makes space for new blocks. However, if the discarded item is a singlet, the client forwards the singlet to another client's cache rather than allow the last copy of the block to drop out of the cooperative cache. The client that receives the data adds the block to its LRU list as if it had recently referenced the block.

To limit the amount of memory consumed by old singlets, each block has a *recirculation count* that clients increment when forwarding singlets. Clients discard, rather than forward, singlets whose recirculation count reaches n. If a client references a local singlet, it resets the recirculation count to zero, and if a client references a remote recirculating singlet, the remote client discards the singlet after reseting the recirculation count to zero and forwarding it to the client that referenced it. Thus, an unreferenced singlet survives n cache lifetimes in the global cooperative cache, giving the algorithm its name. After many experiments the value of n proposed by the NOW team is 2.

Using the recirculation count, this algorithm provides a dynamic trade-off for each client cache's allocation between local data (being cached because the client referenced it) and global data (singlets being cached for the good aggregate system performance). Active clients will tend to force any global data sent to them out of their caches quickly as local references displace global data. Idle clients, in contrast, will tend to accumulate global blocks and hold them in memory for long periods of time.

An implementation of this algorithm must prevent a ripple effect where a block forwarded form one client displaces a block to another client and so on. Note that in the most common case, the displaced block is not a singlet, so no ripple occurs. However, to guard against the uncommon case, the system imposes a policy that prevents deep recursion from ever occurring: a client receiving a recirculating block is not allowed to forward a block to make space.When a client receives such a block, it uses a modified replacement algorithm, discarding its oldest duplicate. If the cache contains no duplicates, the client discards the oldest recirculating singlet with the fewest recirculating remaining.

A.3 xFS: GENERAL OVERVIEW

In order to implement a serverless file system where each node may do any of the tasks of the system, we need to define a set of servers (or functions) that have to be done in the nodes depending on the roles they want to assume. Any node in the system my have the following roles: client, manager, storage server and cleaner. Actually, a given

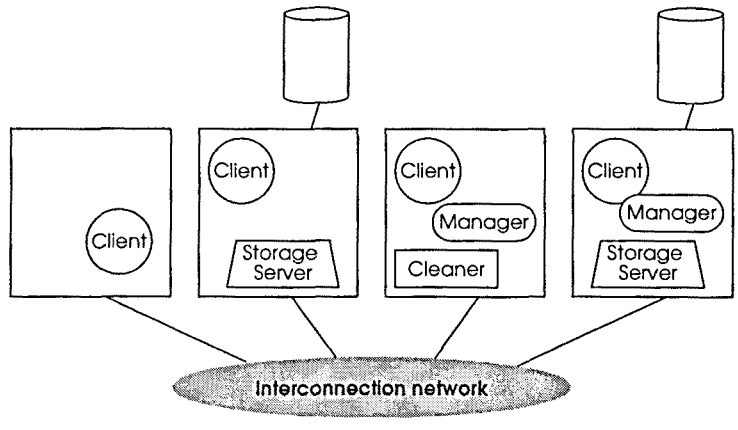

Figure A.1 Possible xFS configuration in a network of workstations.

node may assume one, or more than one, of these roles. Figure A.1 presents a possible configuration of a network of workstations with xFS running. In this example we can see a node that only acts as client and other nodes that have more than one of these functions.

Among the four possible roles, there are only three in which we are interested: clients, managers and storage servers. Client are in charge of handling file system requests from the applications running on their node. Managers are in charge of keeping track of the file system data and meta-data distributed in the whole system. They are also responsible for the consistency of the cached blocks. Finally, the storage servers are in charge of the physical placement of the data on the the disks. This last role has been simplified as we are not interested in simulating the stripe group mechanism.

A file-system operation starts when the application requests a block to the operating system (client). If the requested block is in the client's local cache, it is handled to the user. So far, there are no differences if compared to a regular *Unix* system. The differences appear when the client does not have the block cached. As this block may be cached in a remote node, the client contacts the block manager and requests this block. The manager checks if a copy of the block is already cached in any node. If such copy exists, the request is forwarded to the remote node holding the block. The kernel of this node will send the copy to the requesting node which will also cache it in its local cache. If there are no copies in the system, then that manger contacts a storage

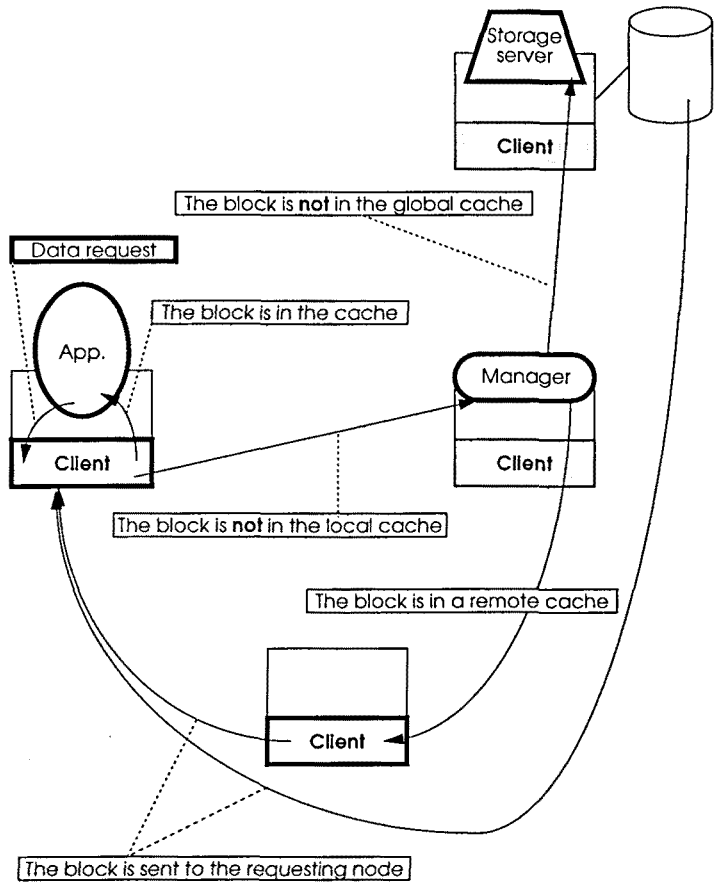

Figure A.2 Steps followed by xFS to fulfill a client request.

server which will access the block from the disk and will send it to the requesting node. Once the client obtains the block, either from a remote cache or the disk, it sends it to the application that requested it. A diagram of this process is presented in Figure A.2.

Each manager is responsible for a set of files. Clients know which manager is responsible of which files indexing the Manager-Map using the file-ID. This Manager-Map is a globally replicated table that keeps the information of which servers are responsible of which files. To increase the efficiency of the system, xFS tries to assign the files used by a client to a manager located on the same node. This is done using a policy named First-Writer. When a client creates a file, xFS chooses a manager located in the same node.

Managers implement a token-based cache-consistency scheme similar to Sprite [NWO88], Andrew [HKm+88], Spritely NFS [SM89], Coda [KS92], and Echo [BHJ+93] except that in xFS this consistency is managed on a per-block basis instead of using the file as the basic consistency unit [Bur88]. Before a client modifies a block, it must acquire write ownership of that block, so the client sends a message to the block's manager. The manager then invalidates any other cached copies of the block, updates its cache consistency information to indicate the new owner and replies to the client, giving permission to write. Once the client owns a block, the client can write the block repeatedly until some other client reads or writes the data, at which point the manager revokes ownership, forcing the client to stop writing the block.

A.4 COMPARISON BETWEEN COOPERATIVE-CACHE ALGORITHMS

In this section, we detail the most significant differences between xFS and the systems presented in this thesis.

The first 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.

Finally, only one level of fault-tolerance is implemented by xFS while PAFS offers three different levels of fault tolerance to adapt to the needs of each configuration.

A.5 MODIFICATIONS MADE IN OUR SIMULATIONS

As we are not using the real system but a simulation build on DIMEMAS, some modifications have been made to the original system. These modifications have been kept to a minimum as we wanted to compare with a system as similar to the original one as possible.

Most of the code implemented in xFS is placed inside the kernel and some operations can be handled locally without sending or receiving any messages. As our system could not simulate this in-kernel operations, we placed the file-system functions of the kernel in an out-of-kernel server leaving all the rest untouched. We also placed one such server in each node in order to minimize the impact of sending a message for each request. This was proved to be nearly irrelevant in the overall system performance [CGL97a, CGL96a].

The second difference was that only the relevant parts of the system were implemented. As we have already said, the scope of xFS is broader than the one of PAFS. This means that some of the features of xFS are not relevant when comparing both systems. In order to simplify the comparison, we have not simulated features like stripe groups, logs, cleaning mechanisms, etc. because they have nothing to do with the cooperative cache nor the load distribution mechanism studied in our work. Ignoring these features should not affect the results presented in this thesis as they are not specially related to the cooperative cache.

Finally, the last modification was the centralized version simulated to compare with PACA. This simplified version was only used in Chapter 4 to get an easier understanding of the cooperative cache mechanisms. Once understood, the distributed version of xFS has been used as it is.

REFERENCES

[ACPtNt95] ANDERSON, T. E., CULLER, M. D., PATTERSON, D. A., AND THE NOW TEAM. A case for NOW (networks of workstations). In *Proceedings of the Micro*, pages 54–64. IEEE Computer Society Press, 1995.

[ACR95] ARUNACHALAM, M., CHOUDHARY, A., AND RULLAN, B. A prefetching prototype for parallel file system on the Paragon. *ACM Measurement and Modeling of Computer Systems (SIGMETRICS)*, 23(1):321–323, 1995.

[ADN+95] ANDERSON, T. E., DAHLIN, M. D., NEEFE, J. M., PATTERSON, D. A., ROSELLI, D. S., AND WANG, R. Y. Serverless network file systems. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 109–126, December 1995.

[Adv93] ADVANCED MICRO DEVICES, INC. *Am29F040 Datasheet*, 1993.

[BAD+92] BAKER, M., ASAMI, S., DEPRIT, E., OUSTERHOUT, J., AND SETZER, M. Non-volatile memory for fast and reliable file system. In *Proceedings of the 5th Symposium on Architectura Support for Programming Languages and Operating Systems*, pages 10–22, October 1992.

[BBS+94] BENNET, R., BRYANT, K., SUSSMAN, A., DAS, R., AND SALTZ, J. Jovian: A framework for optimizing parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 10–20. IEEE Computer Society Press, October 1994.

[BdC93] BORDAWEKAR, R. R., DEL ROSARIO, J. M., AND CHOUDHARY, A. N. Design and evaluation of primitives for parallel I/O. In *Proceedings of the Supercomputing'93*, pages 452–461. IEEE Computer Society Press, November 1993.

[BHJ+93] BIRELL, A. D., HISGEN, A., JERIAN, C., MANN, T., AND SWART, G. The Echo distributed file system. Technical Report 111, Digital Research Center, September 1993.

[BHK+91] BAKER, M. G., HARTMAN, J. H., KUPFER, M. D., SHIRRIFF, K. W., AND OUSTERHOUT, J. K. Measurements of a distributed file system. In *Proceedings of the 13th Symposium on Operating System Principles*, pages 198–212. ACM Press, July 1991.

[BM93] BURKHARDT, W., AND MENON, J. Disk array storage system reliability. In *Proccedings of the 23rd Annual International Symposium on Faul-Tolerant Computing*, pages 432–441, 1993.

[Bur88] BURROWS, M. *Efficient Data Sharing.* PhD thesis, Cambridge University, 1988.

[Cao96] CAO, P. *Application Controlled File Caching and Prefetching.* PhD thesis, Princeton University, January 1996.

[Car95] CARRETERO, J. *Un sistema de ficheros paralelo con coherencia de cache para multiprocesosadores de proposito general.* PhD thesis, Universidad Politecnica de Madrid, 1995.

[CBF93] CORBETT, P. F., BAYLOR, S. J., AND FEITELSON, D. G. Overview of the Vesta parallel file system. *ACM Computer Architecture News*, 21(5):7–15, 1993.

[CBM+94] CHOUDHARY, A., BORDAWEKAR, R., M.HARRY, KRISHNAIYER, R., PONNUSAMY, R., SINGH, T., AND THAKUR, R. PASSION: Parallel and scalable software for input-output. Technical Report SCCS-636, NPAC, September 1994.

[CF96] CORBETT, P. F., AND FEITELSON, D. G. The Vesta parallel file system. *ACM Transaction on Computer Systems*, 14(3):225–264, August 1996.

[CFF+95] CORBETT, P. F., FEITELSON, D., FINERBERG, S., HSU, Y., NITZBERG, B., PROST, J. P., SNIR, M., TRAVERSAT, B., AND WONG, P. Overview of the MPI-IO parallel I/O interface. In *Proceedings of the*

3rd Workshop on I/O in Parallel and Distributed Systems, pages 1–15, April 1995.

[CFKL95] CAO, P., FELTEN, E. W., KARLIN, A. R., AND LI, K. A study of integrating prefetching and caching strategies. *Proceedings of the SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1995.

[CFL94a] CAO, P., FELTEN, E. W., AND LI, K. Application-controlled file caching policies. In *Proceedings of the Summer Technical Conference*. USENIX Association, June 1994.

[CFL94b] CAO, P., FELTEN, E. W., AND LI, K. Implementation and performance of application-controlled file caching. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, pages 165–177. USENIX Association, November 1994.

[CGL95] CORTES, T., GIRONA, S., AND LABARTA, J. PACA: A distributed file system cache for parallel machines. performance under unix-like workload. Technical report, Universitat politècnica de Catalunya, Departamanet d'Arquitectura de Computadors, 1995.

[CGL96a] CORTES, T., GIRONA, S., AND LABARTA, J. I/O performance of scientific-parallel applications under PAFS. Technical Report UPC-CEPBA-RR-96/23, Universitat politècnica de Catalunya, Centre Europeu de Paral.lelisme de Barcelona, 1996.

[CGL96b] 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*, pages I:477–486, August 1996.

[CGL97a] CORTES, T., GIRONA, S., AND LABARTA, J. Avoiding the cache-coherence problem in a parallel/distributed file system. In *Proceedings of the High Performance Commputing and Networking Conference*, pages 860–869, April 1997.

[CGL97b] CORTES, T., GIRONA, S., AND LABARTA, J. Design issues of a cooperative cache with no coherence problems. In *Proceedings of the 5th I/O in Parallel and Distributed Systems Workshop*, November 1997.

[CKV93] CUREWITZ, K. M., KRISTIAN, P., AND VITTER, J. S. Practical prefetching via data compression. In *Proceedings of the SIGMOD Management of Data*, pages 257–266. ACM Press, May 1993.

[CL91] CABRERA, L.-F., AND LONG, D. D. E. Swift: Using distributed disk striping to provide high I/O data rates. *Computing Systems*, 4(4):405–436, Fall 1991.

[CLG+94] CHEN, P. M., LEE, E. K., GIBSON, G. A., KATZ, R. H., AND PATTERSON, D. A. RAID: High-performance and reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, 1994.

[Com94] COMPUTER SYSTEMS RESEARCH GROUP, UNIVERSITY OF CALIFORNIA AT BERKELEY. *4.4BSD-Lite Source CD-ROM.* USENIX Asociation and O'Reilly & Associates, 1994.

[CPd+96] CARRETERO, J., PEREZ, F., DE MIGUEL, P., GARCIA, F., AND ALONSO, L. A massivelly parallel and distributed I/O subsystem. *Computer Architecture News*, 24(3):1–8, 1996.

[CPdA96] CARRETERO, J., PEREZ, F., DE MIGUEL, P., AND ALONSO, L. ParFiSys: A parallel file system for MPP. *Operating System Review*, 30(2):74–80, April 1996.

[CS92] CARSON, S., AND SETIA, S. Optimal write batch size in log-structured file systems. In *Proceedings of the File System Workshop.* USENIX Association, 1992.

[Dah95] DAHLIN, M. D. *Serverless Network File System.* PhD thesis, University of California at Berkeley, 1995.

[Dah96] DAHLIN, M. D. Personal comunication. 1996.

[dBC93] DEL ROSARIO, J. M., BORDAWEKAR, R. R., AND CHOUDHARY, A. N. Improved parallel I/O via two-phase run-time access strategy. *ACM Computer Architecture*, 21(5):31–39, 1993.

[DS89] DIBBLE, P. C., AND SCOTT, M. L. Beyond striping: The bridge multiprocessor file system. *ACM SIGARCH*, 15(15):32–39, 1989.

[DSH+94] DRAPEAU, A. L., SHIRRIF, K. W., HARTMAN, J. H., MILLER, E. L., SESHAN, S., KATZ, R. H., LUTZ, K., PATTERSON, D. A., LEE, E. K., CHEN, P. H., AND GIBSON, G. A. RAID-II: A high-bandwidth network file server. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 234–244, 1994.

[DWAP94] 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 *Proceedings of the 1st Symposium on Operating System Design and Implementation*, pages 267–280. USENIX Association, November 1994.

[FBD96] FREEDMAN, C. S., BURGER, J., AND DEWITT, D. J. SPIFFI – a scalable parallel file system for the Intel Paragon. *IEEE Transactions on Parallel and Distributed Systems*, 7(11):1185–1200, November 1996.

[FCNL94] FEELEY, M. J., CHASE, J. S., NARASAYYA, V. R., AND LEVY, H. M. Integrating coherency and recoverability in distributed systems. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, pages 215–227. USENIX Association, November 1994.

[FMP+95] 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 *Proceedings of the 15th Symposium on Operating Systems Principles*, December 1995.

[FO71] FEIERTAG, R. J., AND ORGANICK, E. I. The Multics input/output system. In *Proceedings of the 3rd Symposium on Operating System Principles*, pages 35–41, October 1971.

[GA93] GRIFFIOEN, J., AND APPLETON, R. Automatic prefetching in a WAN. In *Proceedings of the Workshop on Advances in Parallel and Distributed Systems*, pages 8–12. IEEE Computer Society Press, October 1993.

[GA94] GRIFFIOEN, J., AND APPLETON, R. Reducing file system latency using a predictive approach. In *Proceedings of the Summer Technical Conference.* USENIX Association, 1994.

[Gar96] GARCIA, F. *Coherencia de Caches en Sistemas de Ficheros para Entornos Distribuidos y Parallelos.* PhD thesis, Universidad Politecnica de Madrid, 1996.

[GBDJ93] GEIST, A., BEGUELIN, A., DONGARRA, J., AND JIANG, W. *PVM3 User's Guide and Reference manual.* Oak Ridge National Laboratory, May 1993.

[GC89] GRAY, C. G., AND CHERITON, D. R. Leases: An efficiente fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th Synposium on Operating Systems Principals.* ACM Press, December 1989.

[GCP+96] GARCÍA, F., CARRETERO, J., PÉREZ, F., DE MIGUEL, P., AND ALONSO, L. Coherencia de cache en sistemas de ficheros paralelos. In *Proceedings of the VII jornadas de paralelismo*, pages 215–227, September 1996.

[Gil96] GILLET, R. B. Memory channel network for PCI. *IEEE Micro*, February 1996.

[GL91] GRIMSHAW, A. S., AND LOYOT, E. C. Elfs: Object-oriented extensible file systems. Technical Report TR-91-14, University of Virginia, 1991.

[Gra96] GRAPHICS, S. Origin servers. technical overview of the origin family,. http://www.sgi.com/Products/hardware/servers/technology/overview .html, 1996.

[GSC+95] GIBSON, G. A., STODOLSKY, D., CHANG, F. W., COURTRIGHT, W. V., DEMETRIOU, C. G., GINTING, E., HOLLAND, M., MA, Q., NEAL, L., PATTERSON, R. H., SU, J., , YOUSSEF, R., AND ZELENKA, J. The Scotch parallel storage systems. In *Proceedings of the 40th IEEE Computer Society International Conference (Compcon'95)*, pages 403–410. IEEE Computer Society Press, Spring 1995.

[Har93] HARTMAN, J. H. Using the Sprite file system traces. Technical report, University of california at Berkeley, 1993.

[HdC95] HARRY, M., DEL ROSARIO, J. M., AND CHOUDHARY, A. The design of VIP-FS: A virtual and parallel file system for high performance parallel and distributed computing. *Operating System Review*, 29(3):35–48, 1995.

[Hel93] HELLWAGNER, H. Design considerations for scalable parallel file systems. *The Computer Journal*, 36(8):741–755, 1993.

[HER+95] HUBER, J. V., ELFORD, C. L., REED, D. A., CHIEN, A. A., AND BLUMENTHAL, D. S. PPFS: A high performance portable parallel file system. In *Proceedings of the 9th International Conference on Supercomputing*, pages 385–394. ACM Press, July 1995.

[Hig93] HIGH PERFORMANCE FORTRAN FORUM. High-performance fortran language specification. version 1.0. *Scientific Programming*, May 1993.

[HKm+88] HOWARD, J., KAZAR, M. L., MENEES, S. G., NICHOLS, D. A., SATYANARAYANAN, M., SIDEBOTHAM, R. N., AND WEST, M. J. Scale and performance in a distributed file system. *ACM Transaction on Computer Systems*, 10(1):51–81, February 1988.

[HO92] HARTMAN, J. H., AND OUSTERHOUT, J. K. Zebra: A striped network file system. In *Proceedings of the File System Workshop*, pages 71–78. USENIX Association, 1992.

[HO95] HARTMAN, J., AND OUSTERHOUT, J. K. The zebra striped network file system. *Transactions on Computer System*, 13(3):274–310, 1995.

[HY96] HU, Y., AND YANG, Q. DCD - disk caching disk: A new approach for boosting I/O performance. *Computer Architecture News*, 24(2):169–178, 1996.

[Int94] INTEL CORPORATION. *Flahs Memory*, 1994.

[Jai91] JAIN, R. *The art of computer systems performance analysis.* Wiley Professional Comnputing, 1991.

[Kat92] KATZ, R. H. Network-attached storage systems. In *Proceedings of the Scalable High Performance Computing Conference*, pages 68–75, 1992.

[Kaz88] KAZAR, M. L. Synchronization and caching issues in the Andrew file system. In *Proceedings of the Summer Technical Conference*. USENIX Association, 1988.

[KE90] KOTZ, D., AND ELLIS, C. S. Prefetching in file systems for MIMD multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):218–230, April 1990.

[KE93] KOTZ, D., AND ELLIS, C. S. Caching and writeback policies in parallel file systems. *Journal of Parallel and Distributed Computing*, 17:140–145, 1993.

[Kim97] KIMBREL, T. *Parallel Prefetching and Caching*. PhD thesis, University of Washington, 1997.

[KK96] KIMBREL, T., AND KARLIN, A. R. Near-optimal parallel prefetching and caching. In *Proceedings of the Symposium on Foundations of Computer Science*. IEEE Computer Society Press, 1996.

[KL96] KROEGER, T. M., AND LONG, D. D. E. Predicting the future file-system actions from prior events. In *Proceedings of the Annual Technical Conference*. USENIX Association, January 1996.

[KLW94] KAREDLA, R., LOVE, J. S., AND WHERRY, B. G. Caching strategies to improve disk system performance. *IEEE COMPUTER*, pages 38–46, March 1994.

[KM95] KON, F., AND MANDEL, F. Soda: A lease-based consistent distributed filesystem. In *Proceedings of the 13th Brazilian Symposium on Cpomputer Networks*, 1995.

[KN93] KHALIDI, Y. A., AND NELSON, M. N. Extensible file systems in Spring. Technical Report SMLI TR-93-18, Sun Microsystems Labs, September 1993.

[KN94] KOTZ, D., AND NIEUWEJAAR, N. Dynamic file-access characteristics of a production parallel scientific workload. In *Proceedings of the Supercomputing'94*, pages 640–649. IEEE Computer Society Press, November 1994.

[KNM95] KAWAGUCHI, A., NISHIOKA, S., AND MOTODA, H. A flash-memory based file system. In *Proceedings of the Annual Technical Conference.* USENIX Association, January 1995.

[Kot91] KOTZ, D. *Prefetching and Caching Techniques in File Systems for MIMD Multiprocessors.* PhD thesis, Duke University, Department of Computer Science, April 1991.

[Kot94] KOTZ, D. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1st Symposium on Operating System Design and Implementation*, pages 61–74. USENIX Association, November 1994.

[Kot95a] KOTZ, D. Expanding the potential for disk-directed I/O. Technical Report PCS-TR95-254, Dartmouth College, Computer Science, March 1995.

[Kot95b] KOTZ, D. Interfaces for disk-directed I/O. Technical Report PCS-TR95-270, Dartmouth College, Computer Science, September 1995.

[KS92] KISTLER, J., AND SATYANRAYANAN, M. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, 1992.

[KTP+96] KIMBREL, T., TOMKINS, A., PATTERSON, R. H., BERSHAD, B., CAO, P., FELTEN, E. W., GIBSON, G. A., KARLIN, A., AND LI, K. A trace-driven comparison of algorithms for parallel prefetching and caching. In *Proceedings of the 2nd International Symposium on Operating System Design and Implementation*, pages 19–34. USENIX Association, October 1996.

[Law81] LAWLOR, F. D. Efficient mass storage parity recovery mechanism. *IBM Technical Disclosure Bulletin*, 24(2):986–987, July 1981.

[LGP+92] LABARTA, J., GIMENEZ, J., PUJOL, C., JOVE, T., AND NAVARRO, J. I. PAROS: Operating system kernel for distributed memory paral-

lel machines. In PRES/CIMNE, I., editor, *Proceedings of the Parallel Computing and Transputer Applications*, pages 673–682, September 1992.

[LGP+96] 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*, pages II:665–674, August 1996.

[LMKQ89] LEFFLER, S. J., MCKUSICK, M. K., KARELS, M. J., AND QUARTERMAN, J. S. *The design and implementation of the 4.3 BSD UNIX Operating System.* Addison-Wesley, 1989.

[LWY93] 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):1185–1204, November 1993.

[LWY96] 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):191–206, February 1996.

[Mac94] MACKLEM, R. Not quite NFS, soft cache consistency for NFS. In *Proceedings of the Winter Technical Conference.* USENIX Association, 1994.

[MBH+93] MANN, T., BIRELL, A. D., HISGEN, A., JERIAN, C., AND SWART, G. A coherent distributed file cache with directory write-behind. Technical Report 103, Digital Research Center, June 1993.

[Mes94] MESSAGE PASSING INTERFACE FORUM. *MPI: A message-passing interface standard*, May 1994.

[Mes97] MESSAGE PASSING INTERFACE FORUM. *MPI-2: Extensions to the Message-Pasing Interface*, July 1997.

[MJLF84] MCKUSICK, M., JOY, W., LEFFLER, S., AND FABRY, R. A fast file system for unix. *ACM transactions on Computer Systems*, 2(3):181–197, August 1984.

[MK90] MCVOY, L., AND KLEIMAN, S. Extent-like performance from a unix file system. In *Proceedings of the Summer Technical Conference*, pages 137–144. USENIX Association, June 1990.

[MK95] MILLER, E. L., AND KATZ, R. H. RAMA: Easy access to high-bandwidth massively parallel file system. In *Proceedings of the Annual Technical Conference*, pages 59–70. USENIX Association, January 1995.

[MK97] MILLER, E. L., AND KATZ, R. H. RAMA: An easy-to-use, high-performance parallel file system. *Parallel Computing*, 23(4), 1997.

[Mog94] MOGU, J. C. A better update policy. In *Proceedings of the Summer Technical Conference.* USENIX Association, June 1994.

[MPI96] MPI-IO COMMITTEE. MPI-IO: A parallel file I/O interface for MPI. Version 0.5. Technical report, MPI-IO Committee, May 1996.

[Nel90] NELSON, M. N. Virtual memory vs. the file system. Technical Report 90/4, Digital Western Research laboratory, March 1990.

[NH82] NEEDHAM, R., AND HERBERT, A. *The Cambridge Distributed Computing System.* Addison Wesley, 1982.

[Nie96] NIEUWEJAAR, N. *Galley: A New Parallel File System For Scientific Workloads.* PhD thesis, Department of Computer Science, Dartmouth College, 1996.

[NK95] NIEUWEJAAR, N., AND KOTZ, D. Low-level interfaces for high-level parallel I/O. In *Proceedings of the 3rd Workshop for I/O Parallel and Distributed Systems*, pages 47–62, 1995.

[NKP+94] NIEUWEJAAR, N., KOTZ, D., PURAKAYASTHA, A., ELLI, C. S., AND BEST, M. L. File-access characteristics of parallel scientific workloads. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1075–1089, October 1994.

[NV77] NEVALAINEN, O., AND VESTERINEN, M. Determining blocking factors for sequential files by heuristics methods. *The Computer Journal*, 20(3):245–247, August 1977.

[NWO88] NELSON, M. N., WELCH, B., AND OUSTERHOUT, J. Caching in the Sprite network file systems. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[PC96] PARK, C., AND CHOE, T. Stripind disk array RM2 enabling the tolerance of double disk failures. In *Proceedings of Supercomputing*, 1996.

[PCG+97] PÉREZ, F., CARRETERO, J., GARCÍA, F., DE MIGUEL, P., AND ALONSO, L. Evaluating ParFiSys: A high-performance parallel and distributed file system. *The EUROMICRO Journal of Systems Architecture*, 43(8):533–542, May 1997.

[Pea92] PEACOCK, J. The counterpoint fast file system. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[PG94] PATTERSON, R. H., AND GIBSON, G. A. Exposing I/O concurrency with informed prefetching. In *Proceedings of the 3rd International Conference on Distributed Information Systems*, pages 7–16, 1994.

[PGG+95] PATTERSON, R. H., GIBSON, G. A., GINTING, E., STODOLSKY, D., AND ZELENKA, J. Informed prefetching and caching. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 79–95. ACM Press, 1995.

[PGK88] PATTERSON, D. A., GIBSON, G., AND KATZ, R. H. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the International Conference on Management of Data*, pages 109–116. ACM Press, 1988.

[Pie95] PIERCE, P. A concurrent file system for a highly parallel mass storage system. In *Proceedings of the 9th International Parallel Processing Symposium*, 1995.

[Pre92] PRESS, I. C. S. *IEEE Standard Portable Operating System Interface for Computer Environments*, 1992.

[RD90] ROBINSON, J. T., AND DEVARAKONDA, M. V. Data cache management using frequency-based replacement. In *Proceedings of the Conference on Measurements and Modeling of Computer Systems*, pages 134–142. ACM Press, May 1990.

[RL96] REED, B., AND LONG, D. D. Analysis of caching algorithms for distributed file systems. *Operating Systems Review*, 30(3):12–21, 1996.

[RO90] ROSENBLUM, M., AND OUSTERHOUT, J. K. The lfs storage manager. In *Proceedings of the Summer Technical Conference*, pages 315–324. USENIX Association, June 1990.

[RO92] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transaction on Computer Systems*, 10(1):26–52, February 1992.

[Rob96] ROBINSON, J. T. Analysis of steady-state segment storage utilizations in a log-structured file system with least-utilized segment cleaning. *ACM Operating System Review*, 30(4):29–32, 1996.

[Ros92] ROSENBLUM, M. *The Design and Implementation of a Log-Structured File System.* PhD thesis, University of California at Berkeley, June 1992.

[RT74] RITCHIE, D. M., AND THOMPSON, K. The unix time-sharing system. *Communication of the ACM*, 17(7):365–375, July 1974.

[SCW+95] SEAMONS, K. E., CHEN, Y., WINSLETT, M., CHO, Y., KUO, S., JONES, P., JOZWIAK, J., AND SUBRAMANIAM, M. Fast and easy I/O for arrays in large-scale applications. In *Proceedings of the 7th Symposium on Parallel and Distributed Computing (Workshop on Model ing and Specification of I/O)*. IEEE Computer Society Press, October 1995.

[SGN85] SCHROEDER, M., GRIFFORD, D., AND NEEDHAM, R. A caching file system for a programmer workstation. In *Proceedings of the 10th Symposium on Operating System Principles*, pages 25–34. ACM Press, 1985.

[SH96] SARKAR, P., AND HARTMAN, J. Efficient cooperative caching using hints. In *Proceedings of the 2nd International Symposium on Operating Systems Design and Implementation*, pages 35–46. USENIX Association, October 1996.

[SM89] SRINIVASAN, V., AND MOGUL, J. Spritely NFS: Experiences with cache consistency protocols. In *Proceedings of the 12th Symposium on Operating System Principles*, pages 45–57. ACM Press, 1989.

[Smi78] SMITH, A. J. Secuential program prefetching in memory hierarchies. *IEEE Computer*, pages 7–2, December 1978.

[Smi85] SMITH, A. J. Disc cache - miss ratio analisys and design considerations. *ACM transactions on Computer Systems*, 3(3):161–203, August 1985.

[SO94] SHIRRIFF, K., AND OUSTERHOUT, J. Sawmill: A high bandwidth logging file system. In *Proceedings of the Summer Technical Conference*, pages 125–136. USENIX Association, June 1994.

[SS96] SMITH, K. A., AND SELTZE, M. A comparison of ffs disk allocation policies. In *Proceedings of the Annual Technical Conference*. USENIX Association, January 1996.

[SSB+95] SELTZER, M., SMITH, K., BALAKRISHNAN, H., CHANG, J., MCMAINS, S., AND PADMANABHAN, V. File system logging versus clustring: A performance comparison. In *Proceedings of the Annual Technical Conference*, pages 249–264. USENIX Association, January 1995.

[SW96] SEAMONS, K. E., AND WINSLETT, M. Multidimensional array I/O in Panda 1.0. *The Journal of Supercomputing*, 10(2):191–211, 1996.

[Swe96] SWEENEY, A. Scalability in the XFS file system. In *Proceedings of the Annual Technical Conference*. USENIX Association, January 1996.

[Tri80] TRIVED, K. Optimal selection of CPU speed, device capabilities, and file assignments. *Journal of the ACM*, 27(3):457–473, July 1980.

[VIN+93] VERSO, S. J. L., ISMAN, M., NANOPOULOS, A., NESHEIM, W., MILNE, E. D., AND WHEELER, R. *sfs*: A parallel file system for the CM-5. In *Proceedings of the Summer Technical Conference*, pages 291–305. USENIX Association, 1993.

[VK96] VITTER, J. S., AND KRISHNAN, P. Optimal prefetching via data compression. *Journal of the ACM*, 43(5):771–793, September 1996.

[WGSS95] WILKES, J., GOLDING, R., STAELIN, C., AND SULLIVAN, T. The HP AutoRAID hierarchical storage system. In *Proceedings of the 15th Operating System Review*, pages 96–108. ACM Press, December 1995.

[WGSS96] WILKES, J., GOLDING, R., STAELIN, C., AND SULLIVAN, T. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.

[WMR+94] WHEAT, S. R., MACCABE, A. B., RIESEN, R., VAN DRRESSER, D. W., AND STALLCUP, T. Puma: An operating system for massively parallel systems. In *Proceedings of the 27th Hawaii ICSS*, 1994.

[WZ94] WU, M., AND ZWAENEPOEL, W. envy: A non-volatile, main memory storage system. In *Proceedings of the 6th International Conference on Architectural for Programming Languages and Operating Systems*, 1994.