
COOPERATIVE CACHING AND PREFETCHING IN PARALLEL/DISTRIBUTED FILE SYSTEMS

Antonio Cortés Rosselló

UPC. Universitat Politècnica de Catalunya

Departament d'Arquitectura de Computadors

Barcelona (Espanya)



Jesús Labarta Mancho

Advisor

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

COOPERATIVE CACHING AND PREFETCHING IN PARALLEL/DISTRIBUTED FILE SYSTEMS

Antonio Cortés Rosselló

UPC. Universitat Politècnica de Catalunya

Departament d'Arquitectura de Computadors

Barcelona (Espanya)



Jesús Labarta Mancho

Advisor

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

COOPERATIVE CACHING AND PREFETCHING IN PARALLEL/DISTRIBUTED FILE SYSTEMS

*Als meus pares, Jaume i Maria Lluïsa,
a la memòria del meu avi Toni
que probablement s'hagués sentit molt orgullós de viure aquest moment,
i molt especialment, a la Glòria.*

*To my parents, Jaume and Maria Lluïsa,
to the memory of my grandfather Toni
who would have probably been very proud to live this moment, and
very specially, to Glòria.*

ABSTRACT

If we examine the structure of the applications that run on parallel machines, we observe that their I/O needs increase tremendously every day. These applications work with very large data sets which, in most cases, do not fit in memory and have to be kept in the disk. The input and output data files are also very large and have to be accessed very fast. These large applications also want to be able to checkpoint themselves without wasting too much time. These facts constantly increase the expectations placed on parallel and distributed file systems. Thus, these file systems have to improve their performance to avoid becoming the bottleneck in parallel/distributed environments.

On the other hand, while the performance of the new processors, interconnection networks and memory increases very rapidly, no such thing happens with the disk performance. This lack of improvement is due to the mechanical parts used to build the disks. These components are slow and limit both the latency and the bandwidth of the disk. Thus, the performance of the file-system operations that have to access the disk is also limited. As the mechanical technology cannot keep the pace of the electronic one, the research community is in search of solutions that improve the file-system performance without expecting a significant improvement in the mechanical devices.

In this thesis, we propose a solution to this problem by decoupling the performance of the file system from the performance of the disk. We achieve it by designing a new cooperative cache and some aggressive-prefetching algorithms. Both mechanisms, decrease the number of times the file system has to access the slow disk in the critical path of the user request. Furthermore, the resources used in these solutions are large memories and high-speed interconnection networks which grow at a similar pace as the rest of the components in a parallel machine.

We propose a new approach to design cooperative caches. This kind of caches have traditionally based their performance on achieving a high physical locality. This means that the system tried to cache file blocks in the nodes where they were going to be used. Our thesis is that high-performance cooperative caches can be implemented without exploiting this locality. We will prove that focusing the design on the global effectiveness of the cache, the speed of remote operations and the simplification of the coherence mechanism ends up in a better and simpler cooperative cache.

We also propose a mechanism that converts traditional prefetching algorithms in aggressive ones that take advantage of the large sizes of the cooperative cache.

In this work, we have designed two cooperative caches using the proposed new approach. The first one has a centralized control and its main objective was to increase our knowledge about cooperative caches. With this design, we also wanted to prove that a centralized control is not a bad idea for small networks with tens of nodes. The second design is a distributed one and also includes a fault tolerance mechanism. In this second design, we prove that the new approach proposed in this thesis really achieves high-performance cooperative caches.

All the results presented in this thesis have been obtained through simulations so that a wide range of architectures can be studied. Furthermore, we have also used a new simulation methodology that allows more accurate simulations than the traditional one.

To test this work, we have compared our proposals with the state of the art in cooperative caches and parallel/distributed file systems. These comparisons have been done under two environments. The first one characterizes a parallel machine and the workload used in the one described in the CHARISMA project. The second one emulates a network of workstations using the Sprite trace files.

ACKNOWLEDGMENTS

I would like to thank Jesús Labarta, my thesis advisor, for his guidance of this dissertation and his support. He has taught me all I know about researching and his patience with me has been infinite (unfortunately, not like his free time).

I am specially grateful to Sergi Girona for many insightful discussions in the first stage of this work. I also want to thank him for his patience reading my manuscripts and making interesting comments which I did not always follow. Finally, I am in debt with him for all the modifications, and new features implemented in DIMEMAS without which this work could have never been done.

My gratitude also to Maite Ortega for her encouragement throughout all the time I have been working on this thesis. I also want to thank her for implementing the first prototype of the cooperative cache described in this work.

Among all the teachers I've had during my academic life, I am in debt with Nacho Navarro, Jordi Torres and Teo Jové because they were the first ones to show me what an operating system was. Since then, operating systems have become one of my greatest interests. Furthermore, I also have to thank Nacho Navarro for guiding my first steps in the task of researching.

I have a tendency to get very excited with a successful result and to get very upset when the results are not as successful. Roger Espasa has been the person who has always tried to balance my mood so that bad moments are not so bad. I thank him for this, although I am not sure about his success. I also thank him for being an excellent office-mate during these five years and for helping me every time I needed a counsel.

I also want to thank the system administrators from both LCAC and CEPBA and specially to Oriol Riu, Judit Jiménez and Rosa Castro for their excellent technical support. I also want to thank all of them for those wonderful and relaxing coffee breaks and for our weekly lunches.

There have been two very special friends, Manuel Torralaba and Jeny Panadés, that have encouraged me during all the time I have been working on this thesis. I thank Manuel for all the exciting and very useful discussions we have had in the last years. I also thank him for all the wonderful moments we have shared together. To Jeny, I have to thank her for being such a good friend and for always being ready to listen to me (by the way, did I ever tell you what this thesis is about?).

I would like to thank my parents, Jaume and Maria Lluïsa, and my brother Kikus for their love and support along these years. Their dedication and encouragement has made possible this work. I also thank them for believing that I was going to finish this work more than I did at some times.

There is a person which has suffered the worst part of working on a thesis: bad moods, no free time, etc. Gòria has bared all of these without complaining (or at least, not too often). Furthermore, she has given me all the support I needed in all those bad moments when you really need somebody. For all of this, and for many other things I have no space to write down here, thanks a lot.

Finally, I'd like to tank Xavi Martorell and Yolanda Becerra for reading this document and making many interesting comments.

This work was supported in part by the Ministry of Education of Spain under contracts TIC 537/94 and TIC 0429/95 by the CEPBA (European Center for Parallelism of Barcelona).

CONTENTS

LIST OF FIGURES	xvii
LIST OF TABLES	xxii
1 INTRODUCTION	1
1.1 Motivation	1
1.2 Scope of this Work	4
1.3 Objectives	7
1.4 Work Structure	8
2 PARALLEL/DISTRIBUTED FILE SYSTEMS	11
2.1 Introduction	11
2.2 Parallel Machines vs. Networks of Workstations	12
2.3 Operating System	14
2.4 Interface: State of the Art	16
2.4.1 Conventional Interface	16
2.4.2 Shared File Pointers	17
2.4.3 Data Distribution	18
2.4.4 Collective I/O	22
2.4.5 Interfaces aimed at Multidimensional Arrays	23
2.5 Cache: State of the Art	24
2.5.1 Replacement Policies	24
2.5.2 Delayed Write	26
2.5.3 Application-Directed Caches	26

2.5.4 Virtual Memory vs. File-System Cache	27
2.5.5 Multi-level Caching	27
2.5.6 Cooperative Caches	28
2.5.7 Cache Coherence	29
2.5.8 Non-Volatile RAM	31
2.6 Prefetching: State of the Art	31
2.6.1 User-Directed Prefetching	31
2.6.2 Prefetching via Data Compression	32
2.6.3 Aggressive Prefetching Based on File Openings	32
2.6.4 Parallel Prefetching	33
2.7 Disk: State of the Art	33
2.7.1 Disk-allocation Policies	34
2.7.2 Log-Structured File Systems	34
2.7.3 Striping Data Among Several Disks	36
2.7.4 A Hierarchy of Disks	37
2.7.5 Using Flash Memory instead of Disks	38
2.7.6 Striping and Log-Structured File System	39
3 METHODOLOGY	41
3.1 Simulator (DIMEMAS)	41
3.1.1 Architectural Model	42
3.1.2 Process Model	43
3.1.3 Communication Model	43
3.1.4 Disk Model	44
3.1.5 Process-Scheduling Policies	45
3.1.6 Trace-file Format	46
3.2 Simulation-Methodology Issues	47
3.3 Workload Characterization	50
3.3.1 Parallel Workload (CHARISMA)	50
3.3.2 Network of Workstation Workload (Sprite)	53

3.4	Parallel Environment Characterizations	55
4	COOPERATIVE CACHING AND CENTRALIZED CONTROL	57
4.1	Motivation	57
4.2	File-System Architecture	59
4.3	Block-Distribution and Replacement Algorithms	61
4.3.1	Design Issues	61
4.3.2	Block Distribution Algorithm	62
4.3.3	Replacement Algorithm	62
4.4	Coherence Mechanism	63
4.5	Adaptability to the Variation of Nodes and Virtual Memory	64
4.6	Experimental Results	66
4.6.1	Algorithm Comparison	67
4.6.2	Interconnection Network Bandwidth Influence	70
4.6.3	"Local-Cache" Size influence	71
4.6.4	Number of Links Influence	73
4.6.5	Single Server Performance	74
4.7	Conclusions	75
5	COOPERATIVE CACHING AND DISTRIBUTED CONTROL	77
5.1	Motivation	77
5.2	File-System Architecture	78
5.3	Block Distribution and Replacement Algorithm	80
5.4	Coherence Mechanism	83
5.5	Buffer Redistribution	83
5.6	Adaptability to the Variation of Servers, Nodes and Virtual Memory	86
5.7	Tuning PAFS	87
5.7.1	Queue-tip Size	88
5.7.2	Redistribution Interval	89
5.7.3	Limiting the Number of the Buffers that can be Lost	90
5.7.4	Redistribution Algorithms	91

5.8	Experimental Results	92
5.8.1	Algorithm Comparison	92
5.8.2	Interconnection Network Influence	98
5.8.3	"Local-Cache" Size Influence	100
5.9	Conclusions	102
6	FAULT TOLERANCE	103
6.1	Motivation	103
6.2	Parity Mechanism	104
6.3	Tolerance Levels	106
6.4	Experimental Results	109
6.5	Conclusions	111
7	LINEAR-AGGRESSIVE PREFETCHING	113
7.1	Motivation	113
7.2	Prefetching Algorithms	114
7.2.1	One-Block-Ahead (OBA)	114
7.2.2	Interval-and-Size-Graph (ISG)	115
7.3	Linear-aggressive Prefetching	122
7.3.1	Aggressive One-Block-Ahead (Agr.OBA)	122
7.3.2	Aggressive Interval-and-Size-Graph (Agr.ISG)	122
7.3.3	Limiting the Aggressiveness: Linear-aggressive Prefetching	123
7.4	Including the Linear-aggressive Algorithms in PAFS and xFS	124
7.5	Experimental Results	125
7.5.1	Performance Improvement	126
7.5.2	Disk Traffic	130
7.6	Conclusions	134
8	CONCLUSIONS AND FUTURE WORK	135
8.1	Contributions of this Work	135
8.1.1	Cooperative Caching	136

<i>Contents</i>	xv
8.1.2 Prefetching	138
8.1.3 Methodology	139
8.2 Future Work	140
A xFS AND N-CHANCE FORWARDING	143
A.1 Introduction	143
A.2 N-Chance Forwarding: a Cooperative Cache	144
A.3 xFS: General Overview	145
A.4 Comparison between Cooperative-cache Algorithms	148
A.5 Modifications Made in our Simulations	149
REFERENCES	151

LIST OF FIGURES

Chapter 1

- 1.1 Parts of a parallel/distributed file system. 5

Chapter 2

- 2.1 A *nested-strided* operation with two levels of strides. 19
- 2.2 Schematic representation of a two-dimensional file in *Vesta*. 20
- 2.3 An Example of data partitioning in *MPI-IO*. 21
- 2.4 Flow diagram of blocks in a segmented-LRU cache. 25
- 2.5 Differences between a traditional *Unix* file system and a log-structured file system. 35
- 2.6 *Striping* based on log-structured file systems as used in Zebra. 39

Chapter 3

- 3.1 Example of a parallel machine modeled by DIMEMAS. 42
- 3.2 Distribution of read and write operations on CHARISMA 51
- 3.3 Distribution of read and written MBytes on CHARISMA 52
- 3.4 Distribution of read and write operations during the simulation on the Sprite workload. 54
- 3.5 Distribution of read and written MBytes during the simulation on the Sprite workload. 54

Chapter 4

- 4.1 File-system architecture of the centralized version. 60

4.2	Approximation of the LRU-Interleaved to a global LRU.	63
4.3	Normalized time spent by the centralized version of both file systems serving read and write operations.	68
4.4	Influence of the interconnection-network bandwidth on the centralized version of PACA and xFS.	71
4.5	Influence of the "local-cache" size on the centralized version of PACA and xFS.	72
4.6	Influence of the number of links to the interconnection network on the centralized version of PACA and xFS.	74

Chapter 5

5.1	PAFS architecture.	79
5.2	An example of two cache-server partitions in a three-node machine.	79
5.3	Influence of the queue-tip size on the local-hit ratio (PM/CHARISMA).	89
5.4	Influence of the queue-tip size on the local-hit ratio (NOW/Sprite).	89
5.5	Influence of the interval between redistributions (PM/CHARISMA).	90
5.6	Influence of the interval between redistributions (NOW/Sprite).	90
5.7	Influence of the limitation of lost buffers (PM/CHARISMA).	91
5.8	Influence of the limitation of lost buffers (NOW/Sprite).	91
5.9	Influence of the limitation of gained buffers (PM/CHARISMA).	92
5.10	Influence of the limitation of gained buffers (NOW/Sprite).	92
5.11	Total normalized time spent by PAFS and xFS performing read and write operations (PM/CHARISMA).	93
5.12	Total normalized time spent by PAFS and xFS performing read and write operations (NOW/Sprite).	93
5.13	Interconnection-network bandwidth influence in the average read and write operation times (PM/CHARISMA).	98
5.14	Interconnection-network bandwidth influence in the average read and write operation times (NOW/Sprite).	98
5.15	"Local-cache" size influence in the average read and write operation times (PM/CHARISMA).	100

5.16 "Local-cache" size influence in the average read and write operation times (NOW/Sprite).	100
---	-----

Chapter 6

6.1 Distribution of the cache buffers in parity lines.	105
6.2 Example of a parallel make that needs a high tolerance level	108
6.3 Performance of the fault-tolerance levels (PM/CHARISMA).	109
6.4 Performance of the fault-tolerance levels (NOW/Sprite).	111

Chapter 7

7.1 Example of a couple of access patters and their prefetch graphs.	116
7.2 Example of a graph creation.	118
7.3 Example of a prefetch graph where nodes have more than one link that start from it.	119
7.4 Example of a regular access pattern that cannot be prefetched by ISG.	121
7.5 Average read operation time obtained by the presented prefetching algorithms on PAFS (PM/CHARISMA).	126
7.6 Average read operation time obtained by the presented prefetching algorithms on xFS (PM/CHARISMA).	126
7.7 Average read operation time obtained by the presented prefetching algorithms on PAFS (NOW/Sprite).	129
7.8 Average read operation time obtained by the presented prefetching algorithms on xFS (NOW/Sprite).	129
7.9 Load placed on the disks by ISG with a 1MByte local cache.	132

Chapter 8

Appendix A

A.1 Possible xFS configuration in a network of workstations.	146
A.2 Steps followed by xFS to fulfill a client request.	147

LIST OF TABLES

Chapter 1

1.1 Summary of hardware improvement trends.	2
---	---

Chapter 2

2.1 Interconnection-network and memory bandwidth of some parallel machines (peak values).	12
2.2 Peak bandwidth of the most popular interconnection networks used in NOWs.	12
2.3 Sustainable memory bandwidth of some popular workstations.	13

Chapter 3

3.1 Some of the actions that can be represented in a DIMEMAS trace file.	46
3.2 CHARISMA workload	51
3.3 Sprite workload	53
3.4 Simulation parameters.	56

Chapter 4

4.1 Average operation times and hit ratios obtained using the centralized versions of PACA and xFS.	67
4.2 Influence of the "local-cache" size on the read global-hit ratio using the centralized version of PACA and xFS.	71

Chapter 5

5.1	Average read/write times and <i>hit</i> ratios obtained by both file systems (PM/CHARISMA).	93
5.2	Average read/write times and <i>hit</i> ratios obtained by both file systems (NOW/Sprite).	97
5.3	"Local-cache" size influence on the read <i>global-hit</i> (PM/CHARISMA) ratio.	101

Chapter 6

Chapter 7

7.1	Global miss ratio obtained by the presented prefetching algorithms (PM/CHARISMA).	127
7.2	Global miss ratio obtained by the presented prefetching algorithms (NOW/Sprite).	129
7.3	Variation in disk traffic due to the presented prefetching algorithms (PM/CHARISMA).	130
7.4	Average number of times a block is sent to the disk during its life in the cache.	131
7.5	Variation in disk traffic due to the presented prefetching algorithms (NOW/Sprite).	133

Chapter 8

Appendix A

INTRODUCTION

1.1 MOTIVATION

When examining the structure of the applications that run on parallel machines, we observe that their I/O needs increase tremendously every day. These applications work with very large data sets which, in most cases, do not fit in memory and have to be kept in disk. The input and output data files are also very large and have to be accessed very fast. These large applications also want to be able to checkpoint themselves without wasting too much time. These facts constantly increase the expectations placed on parallel and distributed file systems. Thus, these file systems have to improve their performance to avoid becoming the bottleneck in parallel/distributed environments.

On the other hand, while the performance of the new processors, interconnection networks and memory increase very rapidly, no such thing happens with the disk performance (Table 1.1) [Dah95]. This lack of improvement is due to the mechanical parts

Hardware Parameter	Improvement Rate
Disk latency	10% per year
Disk bandwidth	20% per year
Disk cost/capacity	100% per year
Network latency	20% per year
Network bandwidth	45% per year
Memory bandwidth	40% per year
Memory cost/capacity	45% per year

Table 1.1 Summary of hardware improvement trends.

used to build the disks. Operations such as the movement of the heads and the rotation of the disks are handled by mechanical components. These components are slow and limit both the latency and the bandwidth of the disk. Thus, the performance of the file-system operations that have to access the disk is also limited. As the mechanical technology cannot keep the pace of the electronic one, the research community is in search of solutions that improve the file-system performance without expecting a significant improvement in the mechanical devices.

Many solutions have been proposed either to increase the disk performance or to decouple the file-system performance from that of the disks. Most of these solutions will be described in detail in the state of the art sections from Chapter 2. Among all of them, we are specially interested on caching and prefetching.

A cache, or buffer cache, is a mechanism that tries to minimize the number of times the file system has to access the disks. This decouples the performance of the file system from the performance of the disks. The basic idea consists of keeping the file blocks used by the applications in memory buffers. If applications have temporary locality, the same set of file blocks is used during a certain amount of time. In this case, these requested blocks do not need to be fetched from the disk as they are already kept in the cache. This increases the performance of read operations. The cache also increases the performance of write operations as the modified blocks are not sent to the disk but kept in the cache. Whenever the system believes appropriate, it sends these modified

blocks to the disk. One of the most attractive characteristics of this approach is that the effectiveness of a cache is not fully limited by the mechanical parts of the disks and thus its efficiency can keep the pace of the rest of the parallel machine.

Prefetching is the next logical step after caching. It uses the same data structures and mechanisms but instead of remembering blocks requested in the past it tries to predict the blocks that will be requested in the near future. So far, a cache only keeps the blocks already used by the applications. If we add a predictor, the system can also predict the blocks that will be needed by the applications and fetch them from disk before being requested. This increases the system performance if the prefetched blocks have been correctly predicted and on the right time. This mechanism also helps to decouple the performance of the file system from the performance of the disks.

If we examine the trends in technology, we can observe that memory is becoming cheaper every year. For this reason, the amount of MBytes that can be placed in each node of a parallel machine also increases very rapidly. A portion of these large memories can be used to improve the file-system performance if used as file-system caches. The larger the cache is, the more effective the caching will be.

A second trend in technology that can be used to improve the I/O performance is the increasing interconnection-network bandwidth. Accessing the information kept in another node of the parallel machine is becoming less and less time consuming. This leads to believe that data needed by a given node can be kept in a different node. This information can be requested from the remote node whenever necessary. This can only be done if this remote access is fast enough, as occurs in most parallel machines. Furthermore, it is becoming a general trend in the design of parallel machines to add special hardware that copies blocks of memory between nodes offering very high bandwidths [Gil96, Gra96]. This could be used to increase the size of the file-system caches. The system could use a part of the memory in each node to build a very large global file-system cache. This would increase the effectiveness of the cache and the efficiency of the file system.

Both ideas, large caches and high interconnection network bandwidths, are used to build what it is commonly known as cooperative caching. The idea is very simple. All nodes dedicate a portion of their large private memory to build a large global cache [DWAP94, Dah95]. The way this large cooperative cache is handled is what makes the difference between all the proposed cooperative caches.

In this thesis, we are going to propose a novel approach to design and manage cooperative caches. The efficiency of these kind of caches has been traditionally based on a high physical locality. This means that the system tries to cache the data in the node that will need it. After a deep study of the implication this design issue may have, we have detected that this approach has the problem that achieving this locality has a quite high overhead on remote operations. We propose a new approach where the efficiency of this kind of caches is not based on this locality. We will prove that very efficient cooperative caches can be designed focusing on two basic issues: increasing the global effectiveness of the cache and performing local and remote accesses in the fastest possible way. Locality will only be taken into account if achieving it does not add any overhead to the system.

Furthermore, we are also going to propose a mechanism that uses these large caches to implement aggressive prefetching. This prefetching will be aggressive enough to increase the system performance but will also be intelligent enough not to flood the cache. If the cache is flooded, the performance of the system drops and no benefit is obtained due to the prefetching.

These two mechanisms, cooperative cache and aggressive prefetching, will increase the file-system performance using hardware that grows at a similar pace that the whole parallel-machine performance.

1.2 SCOPE OF THIS WORK

This thesis is focused on designing a new cooperative cache using a novel approach. We will redesign the full cache from scratch and we will test its performance comparing

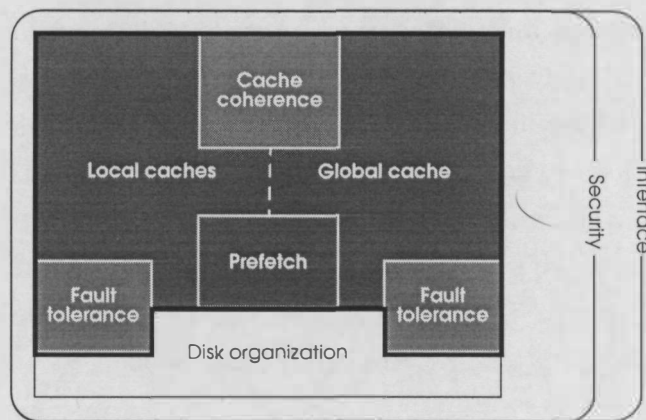


Figure 1.1 Parts of a parallel/distributed file system.

it to the state of the art in cooperative caches. We also want to present prefetching mechanisms that are able to take advantage of the characteristics offered by the cooperative caches. This new design implies a redesign of some other parts of the parallel/distributed file system. Figure 1.1 presents all the parts that make a file system. It also shows which parts have to be modified along with the cooperative cache (light shaded). In this section, we will justify the decision of which parts need to be modified and which ones do not.

In a cooperative cache, the cache can be divided in two parts: a local and a global one. Each node has a local cache and manages it so that the blocks most needed by the applications running on this node are kept in this local cache. The global cache has a more general mission, it keeps file blocks that are important for the whole system, not favoring any given node or application. As the different parts of the cache are managed in a distributed way, it is quite frequent to find replicated information in the system. This replication is allowed because it is assumed to increase the system performance. Having blocks replicated in several nodes means that modifying one of these copies leaves the system inconsistent. Not all the buffers that keep the same block have the same information. The way this problem is handled depends on the way the cooperative cache is designed and the sharing semantic implemented in the system. For this reason, redesigning a cooperative cache implies a redesign of the coherence mechanism.

One of the basic ideas of a cooperative cache is that file blocks may be cached in any node. This means that when an application modifies a file block, this block can be placed in any node of the machine. This modified block may even be placed in a node where the application is not running. If this node fails, the modified block will be lost and the application will be affected by the failure of a node that has nothing to do with the application. This is not acceptable in some environments and a tolerance mechanism should appear in our design. For this reason, we also have to design a fault-tolerance mechanism to solve this important problem.

The other parts of a file system fall out of the scope of this thesis as a new design is not needed nor specially beneficial to the new design proposed. In the remaining of this section, we will describe the assumptions made on the unmodified parts of the parallel/distributed file system.

The interface assumed is the one offered by any *Unix* system. This does not mean that any other more sophisticated interface can not be used, it only means that our cache will not rely on any special information given by a non *Unix* interface. Anyway, it seems reasonable to study how could our cache benefit from interfaces that allow the user to specify more semantic information about their operations, but this study falls out of the scope of this work.

As far as security is concerned, our system will have the same security issues as a centralized system. All possible problems derived from the network and remote nodes are not taken into account. We assume a secure network. We also assume that all nodes in the network are secure. This should not be a very important limitation as the system is designed to run in a parallel machine where nodes and communication are secure. Running this system in a network of workstations should not be a big problem either as they are usually protected from the outside by sophisticated mechanisms such as firewalls. Anyway, it also seems interesting to study a way to increase the system security in the near future.

The disk organization is also out of the scope of this work. Any state of the art algorithm could have been used as the behavior of the cache does not depend much on

this organization. We have assumed an interleaved placement of blocks among disks and a file system structure based on i-nodes. This structure was used because it is very well-known and it should help us, and the reader, to interpret the results presented later in this work.

1.3 OBJECTIVES

In this section, we present, in a very schematic way, the main objectives of this thesis. These objectives will be pursued through out all this work although they may be relaxed a little bit in some chapters.

- Design a cooperative cache for a parallel/distributed file system. This new design will be used to prove that a high-performance cooperative cache can be designed using the following design issues:
 - Favor global effectiveness over physical locality.
 - Favor the efficiency of remote operations over physical locality.
 - Offer a very strict sharing semantic (*Unix*-like) by not allowing data replication. This will eliminate the coherence problem and will also simplify significantly the file-system design.
- Offer a simple and efficient fault-tolerance mechanism that avoids losing any modified data when a node fails.
- Design prefetching mechanisms that take advantage of the large size of a cooperative cache.
 - It has to be aggressive to use the large caches.
 - It should not flood the cache with useless data.

1.4 WORK STRUCTURE

This thesis is organized in 8 chapters as follows:

- Chapter 2 gives a general overview of the target environment of this work. We present the kind of machine, operating system and general assumptions used in this work. We also present the state of the art in the file system field.
- Chapter 3 starts describing DIMEMAS, the simulator used to obtain all the performance results. Then, we go through the methodology used in the simulation. And finally, we describe the simulated workloads and the parameters used in the simulator.
- Chapter 4 presents the first proposal of our cooperative cache. This is a centralized version which is basically used to get a deep understanding on the way cooperative caches work. Furthermore, it also serves to obtain the first feedback of the viability of the novel approach proposed in this thesis.
- Chapter 5 describes the distributed version of the cooperative cache. In this chapter, we present the final version of the file system and cooperative cache designed following the approach proposed in this thesis. We also present and evaluate some alternatives to the basic design but always keeping in mind the main structure proposed in this thesis.
- Chapter 6 modifies the proposed file-system architecture in order to offer some degree of fault tolerance. Furthermore, three tolerance levels are proposed so that the most adequate can be used in each system.
- Chapter 7 describes the linear-aggressive prefetching. This is a mechanism that converts a conservative algorithm in an aggressive one. These new prefetching algorithms are aggressive enough to highly increase the file-system performance. But they are intelligent enough not to flood the cache and decrease the overall system performance.
- Finally, Chapter 8 presents the most important conclusions extracted from this work. It also gives a brief description of the work that could follow and complement the one presented here.

In Chapters 4, 5, 6 and 7, the performance results presented have been compared against the ones obtained by xFS, the distributed file system proposed in the NOW project [ACPtNt95, ADN⁺95, Dah95].

2

PARALLEL/DISTRIBUTED FILE SYSTEMS

2.1 INTRODUCTION

In this chapter, we will set the basis for this work. We will first describe the environment in which this work is designed to run, the differences between a parallel machine and a network of workstations, the kind of operating systems our file system is targeted at and the most important abstractions needed from the operating system in order to implement our file system. Finally, we will examine the state of the art in parallel/distributed file systems. This state of the art will be divided into four parts: interface, cache, prefetching and disk.

Parallel machine	Network bandwidth (per link)	Memory bandwidth
Silicon Origin 2000	0.4 GBytes/s	0.8 GBytes/s
Cray T3E	0.5 GBytes/s	1.2 GBytes/s
Tera MTA	2.67 GBytes/s	2.67 GBytes/s
NEC SX-4 (IXS)	8 GBytes/s	16 GBytes/s

Table 2.1 Interconnection-network and memory bandwidth of some parallel machines (peak values).

Network Name	Network bandwidth
Ethernet	1.25 MBytes/s
Fast Ethernet	12.5 MBytes/s
FDDI	12.5 MBytes/s
ATM	78 MBytes/s
Memory Channel	100 MBytes/s
Myrinet	168 MBytes/s
Dolphin	200 MBytes/s

Table 2.2 Peak bandwidth of the most popular interconnection networks used in NOWs.

2.2 PARALLEL MACHINES VS. NETWORKS OF WORKSTATIONS

The cooperative cache presented in this work is designed to run in a parallel/distributed environment. This can either be a parallel machine or a network of workstations and it is important to clarify the differences between both environments. In this section, the most basic differences are explained.

In both environments, we have a set of nodes connected through an interconnection network. One of the most important differences appears in this network as the one found in a parallel machine is usually much faster than the one found in a network of workstations. For this reason, the services implemented on a parallel machine can rely

Processor	Memory bandwidth
HP C180	262 MBytes/s
DEC 4100 5-300E	261 MBytes/s
Pentium based	(around) 128 MBytes/s
Sun Ultra2 2200	311 MBytes/s
SGI Octane 195	351 MBytes/s
IBM R6000 591	800 MBytes/s

Table 2.3 Sustainable memory bandwidth of some popular workstations. These values have been extracted from the STREAM Benchmarks.

on communication much more than if they were running on a network of workstations. This difference in the interconnection-network bandwidth can be observed comparing tables 2.1 and 2.2. We can also observe that in most parallel machines, the memory bandwidth is around twice the bandwidth of the interconnection network (Table 2.1). Furthermore, this same ratio can also be obtained in a network of workstations if the appropriate network is used (Tables 2.2 and 2.3).

The second difference can be found in the concept of node. In a parallel machine, all nodes are more or less equal and belong to no particular user. The system administrator is the only responsible for all the nodes and the nodes are not especially configured to meet the special needs of a given user. On the other hand, in a network of workstations each workstation may belong to a different user who can configure it in any way he or she wishes. This concept of ownership, that only exists in the network of workstations, is the cause of the following differences.

As each workstation belongs to a given user, this user may decide that the node will not cooperate on the network tasks during a certain period of time. This can be done to increase the node performance when the user needs to do a lot of computational work. This will not happen in a parallel machine as nodes are not supposed to work for any special user. For this reason, allowing temporal disconnections is a must in a network of workstations while it may not be necessary in a parallel machine.

Finally, there is the problem of fault tolerance. In most parallel machines, but not all, the failure of a node means the failure of the whole machine. If this is the case, there is no need to add fault tolerance mechanisms to the services running on top. On the other hand, in a network of workstations, the failure of a single node may be quite frequent. It is quite common for a given user to decide to shut the workstation down and this cannot mean the failure of the whole network. For this reason, all services running on a network of workstations should be able to handle node failures.

Once we have clarified the difference between parallel machines and networks of workstations, it is important to say that the work presented here was originally designed to work on a parallel machine. Anyway, as networks of workstations are becoming very popular and are starting to compete with parallel machines, some modifications to the original design have been done to be able to run our system in a network of workstations.

2.3 OPERATING SYSTEM

The philosophy of our file system is that responsibilities are distributed among the nodes in the network. We want to have several nodes in charge of coordinating the contents of the cache and serving the client requests. We also want other nodes to be responsible for the disks. These nodes will have the disks connected to them and will perform all physical read and write operations. Another set of nodes will also be responsible for maintaining the information needed to have a fault-tolerant system. In this way, all file-system responsibilities are distributed among the nodes in the network. This does not mean that a given node may not have more than one responsibility, but a good distribution of tasks is desirable.

This distribution of tasks can be done in two different ways. The first one consists of implementing a set of servers that take care of each part of the file system. In this case, the personality of each node depends on which server, or servers, are running on it. The other way to make this distribution consists of modifying the kernel to manage a certain part of the file system. We have decided to use the first approach as it is

simpler to implement. Furthermore, in the experimental part of this work we have seen that it does not introduce a significant overhead.

Another important aspect of this work, is the assumptions we make about the operating system our file system is designed to run on. These assumptions are described in detail in the following paragraphs.

The first assumption is that the operating system is a multi-programmed and multi-threaded one. This is needed for a couple of reasons. The first one is to be able to run servers and clients in the same node sharing the CPU. Furthermore, it would be quite interesting to be able to have multi-threaded servers in order to have several requests handled in parallel.

Second, we need ports to communicate the clients with the servers. This mechanism will only be used to send requests from the client to the server and to send notifications from the server to the clients. No file data will be send using this mechanism.

Finally, a *memory_copy* mechanism is needed. This mechanism is used to transfer data between the cache and the user. This mechanism should be faster than sending the information through a port. Our assumption is that any processor can set up a data transfer between any other two processors. Similar remote-memory-access mechanisms are supported in a variety of distributed-memory systems [LGP⁺92, WMR⁺94].

Operations such as *memory_copy* have to be supported by the hardware. We need a fast hardware mechanism to copy memory blocks from one node to another. This kind of fast copies are being implemented in most currently designed parallel machines [Gil96, Gra96]. For this reason, we believe that assuming a fast remote-memory copy operation is a reasonable assumption as it is a trend in the design of new parallel architectures.

Through out all this work we have assumed that the file system is built on top of a micro-kernel operating system. We have chosen this operating-system architecture instead of a monolithic one as it seems to be best fitted for a parallel machine and to

support subsystems made by servers. Anyway, a monolithic operating system could have also been used.

2.4 INTERFACE: STATE OF THE ART

In the last years, many research groups have been working on different I/O interfaces. Some of them are quite conservative and only present small modifications over the traditional one found in *Unix*. Others, which try to be more innovative, propose completely new semantics. These new semantics are designed to allow users to specify the nature of their applications much more precisely.

All these projects have been developed basically following two different approaches. The first one consists of integrating the new semantics into the file system itself. The second one is to build a run-time library in order to offer the new semantics to the user. These libraries use the old semantics built in the file system underneath in order to offer the new ones to the user. The first option achieves a better integration with the file system, thus achieving a higher performance. The second option is much more flexible and portable as it is not chained to a specific file system. In this work, we will not take into account this difference as all the semantics proposed could have been implemented using any of the two techniques, without losing any of their basic properties.

In the following subsections, we will briefly describe these projects and the new concepts they have proposed. We will start describing the conventional interface which was designed in the *Unix* systems and then we will also give an overview of the modifications done to this interface as the shared file pointers. Ideas as data distribution, collective I/O and special interfaces for a given kind of applications will also be covered.

2.4.1 Conventional Interface

The conventional interface is more or less the one proposed by Ritchie and Thompson in the early-seventies. This interface was originally designed for *Multics* [FO71] but later developed as part as the *Unix* operating system [RT74]. In *Unix*, a file is nothing

more than a linear sequence of bytes where operations such as **open()**, **close()**, **read()**, **write()** and **seek()** can be performed.

Among the parallel file systems which use this interface we could mention sfs, implemented for the CM-5 [VIN⁺93], and Concurrent File System, implemented on an Intel iPSC/2 [Pie95].

This interface, that has achieved quite good results in mono-processor file systems and sequential applications, is not able to satisfy the needs of most parallel and distributed applications [KN94, NKP⁺94]. This is due to its lack of flexibility to express the needs of such applications.

2.4.2 Shared File Pointers

A small improvement over the conventional interface is the introduction of the shared pointer. This mechanism allows several processes to work with the same file pointer. This gives these processes the possibility to coordinate themselves to access a shared file in a cooperative way.

The first example of such file pointers is found in BSD version 4.3 [LMKQ89]. In this version of *Unix*, there is an atomic way to append data to a file. Using this mechanism that is specified by a flag in the open operation, a file can grow with information coming from several processes in a concurrent way. We can see that, although it is a kind of shared file pointer, it is a very primitive one.

MPI-IO also offers the possibility to use shared pointers [MPI96, Mes97, CFF⁺95]. Whenever a file is opened by several processes, MPI-IO creates a set of pointers made of as many local ones as processes plus one global pointer. This last one is shared by all processes. The usage of this pointer is not limited to the append operations as in the above example. It is much more flexible and can be used in any kind of operation.

In the Concurrent File System, we find several kinds of shared file pointers [Pie95]. The first one is a shared pointer with no limitations as the one offered by MPI-IO. The

second one allows all processes to access the same file but using a round-robin order. This means that if a process wants to access the file out of its turn, it will have to wait until all other processes have accessed the file. Finally, the last shared file pointer is basically as the previous one but the size of the accessed blocks has to be the same in all requests.

Besides all the shared pointers offered by the Concurrent File System, SPIFFI proposes a distributed shared pointer [FBD96]. The idea is to offer a pointer that avoids, in a distributed manner, the overlap between the data accessed by any two processes. This new shared pointer allows each process to access a different, and not overlapped, portion of the file. The control of this pointer has been designed to add as little overhead as possible because it is mostly handled locally by the processes. To achieve it, the file system assigns a disk to each process using a round-robin algorithm. Once a process knows which disk to access, it reads or writes all the blocks from this file located in its assigned disk. Once all blocks from this file which are located in this disk have been accessed, the file system assigns a new disk to the process. The process will use this new disk as before. This assignment of disks to processes is repeated until all disks where this file has blocks have been assigned to one process.

2.4.3 Data Distribution

Strides

The experience obtained from the CHARISMA project showed that a great number of the I/O operations done in a parallel environment used strided patterns [KN94, NKP⁺94]. A strided operation is an operation that accesses several pieces of data that are not contiguous but separated by a certain number of bytes. As this kind of operations cannot be handled easily nor efficiently using the conventional interface, the Galley file system proposes three new interfaces to perform such operations [Nie96, NK95]. A description of these three interfaces follows.

The first of them, named *simple-strided*, allows the user to access, in a single operation, to N blocks of M bytes placed P bytes away from each other.

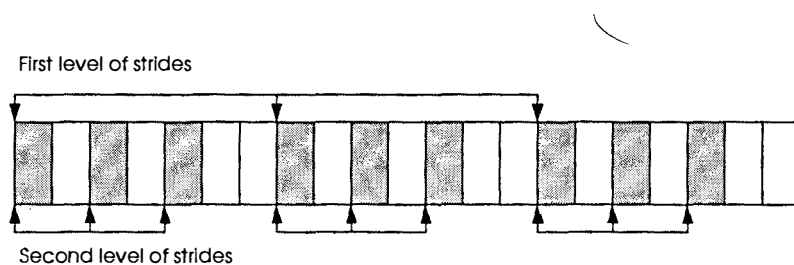


Figure 2.1 A *nested-strided* operation with two levels of strides.

The second proposal is the *nested-strided* operation. With this interface, the user can build a vector of strides that define the different levels of stride. From these levels, only the last one represents the placement of data blocks. All other stride levels indicate in which positions does the next level of strides start. Figure 2.1 presents an example of this kind of operations. In this example, there are two levels of strides. The first one indicates that the next level has to be applied three times once every seven elements. The second, and last, level indicates that user wants to access three elements leaving one element between each pair. This last level is used three times as indicated by the first level. The requested elements are represented as the shaded blocks in the figure.

Finally, they also propose a *nested-batched* operation. This operation allows the user to make several *simple* and *nested-strided* operations as a single one. In order to do this, the user builds a list with the strided operations to be done. Afterwards, this list is used as one of the parameters in the read or write operation.

Two-Dimensional Files

The concept of two-dimensional files was defined as part of the Vesta file system [CF96, CBF93]. In Vesta, a file is made of a set of cells, or partitions, where each of them is divided into Basic Striping Units (BSUs), or registers.

Cells are contiguous portions of a file. They are defined at creation time and its number remains constant during the whole life of the file. Each of these cells may be placed in a different I/O node. This means that the number of cells equals the maximum parallelism that can be obtained accessing a given file. This placement is made by the file system with no interaction from the user.

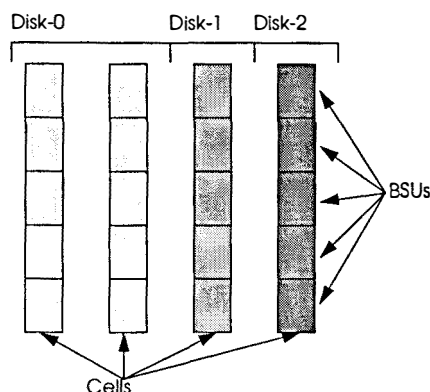


Figure 2.2 Schematic representation of a two-dimensional file in *Vesta*.

BSUs are sets of bytes that behave as the basic data units used by the repartition mechanism. The size of BSUs is also set at creation time and remains unmodified during the whole life of the file.

This division of a file in cells and BSUs allows us to see the file as a two-dimensional array. The cells behave as columns while the BSUs take the place of the rows (Figure 2.2). This interpretation of files as two dimensional arrays allows the user to repartition the BSUs in the same way that array elements can be distributed using High-Performance Fortran [Hig93]. The same partition mechanisms are offered by the system. Any possible partition of the file is what they call a subfile. Actually, a user can only open subfiles and thus when opening a file, the file and the partitioning scheme has to be given as parameters. Using this mechanism, a process can only access the bytes contained in its subfile avoiding any sharing problems.

Data Partitioning in *MPI-IO*

MPI-IO was proposed as an I/O interface standard [MPI96, Mes97, CFF⁺95]. It is based on the MPI message passing library [Mes94, Mes97]. This interface only allows the user to specify the distribution of the file blocks among the user space. The distribution of this data among the disks is left to the file system underneath.

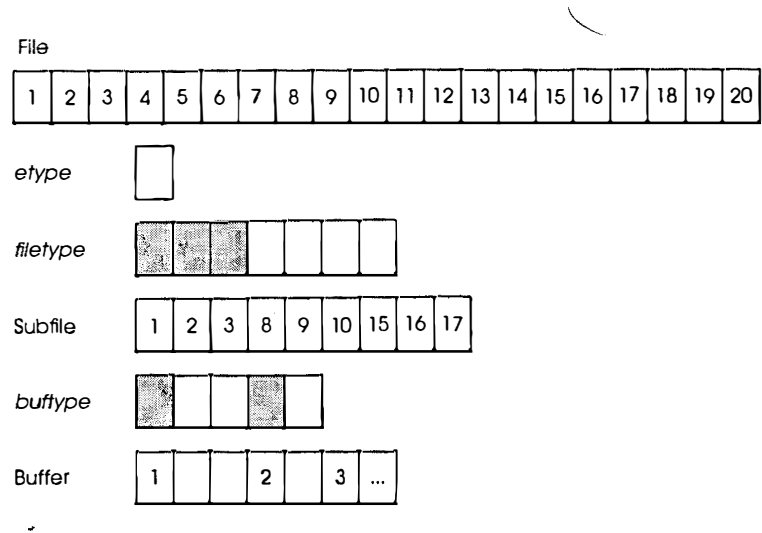


Figure 2.3 An Example of data partitioning in MPI-IO.

This interface was designed with a very clear set of goals. It was targeted primarily for scientific applications. It favored common usage patterns over obscure ones. It ties to support 90% of parallel programs easily at the expense of making things more difficult in the other 10%. It also intends to correspond to real world requirements and new features are only added when they are proved to be necessary. Finally, the design favors performance over functionality.

In MPI-IO, the data partitioning is done in three steps. The first one consists of defining the size of the basic element or *etype*. This element is used to construct the patterns needed in the next two phases. The second one is done in the open operation and specifies which parts of the complete file will be used to construct the opened subfile. Finally, in the third step, the user defines the accessed elements going to be placed in the buffer. This third step can be done in each read/write operation. These two last steps or mappings are done building patterns from the basic element, or *etype*, defined in the first step. When a file is opened, the user specifies the *filetype* which is a pattern of *etypes*. This pattern defines the subfile to access (Figure 2.3). Using this concept of subfile, the user may specify which blocks are going to be used by each process in the application. Once the subfiles have been specified, every read and write operations may specify a pattern telling the system where to place the elements accessed from the file. This new pattern is also build from *etypes* and is called *buftype* (Figure 2.3). It is

important to notice that there are no restrictions when building subfiles and they may be overlapped.

Going back to the shared pointers issue, it is straightforward to see now that the global pointer will only make sense if all processes have the same subfile.

2.4.4 Collective I/O

A collective I/O is that situation where all computing nodes cooperate to perform an I/O operations from disk in order to improve its efficiency. This allows the system to build a single big operation from all the small ones requested by each client. In this way, the system is able to obtain more semantic information about the operations and may improve their efficiency. This cannot be done if many small operations are requested at different instants of time. This basic idea has led some very interesting projects such as Two-phase I/O [HdC95, CBM⁺94, BdC93, dBC93], Jovian [BBS⁺94] and Disk-directed I/O [Kot95a, Kot95b, Kot94].

Two-phase I/O

This I/O strategy involves a division of the parallel I/O task into two separate phases. In the first phase, the parallel data access is performed using a data distribution, stripe size, and set of reading nodes which conforms with the distribution of data over the disks. Subsequently, in phase two, the data is distributed at run time, within the processor array, to match the application's desired data distribution.

By employing the two-phase redistribution strategy, the number of requests from the processor array to disks is reduced compared to direct access. Furthermore, the redistribution phase improves performance because it can exploit the higher bandwidths made available by the higher degree of connectivity within the interconnection network of the computational array.

Jovian

Jovian is another run-time library that implements collective I/O. It is based on the idea that applications run alternating computing and I/O phases. As these two phases appear in all members of an application, Jovian synchronizes all of them in each I/O phase. In this way, all I/O operations can be grouped in order to make a collective I/O. At the beginning of an I/O phase, all processes in the application send their request to the processes in charge of the collective I/O. Even if a process does not want to perform an I/O operation, it will also send a request (an empty one). The processes which receive the requests will try to optimize them and access the information from the disks using the most efficient way.

Disk-Directed I/O

Finally, Disk-Directed I/O tries to go a little further than the previous proposals. Besides synchronizing all I/O operations, it also tries to send these requests to the disk minimizing the mechanical overheads such as the seek operation. It also improves the performance of the I/O operations overlapping disk accesses with the distribution of the information among the computing nodes.

2.4.5 Interfaces aimed at Multidimensional Arrays

Panda [SW96, SCW⁺95] provides an interface that eases the work with files that contain multidimensional arrays. When using this interface, the user only has to set some characteristics of the array and some distribution directives. With this information, Panda is able to distribute the array among the disks and nodes very efficiently. Furthermore, as only semantic information is set by the user, this interface becomes very portable. Thus it is very useful to develop parallel applications and kernels.

2.5 CACHE: STATE OF THE ART

As far as caches are concerned, there are three main issues that have drawn the attention of researchers in the last years. First, different replacement algorithms have been proposed in order to make the caches more effective. Along with these algorithms, there has also been some work devoted to develop extensible caches where the user can set the replacement algorithms or, at least, give some hints to the system. Second, some studies have been done to examine the influence of the distributed environments on the file-system caches. Finally, some techniques to distribute the cached information among the nodes in the network have also been proposed.

2.5.1 Replacement Policies

When the system decides that a block has to be kept in the cache, we may not have any free buffers in which to place this new block. In such situation, one of the buffers in the cache has to be emptied. The algorithm that decides which block will be discarded is what it is called the replacement algorithm. The more effective this algorithm is, the better the cache behaves. In this subsection, we will explain the three replacement algorithms that have obtained the best results [RL96, K LW94, RD90].

Least-Recently Used (LRU)

As can be extracted from its name, this algorithm always replaces the least-recently used block in the cache. This algorithm tries to take advantage of the temporal locality. If a block has been recently used it will probably be used again in a short period of time. On the other hand, if a block has not been used recently, it will probably not be used shortly. This is the replacement algorithm most widely used in commercial file-system caches. Its popularity is due to its high effectiveness and the simplicity of its implementation.

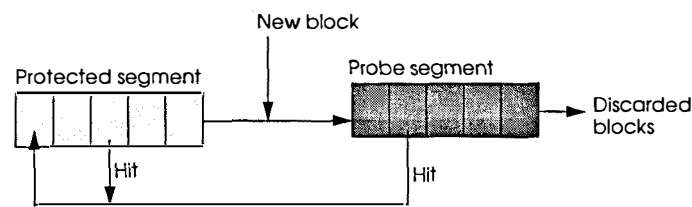


Figure 2.4 Flow diagram of blocks in a segmented-LRU cache.

Segmented LRU

The idea behind this algorithm is that blocks which have been requested more than once will probably be used again. In order to take advantage of this heuristic, the replacement algorithm will try to avoid replacing blocks that have been accessed more than once [KLW94].

A segmented-LRU cache is divided into two segments: the probe segment and the protected one. In both segments, all blocks are kept in a queue ordered from the least to the most recently used. When a new block enters the cache it is placed in the probe segment. As it has not been accessed more than once, there is no indication that it will be used again thus it cannot be placed in the protected segment yet. If the user requests a block that it is already in the cache, this block will be placed as the most recently used of the protected segment. This is done because the block has been accessed at least twice and the heuristic says that it will probably be used again. When the system has to discard a block, the least recently used one from the probe segment is discarded. Finally, when a block has to leave the protected segment it will be placed as the most-recently used one of the probe segment. Figure 2.4 shows the flow diagram of the blocks in a segmented-LRU cache.

Frequency-Based Replacement

Finally, we present the frequency-based replacement algorithm. The objective of this algorithm is to find the most popular blocks and keep them in the cache. In order to do it, the system keeps a counter with the number of times each block has been referenced. Whenever a block has to be discarded, the one with the smaller number of references

is replaced. Although this is the basic idea, an exact implementation of this algorithm may be very ineffective in some cases. In order to solve problems such as blocks that are accessed many times in a very short period of time and never referenced again, some modifications have been proposed [RD90].

2.5.2 Delayed Write

It is quite straight forward to see that in order to achieve efficient write operations, the physical disk write has to be done in a delayed way. Furthermore, this delayed write may even avoid some disk writes if the block is deleted before reaching the disk. This happens quite often with temporal files or data. On the other hand, this physical write cannot be delayed too long. If the new information is kept from disk too long, the probability of losing the modifications due to a system failure increases.

Deciding the best moment to physically write the modified blocks to disk has been studied quite widely [KE93, Kot91, Mog94, RT74]. The most popular proposals try to write the modified blocks either when the buffer is needed, or periodically, or when a buffer is filled completely, or when the disk is idle, etc. The decision of which one is the best depends on whether the efficiency is more important than the fault tolerance or not.

2.5.3 Application-Directed Caches

Although a file-system cache has the potential of improving the performance of the I/O operations, this improvement is not always achieved [Kot91, Smi85]. There are some situations where keeping a file in the cache makes no sense as there is no temporal locality on its usage. If this file is kept in the cache, it will uselessly fill some buffers that might hold some other important blocks. This will decrease the effectiveness of the cache and it may even decrease the I/O performance. For this reason, many systems allow the user to activate/deactivate the caching of a given file [Swe96, CPdA96, CPd⁺96, Car95, PCG⁺97, HER⁺95].

Another common problem with caches is that the replacement algorithm of a given system may not be the right one for some applications. For this reason, some systems let the user define the replacement algorithm that has to be used with its files. This mechanism has to guaranty that the effectiveness of the cache for the rest of the applications is not damaged by a bad decision made by one application. One of the best examples of one such mechanism was proposed by P. Cao [Cao96, CFL94b, CFL94a]. In her work, she proposes a replacement algorithm with two levels. The application controlled level allows the application to decide which of its cached blocks are to be replaced. The system level controls the number of cache buffers assigned to each application. This two-level replacement algorithm guarantees that an application may only decrease its own performance. This happens because the distribution of buffers among the applications is a task left to the system. Other example can be found in extensible file systems such as ELFS [GL91] and [KN93]. This file systems allows the user to implement the best cache strategies for its application using extensible mechanisms.

2.5.4 Virtual Memory vs. File-System Cache

On one hand, it is a good idea to have big file system caches in order to be very effective. On the other hand, if the cache is too big, the physical memory available for the virtual-memory systems may not be enough. When this happens, the system starts trashing and all the applications slowdown their execution. In order to solve this problem, the Sprite operating system proposes a variable-size cache [Nel90]. This mechanism decides dynamically the portion of physical memory given to the cache and to the virtual-memory system. To avoid trashing, a higher priority is given to the virtual-memory system. With this mechanism memory-bound applications are not affected by the file-system cache. Furthermore, when memory is available, the I/O-bound applications may get the advantages of a big cache.

This mechanism has been adopted in many modern systems such as HP-UX and Solaris.

2.5.5 Multi-level Caching

It is important to notice that caching can be done at many different levels. This is specially important if the environment is a parallel or distributed one. Caching may be basically done in the disk controllers, in the servers, in the client nodes, in the I/O libraries, etc. Plenty of researching has been focused on studying the influence of the different levels of cache [Car95, K LW94, Hel93, NWO88, Smi85]. As a common result of these studies, it has been shown the convenience of having more than one level of caches, although it has also been shown that higher level caches decrease the hit ratio of the lower level ones. This happens because the higher level caches absorb most of the locality.

2.5.6 Cooperative Caches

A cooperative cache is a cache that tries to improve the performance of a parallel/distributed file system by coordinating the contents of the caches found in all nodes. This coordination allows a request from a given node to be served by the local cache of a different node.

Until cooperative caching came into sight, all the client caches were isolated and uncoordinated. In order to increase the potential of cooperation, the NOW team proposed the idea of cooperative caching [DWAP94]. This idea was also implemented in xFS which was part of the NOW project [ACPtNt95, ADN⁺95, Dah95]. Along with the cooperative cache, some algorithms to coordinate the local caches were also proposed. Among all those algorithms, N-Chance Forwarding was presented as the most adequate to coordinate the local caches and still allow them some freedom.

A. Leff et al. proposed a similar idea. In their work, they study the impact that could have a coordination of local caches. They also study the impact that replication could have on the system performance [LWY96, LWY93].

Finally, some ideas have been proposed in order to improve the performance of the cooperative caches. For instance, M.J. Feeley proposed a different algorithm to coordi-

nate the contents of the local caches, which obtained a better effectiveness [FMP⁺95]. P. Sarkar and J. Hartman also presented a mechanism based on hints that avoided much of the load placed on the managers [SH96].

2.5.7 Cache Coherence

Using file-system caches in different nodes may rise coherence problems. Two nodes may be caching the same file block and this one should be kept coherent when one of the nodes modifies it. To solve this problem, two basic approaches have been followed. The first one consists of relaxing the sharing semantics. This allows simpler and more efficient coherence algorithms. The second approach tries to find efficient algorithms that can fulfill the *Unix* semantic.

File-sharing Semantics

The first relaxation is the session semantic. When this semantic is used, all modifications done on a file are only visible, at the same time, to the processes running on that node. These modifications are not visible by any other process which has this file open and is running on a different node. Once the file is closed, all the modifications become visible to all the applications that open that file after it has been closed by the process which modified it. AFS is an example of file system that uses a session semantic [Kaz88, HKm⁺88].

For data-base oriented file systems, a transaction semantic has also been proposed. When this semantic is used, all I/O operations are done between two control instructions: *begin-transaction* and *end-transaction*. All modifications done between these two instructions will not be visible to the rest of nodes until the transaction is over. Once the transaction is finished, all the modifications done between the control instructions are propagated to the rest of the nodes in an atomic way. This kind of semantic has been used in distributed systems like Cambridge [NH82] or in the distributed shared memory proposed by Feeley [FCNL94].

Finally, a semantic where files never change has also been proposed [SGN85]. When this semantic is used, once a file is created it can never be modified. All changes mean a new file (or version).

Coherence Algorithms

The first solution, and also the most drastic one, consists of not allowing the caching of write operations when a file is shared. This avoids all coherence problems allowing a *Unix* semantic. The main problem with this approach is that write operations are much slower than they should be. A similar approach is the one used in the Sprite operating system [NWO88].

The second option is based on the use of tokens. When a client wants to write on a file it requests the write-token for that file. As long as a client keeps the token, this client may do as many modifications to the file as needed without asking any more permissions. Whenever another client requests the right to modify that file, the server sends a message to the current owner informing that the token has been revoked. Once the owner acknowledges the token revocation, this token can be handed to the new client.

A problem found when protecting coherence with tokens is the revocation mechanism. In order to give the token to a new client, the current owner of the token has to release it and stop making modifications on the file. If the node with the token fails or it is disconnected from the network, there will be no way to revoke the token. And, without the token, no other client will be able to modify the file. In order to solve this problem a new mechanism was proposed: the leases [KM95, Mac94, BHJ⁺93, MBH⁺93, GC89]. A lease is very similar to a token but for an expiration time. This means that if a client has a lease to modify a file, this permission will only last for a predefined amount of time. In this way, if the client fails, or it is disconnected, the rest of the clients will only have to wait for the lease to expire in order to get their lease to modify the file. On the other hand, the disconnected client, will stop modifying the file once the lease expires and will have to ask for a new one if it wants to keep modifying the file.

These two mechanisms, tokens and leases, may be used with different granularity. The first possibility is to use the file as the basic sharing unit. This means that if a client wants to modify a block of a file, it has to request the token or lease of the whole file and no other client may modify any other block. The second possibility is to use the file block as the basic sharing unit. With this granularity many clients may be writing on the same file at the same time, but never on the same block. xFS uses this granularity [Dah95, ADN⁺95]. Finally, a user defined granularity has also been proposed [Gar96, GCP⁺96]. Clients define the region they are going to modify which is not necessarily contiguous. Once this region is defined, a client will be able to modify any portion of this region as long as there is no overlapping with other client's region.

2.5.8 Non-Volatile RAM

Due to the decreasing in the price of the non-volatile memories, some researching has been done to use this kind of memories as file system caches. An example is the usage of NVRAM on the client nodes in order to reduce the traffic between the clients and the disk server [BAD⁺92]. As these caches are non volatile, a more aggressive caching can be done with no loss in reliability.

2.6 PREFETCHING: STATE OF THE ART

Most file systems offer some kind of data prefetching in order to increase the performance of the I/O operations. The most commonly used policy consists of prefetching the next n blocks starting from the one currently requested. Other more sophisticated policies try to discover the access pattern in order to predict which blocks are going to be requested next [Kot91, KE90, Smi85, Smi78]. In this section, we will briefly present some new techniques that have been proposed in order to increase the effectiveness of the prefetching.

2.6.1 User-Directed Prefetching

One way to take the predicting task away from the file system, and still be able to do some prefetching, is leaving the task to the application. Applications know, most of the times, the access pattern used in its file better than the system. If good mechanisms are offered to the application to pass this information to the system, a more effective prefetching can be achieved. Based on this idea, a transparent informed prefetching was proposed as part of the Scotch file system [GSC⁺95, PGG⁺95, PG94]. This mechanism allows the user to give hints to the system about which blocks are going to be needed in a near future. This information is only a hint to the system that may, or may not, be used in order to prefetch blocks from disk before the application actually needs them.

In order to offer the user the possibility to implement their prefetching policies, some systems offer asynchronous reads as part of their interface [ACR95, GL91, HER⁺95]. This kind of calls give the user all the needed flexibility to implement any kind of prefetching algorithm.

2.6.2 Prefetching via Data Compression

A quite well studied prediction mechanism is the one used in data compression. Data-compression algorithms try to predict which sequences of bytes are more popular to substitute them by shorter ones. As these algorithms are designed to predict the future quite well, why not use them to predict which block to prefetch. Based on this idea several prefetching algorithms have been proposed with quite good results [KL96, VK96, CKV93]. All these algorithms build a probability graph which grows as the system runs. This graph keeps the information of which blocks are more probable to be requested after a given block has been requested. Using this information, the system tries to predict which data blocks will be requested next by the applications.

2.6.3 Aggressive Prefetching Based on File Openings

In most environments, when a file is opened it is quite easy to predict which files will be opened next. For instance, in a *Unix*-like environment, once the `make` executable is opened it is very probable that the files `makefile`, `cc` and `ld` will also be opened in a near future. Using this idea, R. Griffioen and R. Appleton proposed a quite aggressive prefetching algorithm [GA94, GA93]. In their work, they propose that the file system maintains information of which files have been opened after which other files. Using this information, the system tries to predict which files to prefetch once a given file is opened.

2.6.4 Parallel Prefetching

Recently, there has also been a great deal of interest on prefetching from parallel disks as a technique for improving I/O performance of sequential and parallel applications. This research has been focused both from the theoretical and from the practical point of view.

Kimbrel et al. have done a theoretical study where several parallel prefetching algorithms are presented [KTP⁺96, KK96, Kim97]. They evaluate them assuming that there is only one process that only has one stream of read requests and that all these accesses are known in advance. This work is a continuation of the one done by Cao et al. where similar algorithms are proposed for a single disk [CFKL95]. Both projects designed prefetching algorithms based on a full knowledge of the access pattern in advance.

The Transparent Informed Prefetching also implements parallel prefetching [GSC⁺95, PGG⁺95, PG94]. It allows sequential applications which are not able to use more than one disk in parallel to take advantage of all the disks in the system. This is done by prefetching blocks from more than one disk in parallel increasing the I/O performance of these applications.

2.7 DISK: STATE OF THE ART

One of the main reasons behind the inefficiency of the I/O operations is the limited speed of the disks. The problem with the disks is that most of the parts are mechanical and cannot keep the pace with electronic components as memory or processing units. Actually, the most important problem with the disks is the seek operation. Moving the heads from one cylinder to another is a very slow mechanical function. To hide this problem, several solutions have been proposed. In this section we will go through the most important ones.

2.7.1 Disk-allocation Policies

A way to avoid the number of seek operations consists of allocating the data of a file in a way that once the first block of the request is accessed, the rest of the blocks can be accessed with the minimal number of seeks [NV77]. Another possible optimization consists of allocating the blocks of a file near their i-node. This avoids the seek operation needed to go from the i-node to the file blocks [Tri80].

Based on the above-mentioned ideas, quite a few allocation policies have been proposed [MK90, Pea92, MJLF84]. Latter studies have shown that all these allocation policies degenerate as the file system ages [SS96, SSB⁺95]. In order to avoid this degradation of the system, M.K. McKusick presented some modifications to his original algorithm [Com94]. These modifications consist of adding a new step before the physical write to the disk. In this step all contiguous blocks from a file are grouped and try to be written contiguously. This may need some re-allocation of blocks as they where originally allocated to a disk block while they are now written on a different one. Obviously, this mechanism only makes sense if delayed-write policies are used and some blocks from a file can be waiting to be written to disk.

2.7.2 Log-Structured File Systems

This kind of file systems is based on the assumption that caches obtain very high read hit ratios. As most read operations are handled by the cache, the disk traffic is dominated by write operations. Thus, it seems a good idea to try to improve write operations even if reads are slow-downed a bit.

A log-structured file system performs all write operations in a sequential way. This means that a modified block is not overwritten on the same disk block, but written in the next sequential disk block after the last write operation. This behavior emulates a log where old information is never erased but superseded in later page of the log. This organization of the file system increases write operation performance as no seek operations are needed [RO92, Ros92, RO90]. Furthermore, as blocks are not overwritten it is quite easy to recover the file system to a consistent state after a failure.

As all the information (data and meta-data) is written sequentially, it is very important to have an efficient mechanism to access this information. Actually, once we have the i-node of a file, the algorithm needed to access the file blocks is the same as in the traditional *Unix* file system. Thus, we have to find an efficient way to locate a given i-node in the log. In this kind of file systems, the i-nodes are not placed in a fixed position that can be calculated using the i-node number. They may be found in any part of the disk. To solve this problem, a new data structure is added to the file system. The i-node map is an array that contains the information on which disk block is kept the last copy of each i-node. As this data structure is compact enough, it can be kept in memory. Thus no disk accesses are needed in order to use it. For fault-tolerance reasons, this i-node map is periodically saved in the disk and can be recovered after a system failure.

In Figure 2.5 we present an example of the changes suffered in the structure of the file system when creating files `dir/file1` and `dir/file2`. These changes are presented for a traditional *Unix*-like file system and for a log-structured one. The most important difference is that, in a traditional *Unix*-like file system, the meta-data does not change its placement. On the other hand, the meta-data of the modified file is written

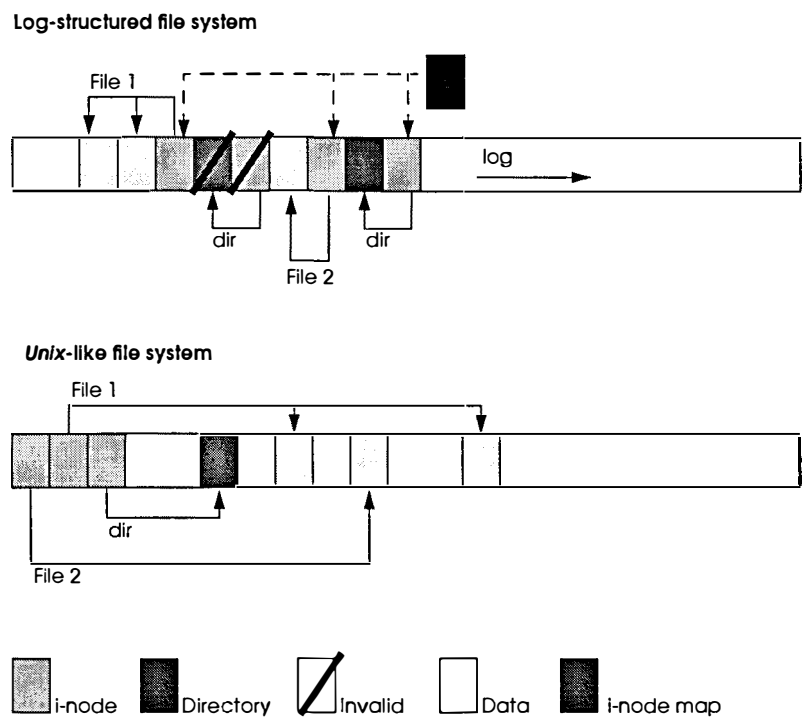


Figure 2.5 Differences between a traditional *Unix* file system and a log-structured file system.

sequentially with the rest of the modified information. It is also important to notice that the information in the directory (i-node and directory) becomes obsolete and it is superseded by a new up-to-date copy in a different disk block.

To make this kind of file systems work efficiently, it is necessary to keep large amounts of contiguous free blocks. These free areas are obtained by recovering the blocks that have been superseded by a new version of the information. In order to get these areas big enough there is also the need to compact the file system information and regroup these free blocks. Many algorithms have been presented to solve this problem but their explanation goes beyond the scope of this chapter [Rob96, CS92, Ros92].

2.7.3 Striping Data Among Several Disks

Another mechanism that is frequently used to increase the disk bandwidth is striping the information among several disks. Thanks to this distribution, information from all disks can be accessed in parallel. This mechanism can be used in two levels. The first one consists of connecting several disks to a single controller board and offers a faster disk to the user. The second level consists of striping the information among several disks attached to different nodes connected through a network.

Redundant Arrays of Inexpensive Disks (RAID)

RAIDs are the first proposal to stripe information among several disks giving the image of a single disk with a higher bandwidth [CLG⁺94, PGK88, Law81]. The idea is very simple. It consists of striping the information on several disks in order to make all seek operations in parallel. Furthermore, this mechanism increases the disk bandwidth as many times as disks connected to the board.

The main problem presented in this kind of disks is its higher probability of physical failure. It is straightforward to see that the higher the number of disks the probability of one of them failing also increases. In order to solve this problem, five levels of redundancy have been proposed to allow some fault tolerance. These levels go from a complete replication (RAID-1) to an interleaved striping of parity blocks (RAID-5). This last level is the most popular as only the capacity of one disk is wasted due to replication. The main problem found with RAID-5 is that only the failure of one disk at a time is allowed. From this initial model other mechanisms have been proposed to allow the failure of more disks without increasing the portion of disk needed to keep parity blocks [PC96, BM93].

RAID-5 also have an efficiency problem when small writes are requested. If only a portion of the information needed to calculate a parity block is modified, the rest of the information has to be read in order to recalculate the new parity block. Some solutions to this problem will be presented before the end of this chapter.

Striping Data among Disks from Different Nodes

As the interconnection networks are becoming very fast, we can extend the concept of striping to the whole system. We can scatter information among all disks in the network and, at the same time, be able to access it in an efficient way. The only conceptual difference between a RAID and this new approach is who controls the striping. In a RAID, the controller is the one that strips the information without the knowledge of the file system. With this new approach, it is the file system the one in charge of distributing the information among all the nodes in the network. Obviously, we still have the same problems found in a RAID and similar solutions can be presented. Among the systems that use this mechanism we could name Swift [CL91], Bridge [DS89], RAMA [MK97, MK95] and sfs [VIN⁺93] among many others.

2.7.4 A Hierarchy of Disks

In a similar way as the main memory hierarchies, some projects have proposed several levels of disks in order to increase their efficiency without losing their reliability.

AutoRAID

One of these projects came out with the AutoRAID designed by Hewlett-Packard [WGSS96, WGSS95]. An AutoRAID is made of two levels of RAID. The first one works as a cache keeping the most recently modified information. As this disk has to be very fast, a RAID-1 is used. This kind of RAID is faster, as keeping the replicated information does not take time but disk space. Furthermore, the small write problem is also avoided. The second level contains the whole file system information and thus cannot waste too much space keeping parity information. For this reason a RAID-5 is used in this second level. The data movement from the first to the second level is done automatically and in a transparent way.

Disk-Cache Disk (DCD)

The idea behind this kind of disk is solving the efficiency problem of the small writes. In order to solve this problem a small disk is used to cache a big one. The first one is structured like a log, thus all the write operations are done sequentially. As this disk does not keep parity information we avoid the small write problem. Later on, when the disk is idle, there is a movement of information from the disk cache to the main disk [HY96].

2.7.5 Using Flash Memory instead of Disks

As the price of flash memories [Adv93, Int94] has dropped dramatically in the last years, some studies have been done in order to use them instead of disks. A. Kawaguchi proposed the implementation of a log-structured file system that uses this kind of memory in a very efficient way [KNM95]. There has also been another implementation of file systems on top of flash memories that placed special interest on optimizing transactional operations [WZ94].

2.7.6 Striping and Log-Structured File System

This combination of log-structured file systems and RAID has been widely used as the problems found in one system can easily be solved by the characteristics of the other. Among all this mixed file systems we would like to mention Zebra [HO95, HO92], Sawmill [SO94] and xFS [Dah95, ADN⁺95].

Zebra

In Zebra data is striped among all the disks in the network. In order to solve the small-write problem, Zebra takes advantage of the large sequential writes that can be obtained by the log-structured file systems. Each client keep a log of all the modifications done by its clients. Once the log is long enough to fill all the disks, the parity buffer is calculated. Then, the client log and parity block are sent to the disks. This parity can

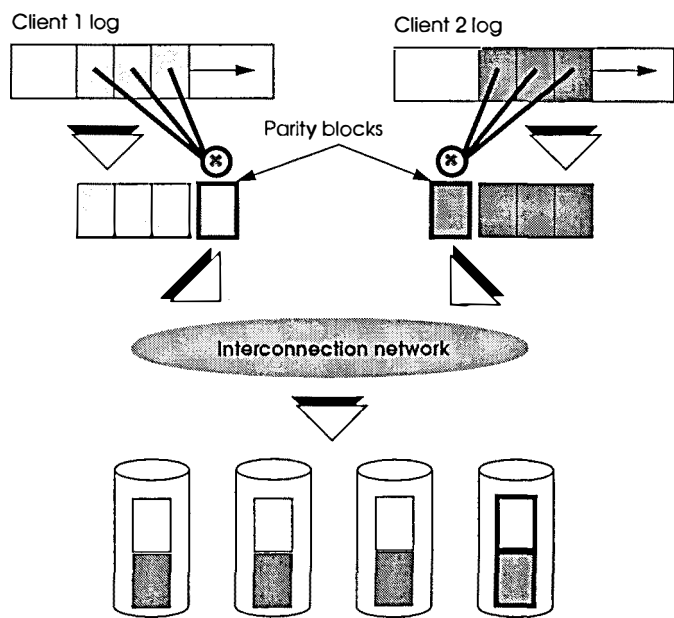


Figure 2.6 *Striping* based on log-structured file systems as used in Zebra.

be calculated without reading any information from the disks as it will fill a complete line. In Figure 2.6 we present an example of a Zebra file system with two clients and four disks.

Sawmill

Sawmill uses a similar technique to the one used in Zebra but using the second generation of RAIDS. The RAID-II architecture was developed to solve the problem found with RAIDS connected to the network [DSH⁺94, Kat92]. The original idea of a RAID was to offer a very high bandwidth. If we connect this RAID system to a workstation which behaves as a disk server, the bandwidth will be limited by the bandwidth of the workstation’s memory. In order to solve this problem, this new RAID architecture allows the disks to be directly connected to the network using an XBUS controller. This avoids the bottleneck of the workstation memory.

xFS

The basic behavior of xFS is very similar to Zebra. The main difference is that xFS is specially designed for scalability. For this reason, the striping is not done on all disks. As proposed for big RAIDs [CLG⁺94], the system defines stripe groups. Each of these groups of disks works isolated and keeps the parity information without the interaction of any other group. Each client knows to which group it has to write by using a globally replicated stripe-group map.

Furthermore, this system tries to set the bases 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.

A more detailed description of this system is presented in Appendix A as this system will be the performance reference point for the work presented in this thesis.

3.1 SIMULATOR (DIMEMAS)

All the results presented in this thesis have been obtained through simulation. This mechanism was used so that a wide range of environments and architectures may be studied. The file-system and cache simulator used in our experiments is part of DIMEMAS¹ [LGP⁺96] which is a performance-prediction tool that simulates the behavior of a distributed-memory parallel machine.

In this section, we describe the characteristics of DIMEMAS most relevant to this work. We describe the architectural, process, communication and disk models. Finally, we also describe the format of the trace files used by DIMEMAS.

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

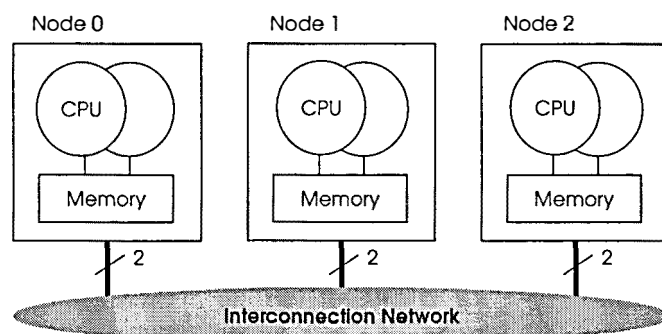


Figure 3.1 Example of a parallel machine modeled by DIMEMAS.

3.1.1 Architectural Model

The architectural model simulated by DIMEMAS is simple and flexible. It is composed of a set of shared-memory multiprocessors and an interconnection network. Each node in the network has a given number of processors and a private memory. This memory is shared by all the processors in the node and it is only accessible by the processes running on the node. Furthermore, this memory can be used as a communication mechanism between the processes located in the node. Using this model, we can simulate shared-memory multiprocessors by having a single node. We can also simulate a traditional distributed-memory machine where each node only has one processor. Furthermore, any other configuration in between can also be simulated using this model.

DIMEMAS implements two kinds of interconnection networks: bus-based and cross-bar. The bus-based interconnection network connects all nodes by a given number of busses and only as many messages as busses can be traveling in parallel. The cross-bar interconnection network simulates that each node can reach any other node using a different path. This means that the system may have as many messages traveling in parallel as the number of nodes multiplied by the number of links each node has.

Nodes are connected to the interconnection network through links. Each one of these links allows the node to send a message to the network. This means that a node with four links can send four messages in parallel as long as the interconnection network has enough bandwidth left for them.

Figure 3.1 presents an example of the typical parallel machine simulated by DIMEMAS. This machine has three nodes with two processors each. All nodes have a local memory, accessible only from the processors located in the node. Furthermore, each node has two links to the interconnection network.

In all the experiments run, we have used nodes with only one processor and a cross-bar interconnection network. Furthermore, all nodes only had one input and one output links to the interconnection network unless otherwise specified.

3.1.2 Process Model

DIMEMAS offers a three-level process model as the one proposed in PAROS [LGP⁺92]. This model is specially designed for distributed-memory parallel machines but can also be used in any other architecture. The three levels are Ptasks, tasks and threads.

The Ptask concept refers to a parallel application. This entity owns the resources allocated to the application. In DIMEMAS, a Ptask refers to an application and the processor set where the application is mapped. A task is a logical address space in which one or more flows of control can be executed. Every task is mapped onto one, and only one, node where the physical address space is located. Finally, in the third level, we find the threads. A thread is a flow of control within a task. All threads in a given task share the address space that belongs to their task.

All three levels have been used when designing the file system simulated in this thesis. The whole file service is handle by a Ptask, or parallel application and each file server is represented by a task which may have many threads. These threads are the ones that will do all the work as all higher levels are only resource allocation entities.

3.1.3 Communication Model

DIMEMAS offers two communication mechanisms: channels and ports. The first one is designed to communicate threads within a Ptask, or parallel application. On the other

hand, ports are the communication mechanism used to communicate threads that are running in different Ptasks.

Besides channels and ports, the simulator also offers a fast memory copy operation that can also be included as a communication mechanism. This operation allows an application to copy blocks of memory from one node to another. This operation emulates the block-mover operations currently implemented in most parallel machines [Gil96, Gra96].

The simulator can emulate several communication models based on two orthogonal semantic aspects: buffering and synchronization. By buffered we mean non *rendez-vous* communication where both tasks involved in the communication do not need to have reached the communication point before the real data transfer is started. The sender can send the message even if the receiver has not started the receive operation. The message is kept in a buffer until the receiver wants to get the message. By synchronous communication, we understand that the sending task must wait until the data transfer is finished before being able to perform any other useful work.

All simulations done in this thesis used buffered and asynchronous communications because it is the most widely used communication model (i.e., PVM [GBDJ93], MPI [Mes94, Mes97], etc.).

The communication time is modeled by a linear function (i.e., latency and bandwidth). The latency is the time needed to start a communication and it is assumed to require CPU activity. The bandwidth along with the size of the message is used to compute the time needed by a message to reach its destination once the startup time has finished. The influence of the distance between nodes is considered irrelevant in accordance to the network state of the art. Both, startup and bandwidth depend on whether the communication takes place within a single node or between two different nodes.

3.1.4 Disk Model

In this work, we have modeled the disk in two different ways. The first one consists of simulating a fixed access time. In this model, reading or writing a block take a fixed amount of time. Although both operation times are constant, they are not necessarily equal and are passed as a parameter to the simulation. This very simplistic model was used during the first part of the thesis (i.e., centralized version) as it simplified the behavior of the system. This eased the task of understanding the way file systems and cooperative caches work.

In the second part of this work (i.e., distributed version), we changed the disk model to a linear function similar to the one used in the communication model. The time needed to access information from the disk is divided into two parts: a latency and an access time. The access time is proportional to the disk bandwidth and the size of the information to be read or written. Of course, this size has to be a multiple of the disk-block size. The latency is the time needed to perform all seek and search operations and depends on the kind of operations. Reading from the disk may have a different latency than writing to it.

We are aware that both models are probably too simplistic and that a better one could have been used. Anyway, we believe that the results presented in this work are quite independent of the disk model used. The important thing is that accessing the disk takes much longer than accessing a remote memory.

3.1.5 Process-Scheduling Policies

DIMEMAS also simulates several process-scheduling policies. Among all of them we could cite Round-robin, FIFO, *Unix* SVR4, etc. In all our simulations, we have used round-robin with priorities. All user applications share the processors with the same priority while the file system has a higher priority. This higher priority of the file system is to speed up this service and to avoid it becoming a bottleneck. This policy was chosen because it was the simplest one that allowed processor sharing and a higher priority to the file system.

Operation	Parameters
CPU Usage	time consumed
OPEN	file name, file descriptor, flags, initial size
CLOSE	file descriptor
READ	file descriptor, size
WRITE	file descriptor, size
SEEK	file descriptor, offset, whence
UNLINK	file name

Table 3.1 Some of the actions that can be represented in a DIMEMAS trace file.

3.1.6 Trace-file Format

The trace file used by DIMEMAS is made of records that describe the possible actions done by an application. Among all possible actions, Table 3.1 shows the ones used in our experiments. In this table, we present the operations and the parameters they need. The behavior of each Ptask is described by one such file, and thus the system needs to have as many trace files as applications. Each of these files describes all actions made by all the threads in the Ptask.

It is very important to notice that this trace-file format is quite different from the traditional one used to represent file-system workloads. Traditionally, trace files contain the absolute time when each file-system event took place [BHK⁺91, KN94, NKP⁺94]. With this kind of traces, an increase in the performance of the file system could not result in a speed up of the simulated applications. Furthermore, if the file system is able to perform the user request faster than in the original system, this might also increase the number of requests received per unit of time. This cannot be simulated using the old trace format, either.

The trace format used by DIMEMAS records the CPU time consumed between two file-system events. This avoids absolute times and the behavior of the simulation gets closer to the one that might be found in a real system. If file-system operations are faster, the application execution time decreases. This kind of effect will be simulated by

DIMEMAS using the new trace format while it could not be done using the traditional simulators and traces.

3.2 SIMULATION-METHODOLOGY ISSUES

In this section, we describe the methodology issues that are not an implicit part of the simulator. We present the mechanism used to translate the traditional trace files to our new format, the way chosen to communicate applications, the resource consumption, etc.

Trace Files

As we have already explained, the traditional trace-file format is not appropriate for our simulation. For this reason, we had to translate the original traces to our new format. The main problem consisted of replacing the absolute time of each event by a relative one. What we did was to use the time interval between two events as the CPU needed between these two events. This is not a perfect translation as the time kept in the trace may be larger than the CPU really consumed by the application. This time interval represents both the time used to perform the operation and the CPU consumed by the application. The problem is that there is no way to know how long the file system operation took in the original execution and thus we cannot compute the exact number of CPU cycles consumed between two I/O operations.

The second problem we had was the possibility of race conditions. As the simulated application may start to run faster than the original one, we could simulate a wrong execution due to race conditions. A race condition may appear between read and write operations and between unlink and open operations. The first one does not change the simulation results as we are not concerned about the real contents of the file blocks. As we are simulating, the important thing is that a read or a write operation took place, not the data used. The unlink/open race condition was a bit more dangerous as it could cause very different executions. To solve this problem we renamed all unlinked files every time they were unlinked. This operation was done in the translation of the

file and the order used was taken from the absolute times of the original trace file. This guarantees that the simulated execution is very similar to the original one.

Finally, we had to map these trace files into the three-level process model. As we only had information about the node where operations took place, we decided to use a very simple mapping. We simulated a single threaded application per node. These applications performed all file-system operations done from their node.

We are aware that these traces are not perfect. The format has some small problems such as the race conditions already mentioned. The translation of the CHARISMA and Sprite traces does not reflect the exact behavior originally recorded either as we have not way to get the exact consumed CPU time. Anyway, the problems found in our traces are not as important as the benefits of this new trace format. We can now measure the impact that the performance of the file system has on the applications. We have used the CHARISMA and Sprite traces instead of gathering new traces because both workloads have been studied in detail by the research community. Furthermore, these traces have been used in many file system simulations making our results easier to compare and understand.

Resource Utilization

When simulating a file system, it is necessary to emulate a reasonable resource utilization. Most of this utilization has already been described by the different models implemented in the simulator. The communication model defines the contention of the interconnection network, the disk model describes the way disks are accessed. etc. But there are some resources that have to be used and have not been defined yet. The first one consists of the CPU used by the file servers. We have already defined the scheduling policy, but we also need to decide how many CPU cycles these servers will use. Another set of resources that have to be managed are the auxiliary buffers used in delayed operations.

As we want to simulate a file system as close to the reality as possible, it has to consume CPU. For this reason, our servers consume a certain amount of CPU after each blocking

operation (i.e, after receiving a message, after accessing the disk, after a memory copy, etc.). The CPU consumed by the servers depends on how fast the simulated processor was.

A second resource that has to be taken into account is the number of auxiliary buffers the system can use. Auxiliary buffers are those buffers where a cache block is kept waiting for a delayed operation. For instance, if a block has to be sent to the disk but this write can wait until the user operation is done, the system will delay this physical write. As the system needs the buffer, the block is temporarily placed in an auxiliary buffer to keep the information. Once the user has been notified, the system can write the block into the disk as a copy of the data is still in the auxiliary buffer. This number of buffers cannot be infinite and thus has to be managed. In our experiments, the system was allowed to use 16 such buffers per node. If an auxiliary buffer was needed in a given node and there were none free, the operation could not be delayed and had to be done in the critical path of the user request.

Communication between Applications

The communication model offers three communication mechanisms: channels, ports and memory copies. In our experiments, we have only used two of them. User requests and system notifications are always done using ports while data movement is always done through the memory-copy mechanism. This second mechanism is used as it is faster than sending the information through a port.

Simplicity in the Model

In this thesis, we are presenting a way to design and implement cooperative caches and aggressive prefetching. As these were the two main issues in this work, we decided to focus on them and left other related issues in the simplest possible way.

We could have used a log-structured file system instead of a traditional *Unix*-like one. We could have also used different ways to distribute blocks among the disks instead of

using an interleaved algorithm. Other replacement policies could have also been used. If none of them was used was to simplify the model. We have always used the most well-known algorithms and mechanisms as they simplify the study of the important issues proposed in this work.

3.3 WORKLOAD CHARACTERIZATION

To test the ideas proposed in this thesis, we need to simulate our proposals under real workloads. We have done this by using two well-known workloads: CHARISMA [KN94, NKP⁺94] and Sprite [BHK⁺91, Har93]. The first one describes the load of a parallel machine while the second one describes the load of a network of workstations. In this section, we describe some of the details we think more relevant to the work presented. More accurate descriptions can be found in the literature.

3.3.1 Parallel Workload (CHARISMA)

This trace file was collected from the Intel iPSC/860 at NASA Ames' Numerical Aerodynamics Simulations (NAS). This multiprocessor had 128 computational nodes, each one with 8 MBytes of memory.

The information kept in this trace file only contains the operations that went through the Concurrent File System. This means that any I/O which was done through standard input and output or to the host file system was not recorded. The complete trace is about 156 hours long and was collected over a period of 3 weeks in February 1994. To reduce the simulation time, we only used a part of this trace file. We used a two-day period (i.e., February 15th and 16th) which was the busiest one in the whole trace. Incidentally, the first 10 hours were used to warm the cache as we want to study the permanent-state behavior.

We have done a study of the workload to get an idea of the load placed on the file system. The results of this study can be seen in Table 3.2 and Figures 3.2 and 3.3. The table presents the total number of operations and accessed 8KByte blocks during

	Total		Warm cache	
	operations	blocks	operations	blocks
Read	1021	4919	812	3493
Write	2303	6201	1987	4653
Open	10		5	
Close	10		5	
Unlink	1		0.5	
Total	3345	11120	2809.5	8146

Table 3.2 . Number of thousands of operations and thousands of 8KByte accessed blocks in the CHARISMA trace file.

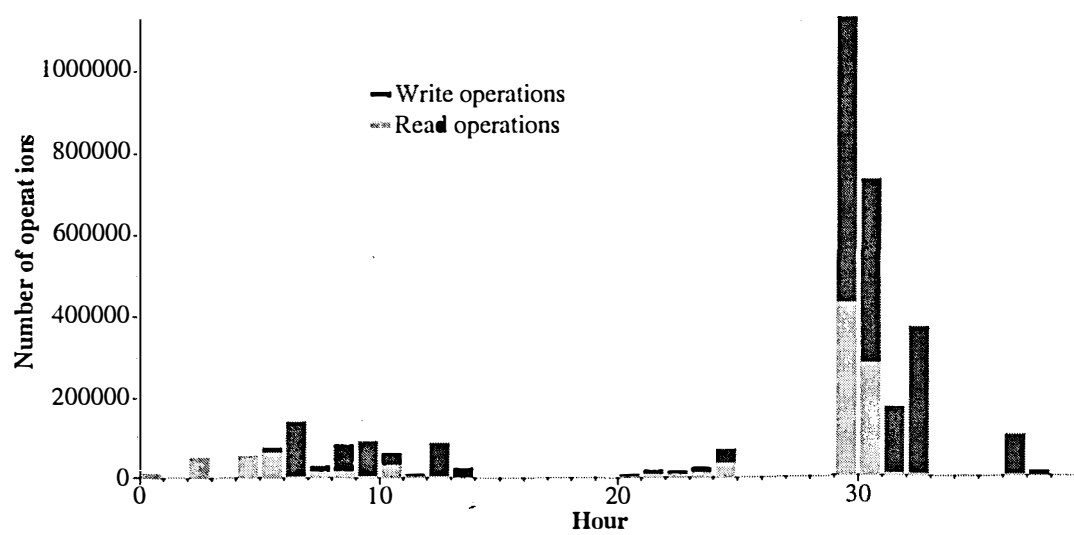


Figure 3.2 Distribution of read and write operations during the simulation on the CHARISMA workload.

the whole simulation. As the simulation only measured what happened after the tenth hour, we have distinguished the load observed in the whole trace and the one found during the simulated period. Both figures present the number of operations and MBytes accessed during the whole simulation. Each bar represents the load observed during an hour of the simulation. Furthermore, each bar is divided into two parts. The dark one represents the load due to write operations and the light one represents the load due to read operations.

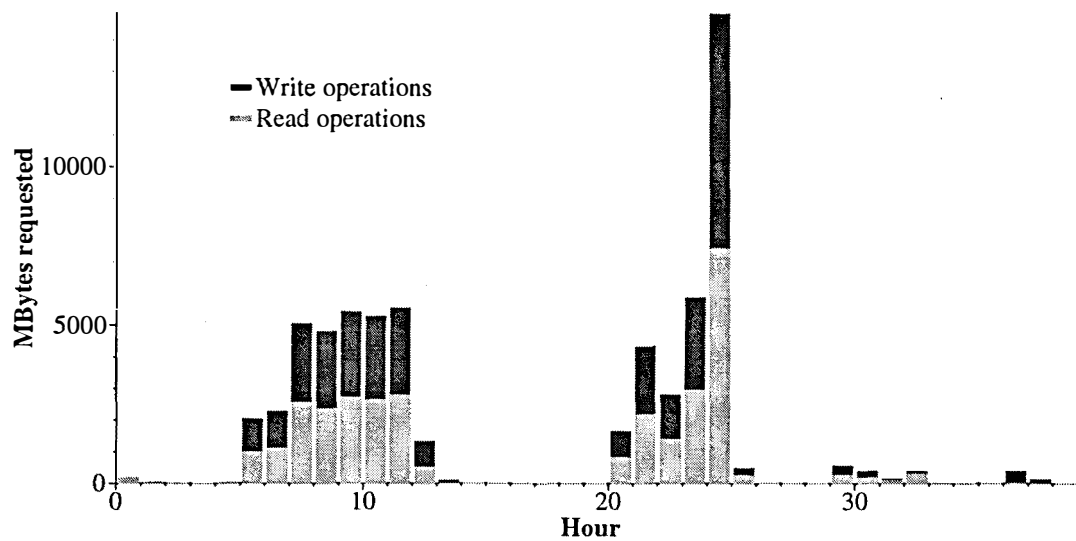


Figure 3.3 Distribution of read and written MBytes during the simulation on the CHARISMA workload.

If we examine Figure 3.2, we observe that during some periods of time many operations are requested to the file system while very few ones are requested during other periods of time. A similar behavior is observed in Figure 3.3. The distribution of accessed MBytes is quite uneven. There are very busy intervals and very quiet ones. If we examine both figures in parallel, we detect two basic behaviors. During the busy periods between the 10th and the 26th hours we have very few requests but many MBytes are accessed. This means that most user operations requested very large blocks of information. On the other hand, after the 29th hour, many operations result in a small number of requested MBytes. This means that the user requested very small blocks of information in each operation. We can also observe that more write than read operations are performed while a very similar number of MBytes is accessed by both kinds of operations. This similarity of accessed MBytes seems to go against the results presented in table 3.2. This table shows that more blocks are written than the ones that are read. This difference is because in the figure we show the real number of accessed bytes while in the table blocks are presented. If the user requests only 10 bytes, these ten bytes are accounted as a block that has been accessed in the table while only a 10-byte request is counted in the graphs.

	Total		Warm cache	
	operations	blocks	operations	Blocks
Read	684	868	520	650
Write	225	384	192	306
Open	541		407	
Close	530		400	
Unlink	60		48	
Total	2040	1252	1567	956

Table 3.3 .Number of thousands of operations and thousands of 8KByte accessed blocks in the Sprite trace file.

In this workload, most of the files are between 10 KBytes and 1 MByte in size. Files are not as big as expected because of the limited disk capacity of the traced system. Other parallel workloads, also studied in the CHARISMA project, showed that parallel environments use larger files. These more adequate traces were not used because only the general results are available, but not the original trace files. Anyway, these files are large enough and are larger than the ones found in traditional mono-processor systems.

Regarding the access patterns used by the applications, they are mostly pure sequential and strided. Furthermore, there is plenty of block reutilization which encourages the idea of file-system caching.

3.3.2 Network of Workstation Workload (Sprite)

The Sprite user community included about 30 full-time and 40 part-time users of the system. These users included operating-system researchers, computer-architecture re- searchers, VLSI designers and "others" including administrative staff and graphics re- searchers. These traces list the activity of around 50 machines including clients and servers. Although the trace is two days long, all measurements presented in this thesis are taken from the 15th hour to the 48th hour. This is done because we wanted to use the first fifteen hours to warm the cache.

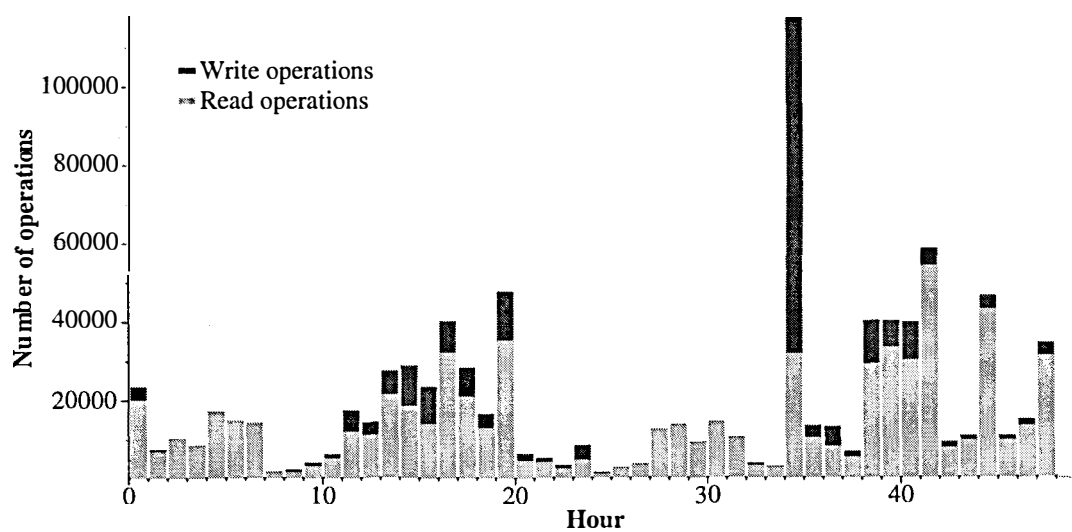


Figure 3.4 Distribution of read and write operations during the simulation on the Sprite workload.

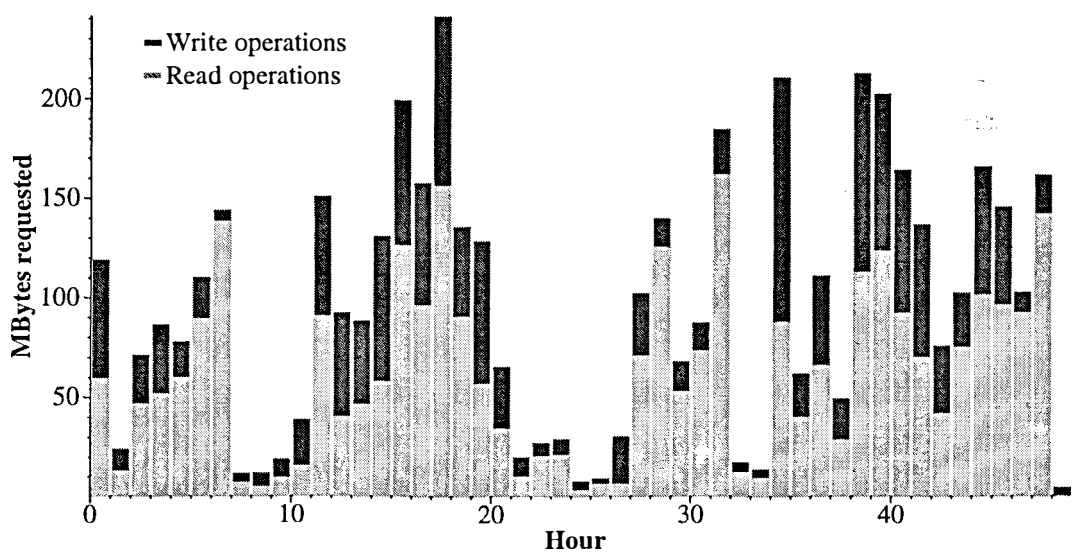


Figure 3.5 Distribution of read and written MBytes during the simulation on the Sprite workload.

As we did with the CHARISMA workload, we have done a study of the traces and the distribution of the load. This study is reflected in Table 3.3 and Figures 3.4 and 3.5. The meaning of the table and figures is the same as the one presented in the CHARISMA study.

The first thing that we can observe is that although the load is not evenly distributed through out all the simulated period, the differences are not as big as in the CHARISMA workload. We can also observe that the number of operations and the number of accessed MBytes is much lower than in the CHARISMA workload. This is due to a couple of reasons. First, the trace was taken from a network of workstations where not as many parallel applications were running. Many of the processes that ran on this trace file were editors, shells, administrative software, etc. that do not necessarily place much stress on the file system. The second reason has to do with the age of the traces. These traces are around 7 years old and the stress placed on the file system was not as high as the one we can find today. We can also observe that reads dominate over writes in both number of operations and quantity of accessed MBytes. This differs from the previous study where the load due to both kinds of operations was very similar. This difference is probably due to the different kind of work done under both environments.

The size of the files is also smaller than in the previous study. Most of the files are smaller than a few tenths of KBytes. The access pattern used by the application is basically sequential and the sharing of data between applications is not very frequent.

3.4 PARALLEL ENVIRONMENT CHARACTERIZATIONS

Once we have described the workload that will be used in the experiments, we also need to set the parameters of the simulator. As we have two different workloads, we have decided to use two different sets of parameters. The first one simulates a parallel machine and will be used to run the CHARISMA trace file. The second one simulates a network of workstations similar to the one used by the NOW team in their experiments. This second configuration will be used to simulate the Sprite trace file. The actual parameters used in both configurations are presented in Table 3.4.

These values will be used through out all this work unless it is otherwise specified.

	Parallel Machine	NOW
Nodes	128	50
Local-cache size	1 MB	16 MB
Buffer size	8 KB	8 KB
Auxiliary buffers per node	16	16
Memory bandwidth	500 MB/s	40 MB/s
Network bandwidth	200 MB/s	19.4 MB/s
Local-port startup	2 μ s	50 μ s
Remote-port startup	10 μ s	100 μ s
Local <i>memory-copy</i> startup	1 μ s	25 μ s
Remote <i>memory-copy</i> startup	5 μ s	50 μ s
Number of disks	16	8
Disk-block size	8 KB	8 KB
Disk bandwidth	10 MB/s	10 MB/s
Disk-read seek	10.5 ms	10.5 ms
Disk-write seek	12.5 ms	12.5 ms

Table 3.4 Simulation parameters.

COOPERATIVE CACHING AND CENTRALIZED CONTROL

4.1 MOTIVATION

As has already been shown by several research groups, cooperative caches are a good alternative to improve the performance of parallel/distributed file systems [LWY96, SH96, Dah95, FMP⁺95]. All cooperative caches implemented so far have been designed with a distributed control to avoid bottlenecks. This distributed control is necessary if many nodes take part in the cooperation. On the other hand, it may not be always necessary when a small number of machines are used in the cooperative cache. It is a good idea to explore the advantages and disadvantages that can be found in a centralized version of a cooperative cache when it is used with a small number of nodes. Furthermore, if we examine most of the currently running parallel machines and networks of workstations, we find that they do not have a great number of nodes. Actually, most of them do not have more than 50 nodes. We will show that, in machines

with only tenths of nodes, there is no need to have a distributed control in order to obtain an efficient cooperative cache.

Among the advantages of a centralized control we can point out its simplicity of implementation. A centralized control is much easier to implement than a distributed one. It avoids all the communication and synchronization problems. Moreover, centralized algorithms have better knowledge of the whole system as they keep all the information of the system. This global information allows the centralized designs to make better decisions to increase the performance of the whole system.

The main disadvantage of a centralized control resides in its lack of scalability. Sooner or later, a centralized system becomes a bottleneck and has to be redesigned in a distributed fashion. As we have already mentioned, in this chapter, we will study the viability of implementing a cooperative cache with a centralized control in small machines.

A second motivation behind this centralized version was getting a better understanding of the behavior of cooperative caches before trying to build a distributed version. A centralized version is always a useful first step before designing the distributed prototype. As it is much simpler, the results obtained are easier to follow and allow a better understanding of the problems that can be found in a cooperative cache. Once we have a good comprehension of the centralized server behavior, better distributed mechanisms can be designed as we can focus on the distributed part.

So far, cooperative caches have been based on the idea of encouraging locality. It is quite reasonable to think that trying to keep as many blocks as possible in the cache of the node that will use them will increase the system performance. The problem is that achieving this locality has a significant overhead. This extra overhead is basically due to reallocation of blocks, replication, maintenance of the coherence, time-consuming remote hits, etc. We are very skeptical about the advantages of a high local-hit ratio compared to the overhead needed to achieve it. We believe that achieving a high local-hit ratio is not the only way to achieve high-performance cooperative caches. We do not say that local hits are bad, but we want to show that it is not the only way to

design this kind of caches. In this first version, we will not encourage locality in order to test if there are other ways to design a fast cooperative cache.

In this first version, we also want to study the possibility of avoiding the coherence problem by not allowing replication. This lack of coherence mechanisms would also simplify the design of the cache. Avoiding these mechanisms by not allowing replication also seems to be in the line of not encouraging locality as replication was specially aimed at increasing the local-hit ratio.

Finally, this centralized version has to be implemented having in mind that a distributed one will follow. All the algorithms used have to be thought to be easily distributed. If this is not done in this way, many problems might be found when distributing the control. Thus, implementing an efficient distributed version might become a very complex task.

In conclusion, we present a cooperative cache with a centralized control as a first step before building a distributed second version. Anyway, it has to be efficient enough to be usable in environments where no distributed control is needed. In order to identify this first centralized version of our cooperative cache we have given it the name of PACA, which stands for PArallel CAche. Actually, as the most important part of this file system is the cooperative cache, we will also use the term PACA when referring to the whole file system.

4.2 FILE-SYSTEM ARCHITECTURE

Before getting into the details of the cooperative cache, it is necessary to present the architecture of the file system we are working on. As was discussed earlier, this work was designed to run on a parallel machine or network of workstations where each node has a micro-kernel operating system on top. All services, including the file system, are implemented by user-level servers which receive the client requests through ports.

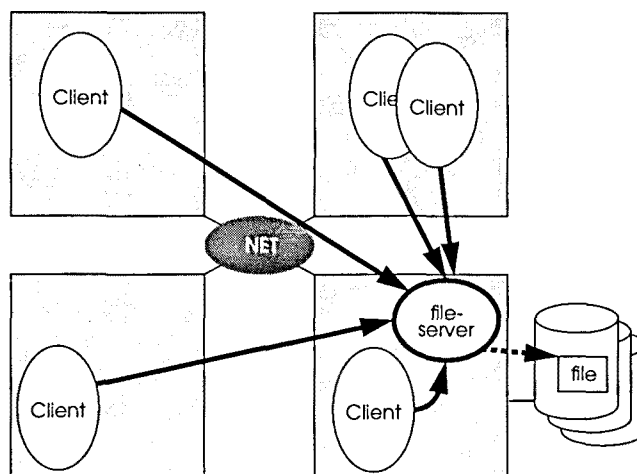


Figure 4.1 File-system architecture of the centralized version.

Our centralized version only needs a single file server which takes care of all the tasks related to the file system (Figure 4.1). This file server handles the user requests and sends replies when the work is completed. It also keeps the information about which blocks are cached in which nodes. Finally, it is also in charge of accessing the disk when a requested block is not in the cache.

This file server does not need to run on a dedicated node and can share the CPU with any number of clients. To make the service more efficient, the file server has a higher priority than the rest of the applications. This should not be a problem as the server was designed to consume very few resources. The only imposed restriction is that the server should run on the node where the disk, or disks, are connected. This is done to simplify the model easing the task of understanding the results.

To increase the performance of the write operations, PACA uses a delayed-write policy. The file-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].

4.3 BLOCK-DISTRIBUTION AND REPLACEMENT ALGORITHMS

4.3.1 Design Issues

In this subsection, we describe the ideas that have guided the design of both, the block-distribution and replacement algorithms. We have to have in mind that all of these ideas come from the thesis that achieving a high-local hit ratio is not the only way to achieve a high-performance cooperative cache.

If locality is not a key issue in the design, file blocks can be placed in any node. The important thing is to have the blocks in the cache, not their actual location. For this reason, we will place file blocks in any node regardless of the nodes which are going to use them.

Second, we will avoid replication. The system will not maintain several copies of the same block, not even in different nodes. Replication has been traditionally used in order to increase the local-hit ratio, but as in our case this is not a key issue there is no need to replicate blocks. Avoiding replication has two important advantages. The first one is that all the buffers in the cache are used to keep different blocks and this should increase the effectiveness of the cache. The second advantage is that avoiding replication also implies that all coherence problems are avoided as will be seen later (§ 4.4).

Finally, our system will also avoid reallocation of information. It is quite typical to keep moving blocks from one node to another to place them in the nodes which will use them. This increases the local-hit ratio but has the overhead of copying blocks through the network. As we want to see if a high-performance cache can be obtained without exploiting the local-hit ratio, it makes no sense to assume the overhead of these extra copies.

It is quite obvious that all these ideas go against the goal of achieving a high local-hit ratio. Placing blocks in any node regardless of the location of the clients obviously does

not help to have much locality. Furthermore, if replication is not allowed, at most one node can have the block in its local cache. And finally, if data reallocation is forbidden, there is no way to achieve a high local-hit ratio if blocks are requested from different nodes at different times. We will show that using these simple ideas as key issues in the design of a cooperative cache can produce systems as efficient, if not more, as trying to achieve a very high local-hit ratio.

4.3.2 Block Distribution Algorithm

The block distribution algorithm presented in this chapter is very simple. Given the file name (or file-ID) and the block number, the system computes a hash function to decide in which node to place this block. Using this algorithm no replication is allowed as the system will not keep the same file block twice in a single node. Furthermore, there is no way to reallocate blocks as the hash function will always give the same node for a given file block.

$$destination_node = fh(file_ID, \#block) = (rand(file) + \#block) MOD nodes \quad (4.1)$$

The hash function we have used is quite simple (equation 4.1). We assign a random number to each file the first time this file is used. When a block has to be located, we add this random number to the block number and the result of this operation modulo the number of nodes returns the location of the block in the cache.

4.3.3 Replacement Algorithm

To decide which file block has to be replaced by a new block, we use the well-known least-recently-used replacement algorithm (LRU). As the node of a given file block is predefined by the hash function, this replacement algorithm will only take into account the blocks in the destination node. This means that the server has to keep a different LRU-list for each node as there is no interaction between them. As this replacement

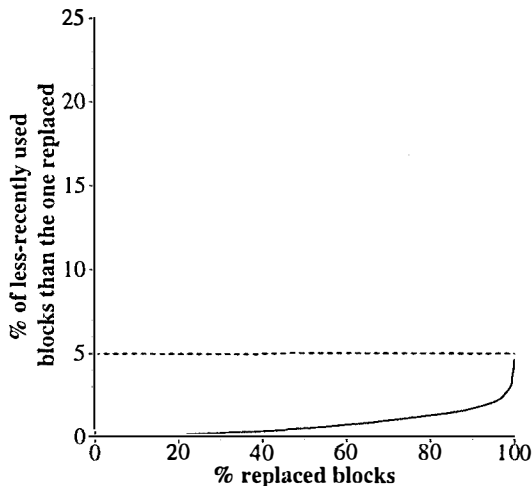


Figure 4.2 Approximation of the LRU-Interleaved to a global LRU.

algorithm is based on a LRU and the blocks are distributed in an interleaved fashion, we have named it LRU-Interleaved.

Although a different LRU list has been used for each node, it behaves nearly as a global LRU. Figure 4.2 presents a cumulative function that shows whether the replaced blocks were among the least-recently used in the whole cache. To build this graph we have used the results obtained in all the simulations presented in the performance section. We can see that 100% of the replaced blocks using our replacement algorithm were among the 5% least-recently-used blocks in the whole cache. Furthermore, 95% of the replaced blocks were among the 2.5% least-recently-used ones also in the whole cache. This shows that this replacement algorithm behaves much like a global LRU.

As each node is treated isolated from the rest of nodes, it should be very simple to distribute the control of the replacement algorithm which was one of the objectives.

4.4 COHERENCE MECHANISM

We have already seen in the state of the art (Chapter 2) that the coherence of the cached data is one of the most important issues in a cooperative cache. Furthermore,

if a *Unix* semantic is to be offered to the user, it becomes even more complex and time consuming. In this work, we present a way to achieve this strict semantic in a very simple and efficient way.

This task should not be a very difficult task in a centralized version. As the server has the information of all the blocks, implementing a cache-coherence mechanism should be quite simple. The file server only has to invalidate the obsolete block in its data structures and there are no notifications to be done. The problem is that we want to present a mechanism that, not only works efficiently in a centralized version, but it is also feasible in a distributed one.

The origin of all coherence problems comes from the fact that replication of blocks is allowed. As we have mentioned above, this is usually done to increase the local-hit ratio. As we want to see what happens when this issue is not taken into account, we have decided to avoid replication which also means that our cache will not have coherence problems. The only copy of a block that can be found in the cache always has the up-to-date information and no invalidations are ever needed.

As can be easily seen, this solution to the coherence problem is also ready to be used in a distributed environment. If good performance can be achieved without encouraging local hits, this is a very simple and efficient solution to the problem.

4.5 ADAPTABILITY TO THE VARIATION OF NODES AND VIRTUAL MEMORY

Finally, there are some environments where it is very important to design mechanisms for the cache to adapt to the variation in the number of nodes and cache sizes. For example, in a network of workstations we should allow the owner of a workstation to isolate its node from the network during a period of time. This user may need the full power of the machine for a given reason and the system should allow the user to have all this power. We should also allow the owner of a workstation to decide the quantity of memory that can be used by the cooperative cache depending on its own needs.

Allowing this kind of variation in the availability of resources can be easily done with very few modifications to the original block distribution algorithm.

Whenever a given node decides to start taking part in the cooperative cache, it only has to send a message to the server. From this point on, the server can use the buffers in this node to cache file blocks. On the other hand, if a given node decides to leave the cooperative cache the system has to go through three phases. First, the node requests permission to the file server to leave the cache. Second, the file server sends to disk all modified blocks that are kept in that node. And third, the node is notified that it can use its buffers freely. This variation in the number of nodes implies a change in the hash function to only use the accessible nodes. It may also be necessary to reallocate the already cached blocks as the new hash function may have placed them in a different node. This overhead is not very important as we assume that increasing, or decreasing, the number of nodes should not be a very frequent operation. Thus, it is not a problem to have some extra work when this situation appears. Furthermore, this can be done without stopping the service if an adequate data structure is used during this transition interval.

In a similar way, our mechanism could allow a given node to change the size of its local cache in favor of the virtual-memory system as proposed in Sprite [NWO88]. This modification in the number of buffers only needs a message to the server. Sometimes, it will also mean that some modified blocks have to be written to disk before passing the buffers to the virtual-memory system.

This variable cache size may affect the performance of the system. As a given subset of all the file blocks will always be placed on the same node, if that node only offers a very small cache, the effectiveness of the cache for those blocks will be very low. This problem has not been solved in this first version but a solution will be proposed in the distributed version presented in Chapter 5.

4.6 EXPERIMENTAL RESULTS

In this section, we present the results obtained by the centralized version of the cooperative cache. With these first results we try to obtain a first validation of the ideas we propose in this work. We want to test if a high-performance cooperative cache can be obtained without exploiting locality. We also want to test if avoiding the coherence problem by avoiding replication does not degrade the system performance. Finally, we want to reach a good understanding of the effects these new ideas have on the system performance in order to start the design of the distributed version.

These results obtained by our system were compared to the ones obtained by xFS. Actually, they were compared with a centralized xFS version implemented by ourselves. This comparison has not been done with the original distributed version because we wanted to compare PACA with a file system that did not have the overhead of the distributed control. Furthermore, this centralized version did not have the need of a coherence mechanism as the distributed version did. All these simplifications of xFS will ease the task of comparing both systems. Anyway, we should keep in mind that the xFS file system used in this chapter is a simplification of the original one which should be a little bit more efficient than the distributed version as long as it does not become a bottleneck. This centralized version should have similar read performance but write operations should be faster than in the distributed version as it does not have to spend time on the coherence algorithm.

In these measures, we have only used one of the workloads (Sprite) presented in chapter 3. This was done because the CHARISMA trace file was taken from a machine with 128 nodes. This machine was too large for a centralized version and we detected that our centralized file system was a bottleneck. On the other hand, the Sprite workload was taken from a network of workstations with 50 nodes which is small enough for a centralized server to handle efficiently.

The parameters we used in these simulations are the ones that emulate a network of workstations which were presented in chapter 3. These parameters have been used because they were also used by the NOW team to test their cooperative cache. As

	Read		Write	
	PACA	xFS	PACA	xFS
Average operation time	3454μs	3564μs	1051μs	1223μs
Local-hit ratio	2.9%	65.5%	2.0%	43.5%
Remote-hit ratio	82.5%	20.4%	47.9%	0.5%
Global-hit ratio	85.4%	85.9%	49.9%	44.0%

Table 4.1 Average operation times and hit ratios obtained using the centralized versions of PACA and xFS.

we want to compare both caches, it seems quite appropriate to use the same network parameters. To study the behavior of both algorithms in different environments, we will also examine the influence several factors such as interconnection-network bandwidth, local-cache size, etc. have on the overall performance.

4.6.1 Algorithm Comparison

The first thing we want to do is a comparison between both file systems in a network of workstations. The results of this comparison are presented in Table 4.1 where we can see that PACA behaves slightly better than the centralized version of xFS. Read operations are 3.1% faster and write operations are 14% faster. This gain may not seem too impressive, specially the one-obtained by read operations. This is true, but the important thing is that similar performance results can be achieved without encouraging a high local-hit ratio (Table 4.1). Furthermore, we have also designed a simpler algorithm that achieves a similar read performance and a higher write performance. To compare these performance results in greater detail, we will first examine the read operations and the write operations will follow.

Before starting to explain these results, it is necessary to examine Figure 4.3 along with Table 4.1 in detail. In this graph, we show the whole time spent performing read and write operations by both file systems. It is important to notice that the values represented are not the average operation times, but the total time spent by the file system working on those operations during the whole simulation. Each bar is divided

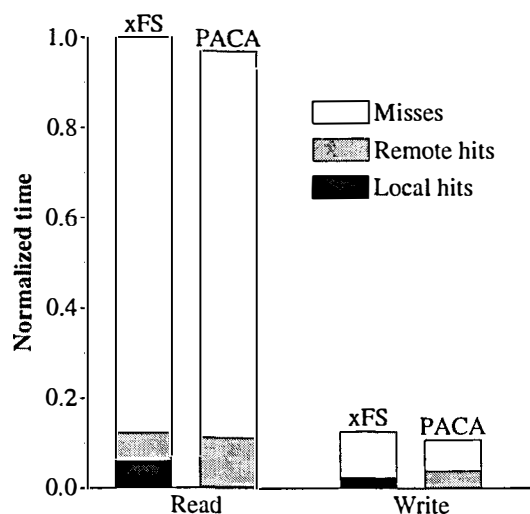


Figure 4.3 Normalized time spent by the centralized version of both file systems serving read and write operations.

into three parts that represent the portion of time spent by the file system working on local hits, remote hits and misses. All these times have been normalized to the time spent by xFS performing read operations in order to make this graph easier to understand.

The first thing we observe is that although PACA obtains a much lower read local-hit ratio, very similar average-read times are achieved. On the other hand, if we examine the global-hit ratio and the time spent working on global hits, we observe that they are very similar in both file systems. The reason behind this behavior is the different overhead produced by a remote hit in both file systems. The average time needed to serve a remote hit in xFS is 3.3 time larger than the one needed in PACA. On the other hand, PACA only has 4 times more remote hits than xFS. This means that most of the overhead produced by the larger number of remote hits is outweighed by their higher speed. The rest of the overhead is balanced by the time spent performing local hits as many more have to be served. This difference in the time needed to serve a remote hit between both file systems is explained in the following paragraphs.

As xFS tries to increase the local-hit ratio, each remote hit implies a copy of the whole file block from the remote cache to the local cache and a copy of the requested bytes

from the local cache to the user. In PACA, a remote hit only needs one memory copy from the cache to the user address space where only the bytes requested by the user are copied. PACA avoids one copy and the information that goes through the interconnection network is smaller as only the requested bytes are copied, not the whole block.

Remote hits take longer in xFS than in PACA also due to the overhead produced by the forwarding of blocks. In xFS, if the block to be replaced is the only copy in the cache, it is not discarded but forwarded to a different node. Actually, a block can only be forwarded twice without being referenced by a client. This forwarding is usually done out of the critical path of the read operation except when all auxiliary buffers are occupied. Auxiliary buffers are those portions of memory where blocks are kept waiting to be forwarded, or sent to disk, until the operation is finished. When there is a lack of such buffers, the block forwarding has to be done before the new block is copied into the buffer. This takes some time and increases the time needed to serve remote hits.

If we go back to Figure 4.3, we can also see that, although the miss ratio is very similar, xFS spends a little more time working on them. This also happens due to the forwarding of blocks. As it has just been explained with the remote-hits, this forwarding cannot always be done away from the critical path of the read operation.

Let's now compare the behavior of the write operations. The first important thing is that write operations are very fast as they do not usually need to access the disk. This happens because a delayed write policy is used. As write operations are very fast, any overhead has a significant impact on its performance. A clear example can be found in the forwarding of blocks. Both, read and write operations, produce a similar amount of blocks being forwarded, but they have a higher influence on the write than on the read performance. This is the main reason behind the performance loss detected in xFS when compared to PACA.

A second difference, but much less important, is the number of misses on dirty. A miss on dirty appears when the block that has to be replaced has been modified but it has not been written to the disk yet. This block has to be written to disk before it is

discarded or forwarded. The number of misses on dirty found in xFS is larger than in PACA. This happens because in xFS, blocks are replaced when only about one third of its life time has been consumed. As a block is usually forwarded twice before it is finally discarded, the first time it is forwarded happens after the first third of its life. When this happens, if the block is dirty, it has to be sent to disk before it is forwarded. On the other hand, in PACA, blocks are replaced after all its life time has been consumed. As this time is longer, the probability of a syncer intervention is much higher. If the syncer is able to send the block to the disk before it is discarded or forwarded, this will not have to be done in the critical path of the operation. This explains the higher number of misses on dirty found in xFS when compared to PACA. It is important to notice that the system also tries to perform these physical writes away from the critical path of the operation using a similar mechanism as the one used to delay the block forwarding. The problem is that there are some times when the system runs out of auxiliary buffers to keep this delayed operations. In this case, the physical write has to be done right on the spot.

4.6.2 Interconnection Network Bandwidth Influence

As we have already explained, one of the things we want to study is the effect of not stressing the number of local-hits. This effect will depend on the interconnection-network bandwidth. If a very slow network is available, the importance of the number of local hits will become much more significant in the overall system performance.

To study this influence, we have run several simulations varying the interconnection-network bandwidth. In Figure 4.4, we present the results obtained in those simulations. In the graph, the X axis represents the ratio between the bandwidth that can be achieved copying data within a single node and the network bandwidth.

As can be seen in Figure 4.4, read operations are faster in PACA than in xFS until the interconnection network is about 6 times slower than the memory bandwidth. As the network slows down, the time needed to serve remote hits also increases. This has a higher effect on PACA than on xFS as our file system has four times more remote hits.

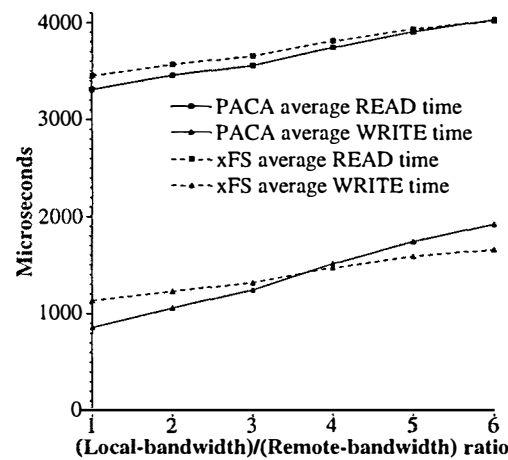


Figure 4.4 Influence of the interconnection-network bandwidth on the centralized version of PACA and xFS.

	0.5MB	1MB	2MB	4MB	8MB	16MB
PACA	66.2%	70.9%	76.1%	78.2%	81.1%	85.4%
xFS	63.4%	68.6%	74.5%	78.0%	81.3%	85.9%

Table 4.2 Influence of the "local-cache" size on the read global-hit ratio using the centralized version of PACA and xFS.

The influence of the interconnection-network bandwidth has a more noticeable effect on the write operation. This happens because, in PACA, nearly all writes mean a remote copy as file blocks are usually placed in a different node than the clients requesting them. In the other hand, a write operation in xFS nearly always means a memory copy within the same node. For this reason, a slow down in the network bandwidth has a much more noticeable influence on PACA than on xFS.

4.6.3 "Local-Cache" Size influence

Another parameter that has to be studied is the size of the "local-caches". All simulations ran so far have been done using a 16-MByte local cache. Let's now present the results obtained when the machine has "local-caches" ranging from 0.5 MBytes to 16 MBytes (Figure 4.5).

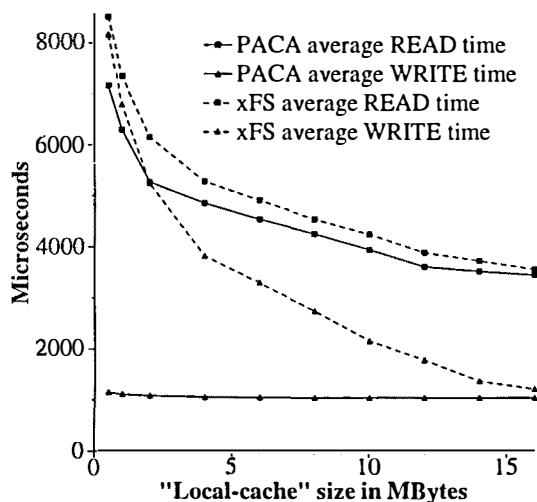


Figure 4.5 Influence of the "local-cache" size on the centralized version of PACA and xFS.

The first thing we observe is that the bigger the local caches are, the better the performance on read operations is. This is basically due to the higher global-hit ratio obtained when bigger caches are used (Table 4.2).

We also observe that there is a difference in the global-hit ratio obtained by both file systems when local caches are small. This difference is due to the better utilization of the cache made by PACA. As no replication is allowed in PACA, all the buffers are used to keep useful file blocks. On the other hand, xFS allows replication and a percentage of the buffers are used to keep replicated blocks when they could be holding other important blocks which were recently discarded. This difference becomes very important with small caches. As this size increases, the buffers used for replication become less important and end up having no influence. Actually, we can also observe that when large local caches are used, xFS obtains a slightly better global-hit ratio. This is because when big caches are used their replacement algorithm has a slightly better behavior. Anyway, the difference is so small that no noticeable loss in performance has been detected in the system.

Write operations have a very different behavior. While in PACA, the write operation is not affected by the cache size, when xFS runs with small caches the same operation

obtains a very poor performance. In PACA, write operations are not affected by the size of the cache because they are not affected by the global-hit ratio. A write miss and a write hit take exactly the same amount of time. In both cases, a copy from the user to the cache is needed. The only difference appears when writing a block implies reading the previous version kept in the disk as only a portion of the block will be modified. This only happens in 1% of the write operations as most of them append information or modify the whole block. A completely different behavior is observed in xFS. The smaller the cache is, the worse the write performance is. This is basically due to a couple of reasons. First, as the cache is smaller, there are less replicated blocks and more blocks have to be forwarded. Second, as the local cache is smaller, the probability of discarding a dirty block increases. As was explained earlier both, forwarding blocks and cleaning blocks, increase the time of write operations.

4.6.4 Number of Links Influence

A parallel machine may have several links to the interconnection network. These links allow a node to send and receive several data blocks in parallel. To study the influence the number of links has, we have done several simulations using 1, 2 and 4 links (Figure 4.6). We should note that with the current network parameters, 4 links is an impossible combination as the network is only twice slower than the memory. These results, though, may help us understand how would the cache behave if a slower network were used.

Let's first examine the impact on read operations. As we can see in Figure 4.6, very little improvement is achieved on read operations. This is because the interconnection network is fast enough to handle read requests and very few requests are sent in parallel.

On the other hand, write operations have a higher influence. Increasing the number of links also improves the write average time in PACA. As most of the times, a write operation only implies a *memory_copy* between two nodes, it is more sensitive to network bandwidth improvements. Besides, the number of bytes requested in write operations is higher than the ones requested in read operations. Thus, more copies can be started in parallel.

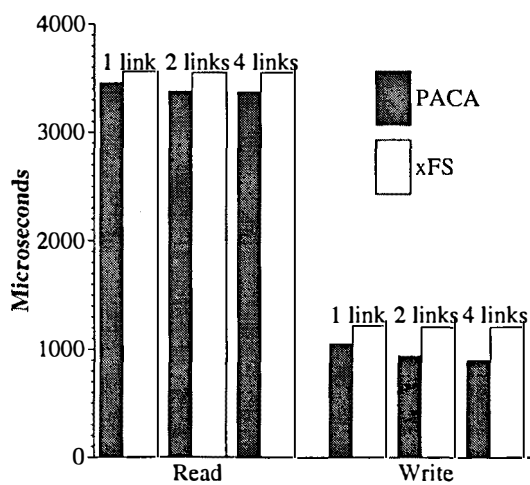


Figure 4.6 Influence of the number of links to the interconnection network on the centralized version of PACA and xFS.

We can also observe that xFS is not significantly affected by the number of links. This is due to the high local-hit ratio which avoids most of the network communication.

4.6.5 Single Server Performance

During all simulations, the time needed to perform a local hit, remote hit, miss on clean, etc. has been measured. Afterwards, these results have been compared with the theoretical values and no significant differences have been found. In this environment, and using the Sprite workload, no bottleneck was detected neither in the CPU, nor in the interconnection network, nor in the physical disk access.

Another important issue is to examine the effect this file server had on the user work done on the same node. This work was slowed down about 10%. This means that applications can share the processor with the centralized server as long as speed is not crucial.

These results prove that a centralized single server can handle the requests of a small network. On the other hand, we have also seen that larger networks like the one we have simulated with 128 nodes are too big for a centralized single server. We started

to simulate the CHARISMA workload on a parallel machine with 128 nodes and we immediately observed that the centralized version became a big bottleneck which led us not to continue those simulations.

4.7 CONCLUSIONS

In this chapter, we have designed and simulated a cooperative cache with a centralized control. We have proven that a centralized control should not be discarded when designing algorithms for parallel machines or networks of workstations with tenths of nodes. We have seen that our centralized version of the cooperative cache works quite well with a network of workstations of 50 nodes. We have also seen that this centralized control is not enough if the network is larger. We have tested it with 128 nodes and the file system has been detected as an important bottleneck. This means that a distributed version has to be designed if large machines are to be used.

The second important conclusion is that encouraging high local-hit ratios is not the only way to achieve high-performance caches if the machine has a high-bandwidth interconnection network. We have designed a cache with an extremely low local-hit ratio that obtains similar or even better results than xFS which places much interest on local hits. This happens because keeping a high local-hit ratio places an important overhead on the remote-hits that outweighs the benefits that can be obtained.

We can also conclude that the write global-hit ratio does not affect the write operations performance. As write misses take more or less the same amount of time than a write hit, both, hits and misses, are not very different from the performance point of view.

Another very important conclusion extracted from this centralized version is that avoiding the cache-coherence problem by avoiding replication is a good solution to the problem. We have seen that, as achieving a local-hit ratio is not the only good solution to achieve high performance, very efficient caches can be designed avoiding replication.

Finally, we have presented a distributed replacement algorithm that behaves much like a global LRU. This algorithm can be easily distributed as no communication between nodes should be needed in the distributed version.

More information about this centralized version, its characteristics and its performance can be found in earlier papers [CGL96b, CGL95].

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

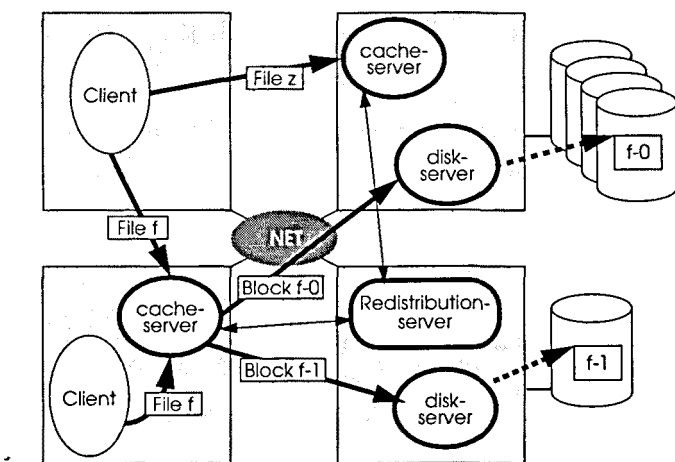


Figure 5.1 PAFS architecture.

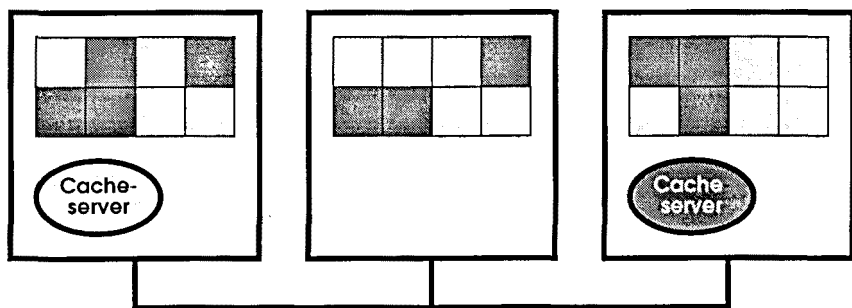


Figure 5.2 An example of two cache-server partitions in a three-node machine.

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.

$$cache_server = fh(file_ID) = (rand(file))MOD\ cache_servers \quad (5.1)$$

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 priori. 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.

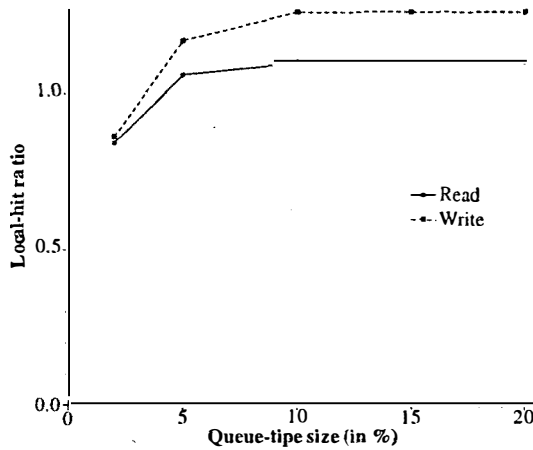


Figure 5.3 Influence of the queue-tip size on the local-hit ratio (PM/CHARISMA).

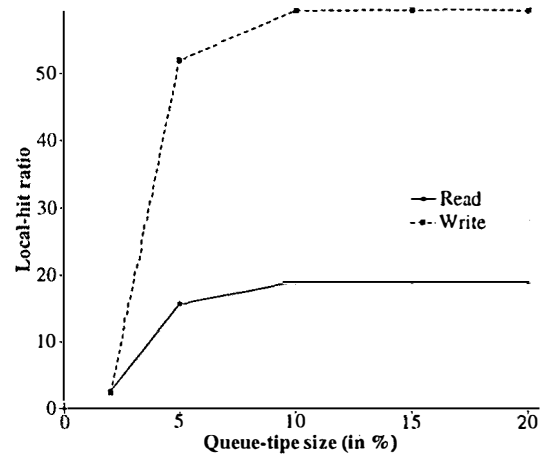


Figure 5.4 Influence of the queue-tip size on the local-hit ratio (NOW/Sprite).

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,

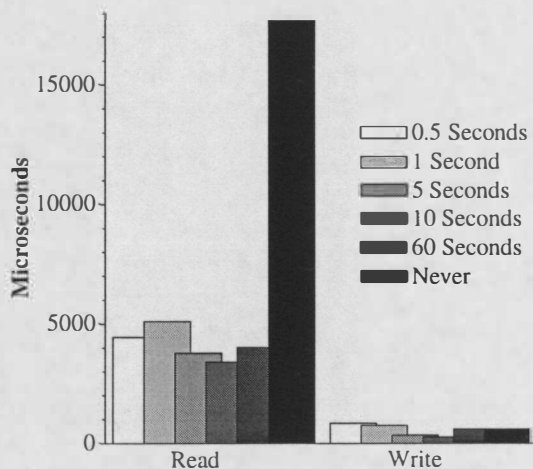


Figure 5.5 Influence of the interval between redistributions (PM/CHARISMA).

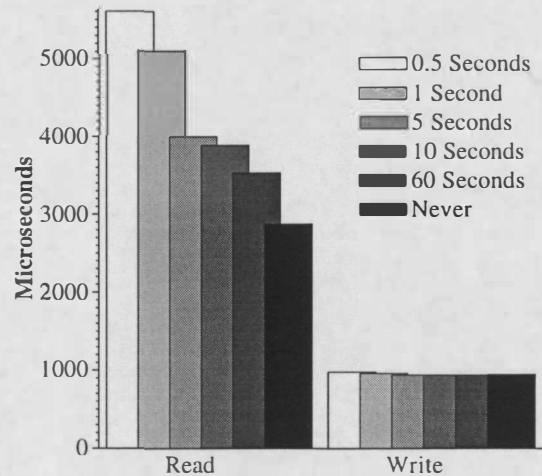


Figure 5.6 Influence of the interval between redistributions (NOW/Sprite).

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.

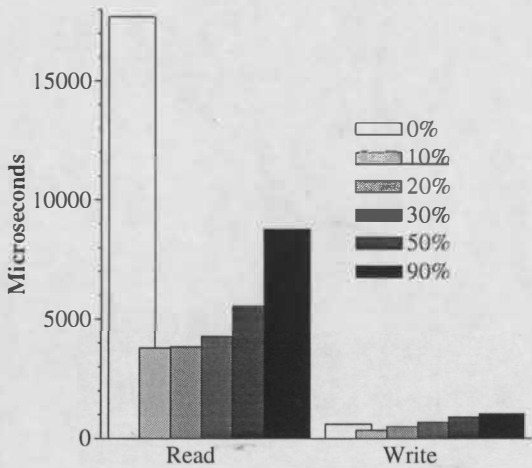


Figure 5.7 Influence of the limitation of lost buffers (PM/CHARISMA).

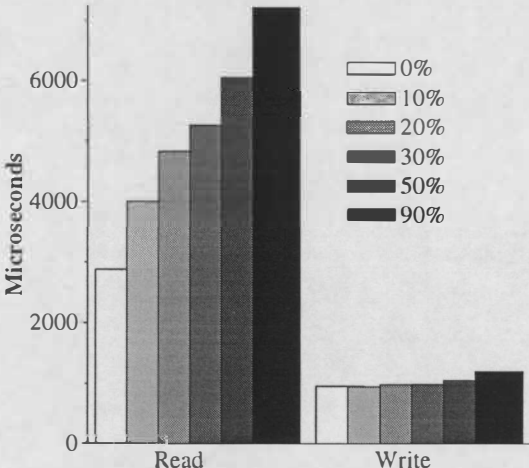


Figure 5.8 Influence of the limitation of lost buffers (NOW/Sprite).

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

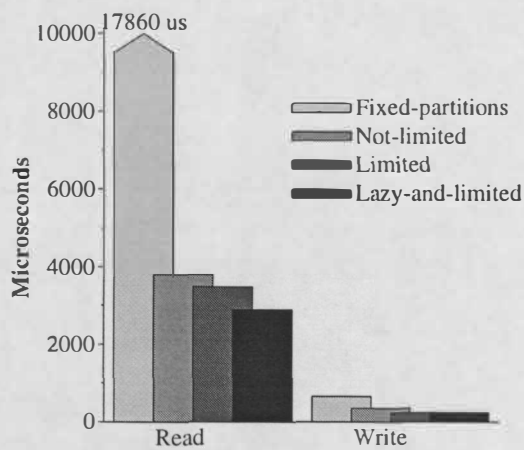


Figure 5.9 Influence of the limitation of gained buffers (PM/CHARISMA).

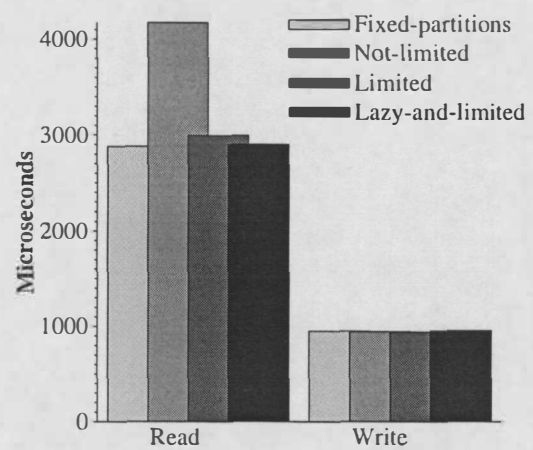


Figure 5.10 Influence of the limitation of gained buffers (NOW/Sprite).

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 μ s	9273.16 μ s
Average write time	245.03 μ s	855.45 μ s
Local read <i>hit</i> ratio	0.79%	63.99%
Remote read <i>hit</i> ratio	94.18%	28.01%
Global read <i>hit</i> ratio	94.97%	92.00%
Local write <i>hit</i> ratio	0.86%	86.32%
Remote write <i>hit</i> ratio	97.87%	11.95%
Global write <i>hit</i> ratio	98.73%	98.27%

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

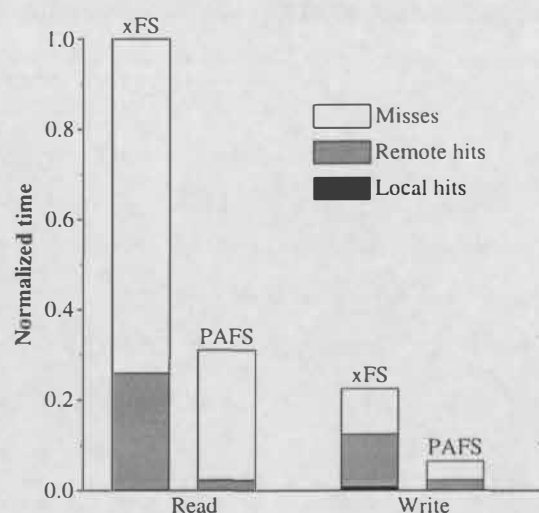


Figure 5.11 Total normalized time spent by PAFS and xFS performing read and write operations (PM/CHARISMA).

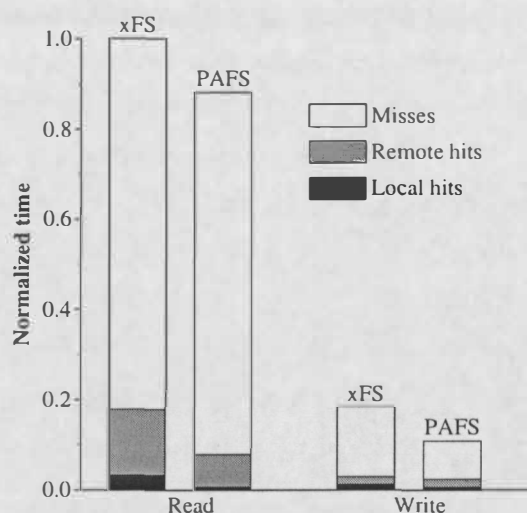


Figure 5.12 Total normalized time spent by PAFS and xFS performing read and write operations (NOW/Sprite).

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 than 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).

$$PAFS_R_hit = R_memory_copy(request_size) \quad (5.2)$$

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 that 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 μ s	3294.77 μ s
Average write time	957.34 μ s	1635.77 μ s
Local read <i>hit</i> ratio	12.22%	65.51%
Remote read <i>hit</i> ratio	73.15%	20.31%
Global read <i>hit</i> ratio	85.37%	85.82%
Local write <i>hit</i> ratio	24.71%	43.52%
Remote write <i>hit</i> ratio	25.21%	6.35%
Global write <i>hit</i> 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.

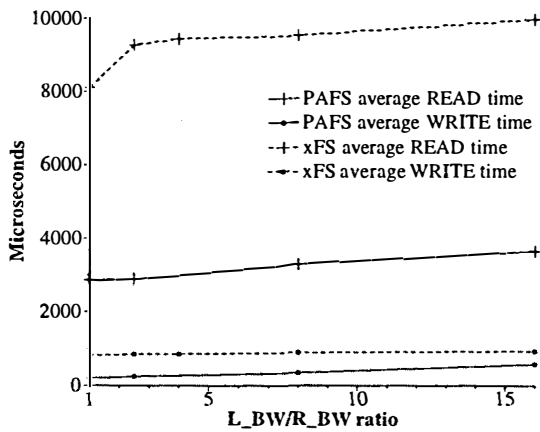


Figure 5.13 Interconnection-network bandwidth influence in the average read and write operation times (PM/CHARISMA).

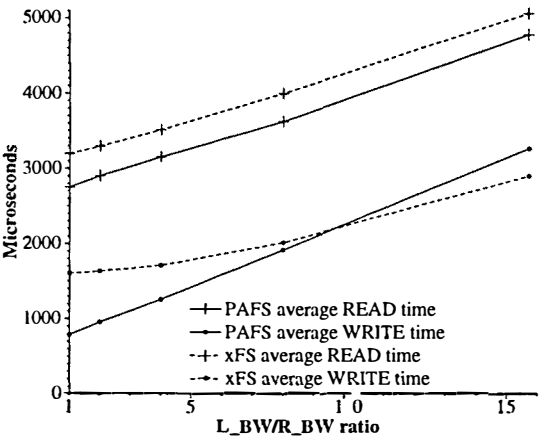


Figure 5.14 Interconnection-network bandwidth influence in the average read and write operation times (NOW/Sprite).

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.

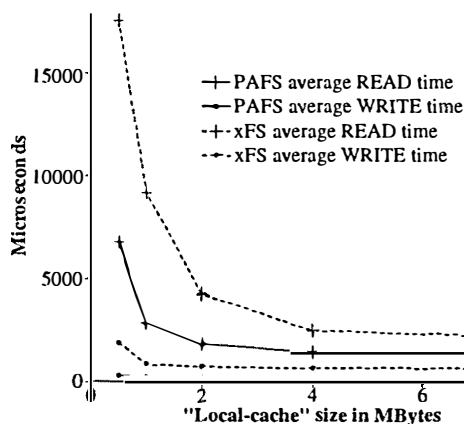


Figure 5.15 "Local-cache" size influence in the average read and write operation times (PM/CHARISMA).

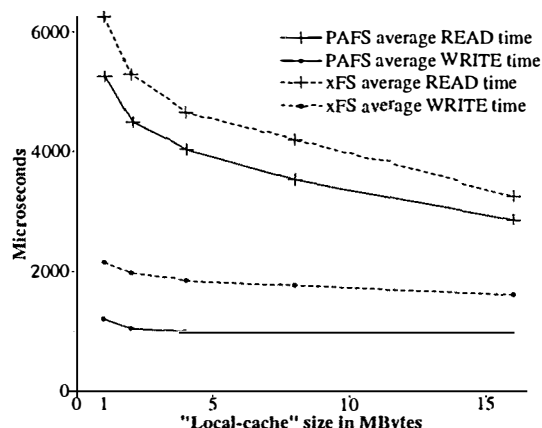


Figure 5.16 "Local-cache" size influence in the average read and write operation times (NOW/Sprite).

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.

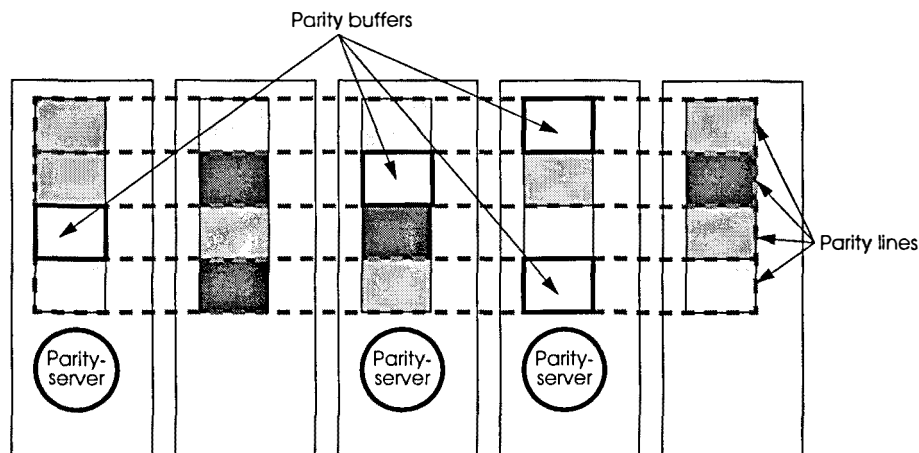


Figure 6.1 Distribution of the cache buffers in parity lines.

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 through 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 an environment with a lot of file sharing between applications. It would also not be sufficient in an environment where a lot of communication between applications is done through 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-

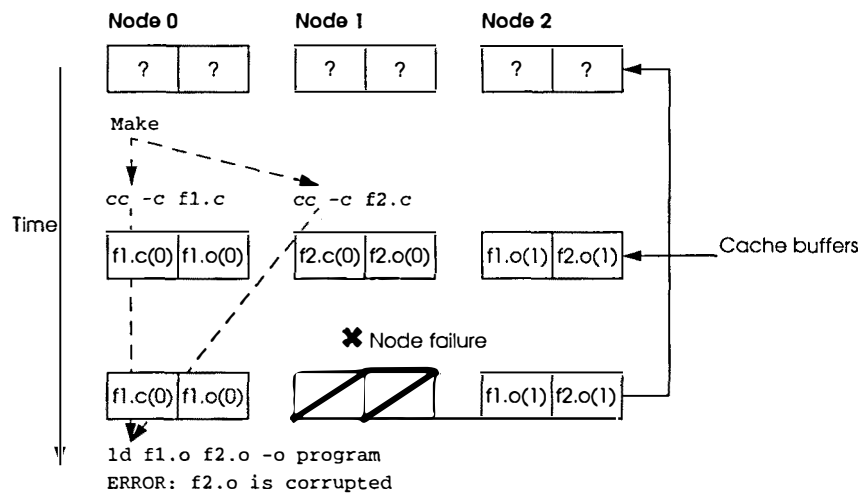


Figure 6.2 Example of a parallel make that needs a high tolerance level. The contents of each cache block is represented by the file name and the block number in parenthesis (i.e. name(#block)).

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.

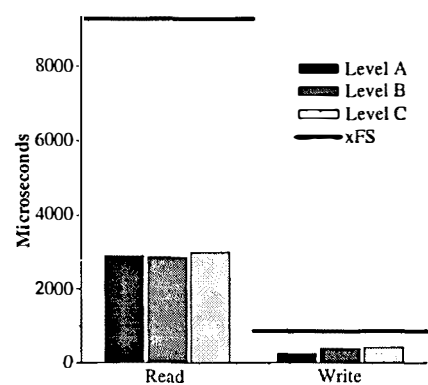


Figure 6.3 .Performance of the fault-tolerance levels (PM/CHARISMA).

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

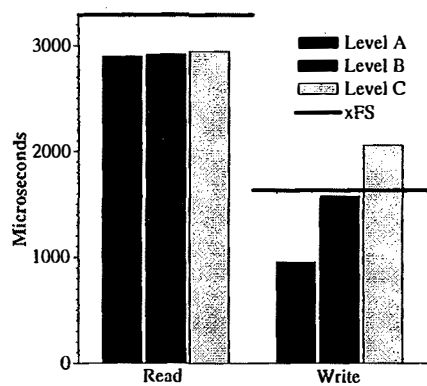


Figure 6.4 Performance of the fault-tolerance levels (NOW/Sprite).

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].

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 many 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

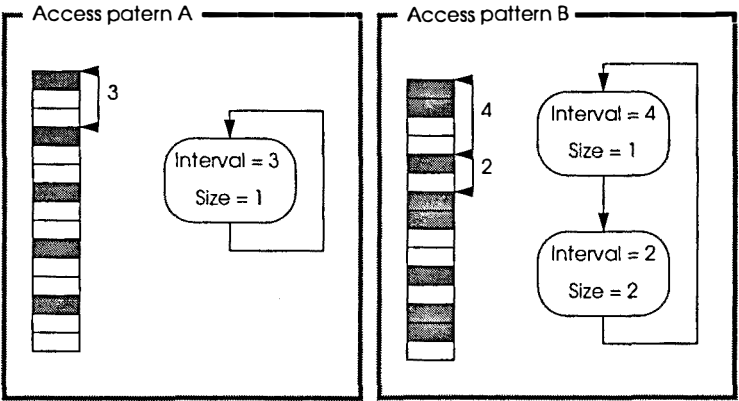


Figure 7.1 Example of a couple of access patterns and their prefetch graphs.

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

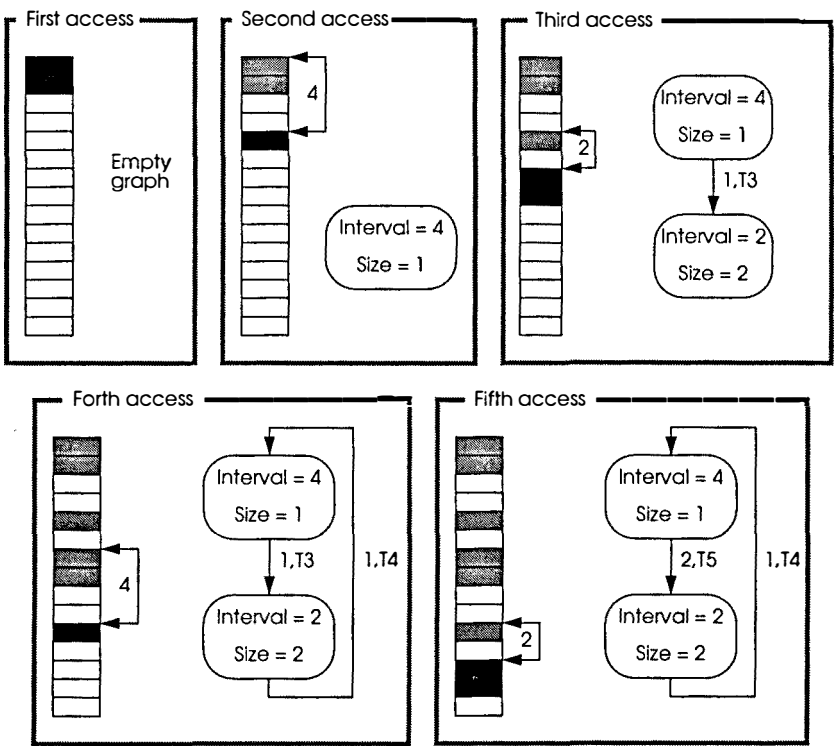


Figure 7.2 Example of a graph creation.

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**

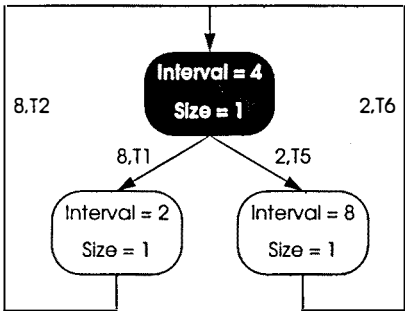


Figure 7.3 Example of a prefetch graph where nodes have more than one link that start from it.

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 an 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.

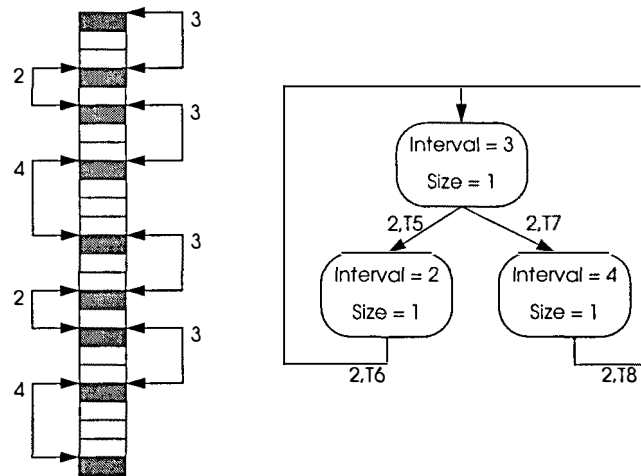


Figure 7.4 Example of a regular access pattern that cannot be prefetched by ISG.

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 pattern 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 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).

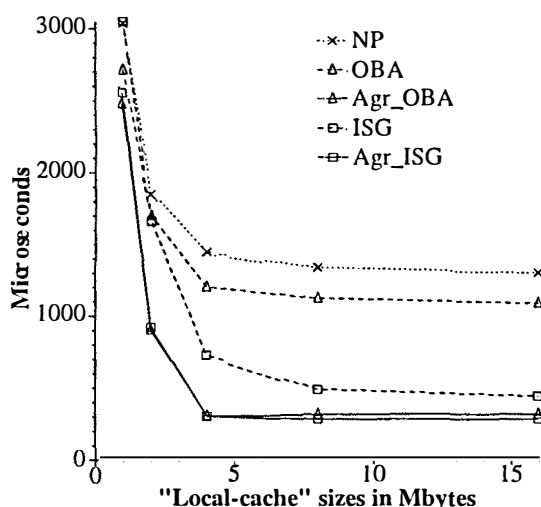


Figure 7.5 Average read operation time obtained by the presented prefetching algorithms on PAFS (PM/CHARISMA).

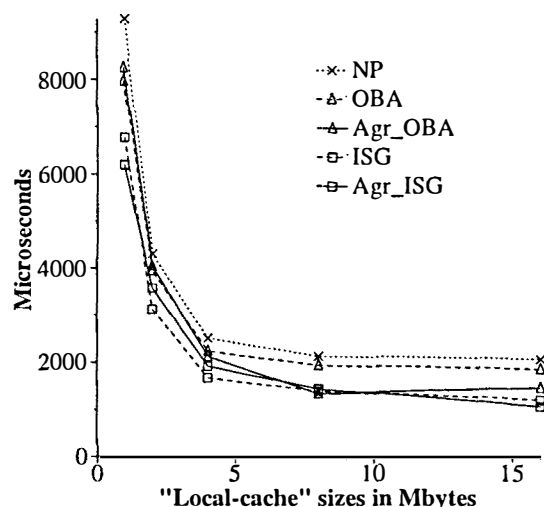


Figure 7.6 Average read operation time obtained by the presented prefetching algorithms on xFS (PM/CHARISMA).

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.

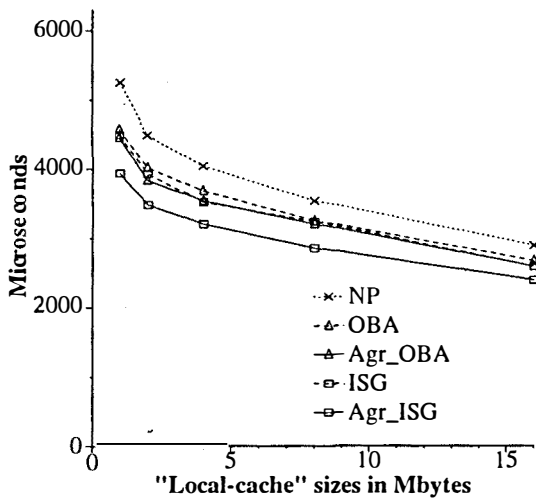


Figure 7.7 Average read operation time obtained by the presented prefetching algorithms on PAFS (NOW/Sprite).

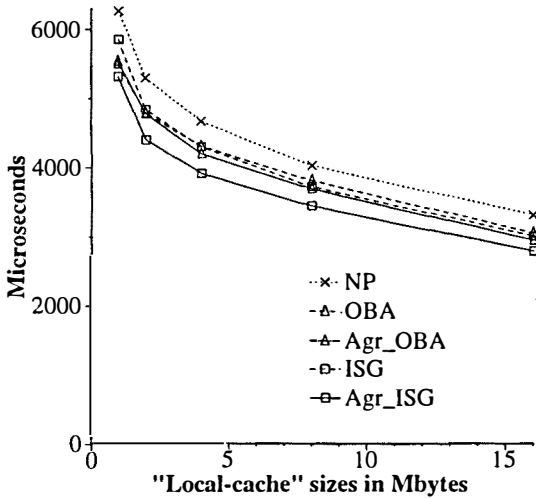


Figure 7.8 Average read operation time obtained by the presented prefetching algorithms on xFS (NOW/Sprite).

	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 to the 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 misleading 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-

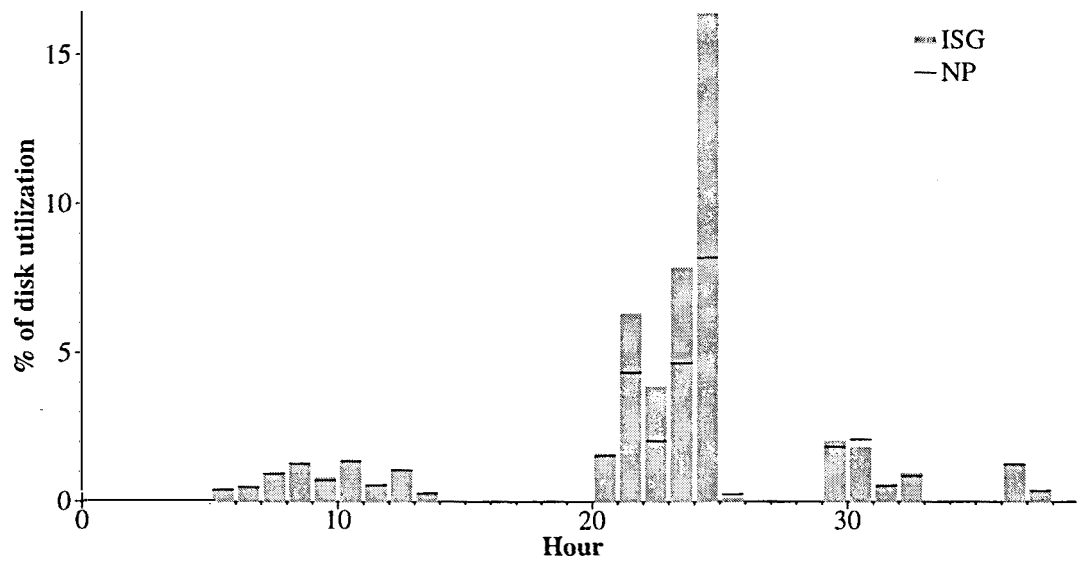


Figure 7.9 Load placed on the disks by ISG with a 1MByte local cache.

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.

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 than 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 policies. 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 commercial 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.

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 resetting 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 from 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 may have the following roles: client, manager, storage server and cleaner. Actually, a given

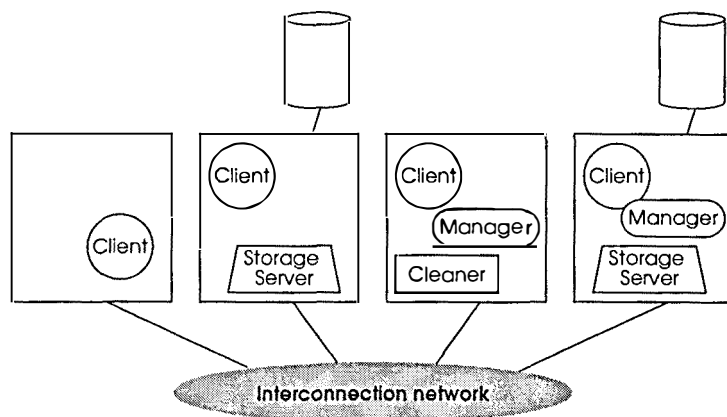


Figure A.1 Possible xFS configuration in a network of workstations.

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

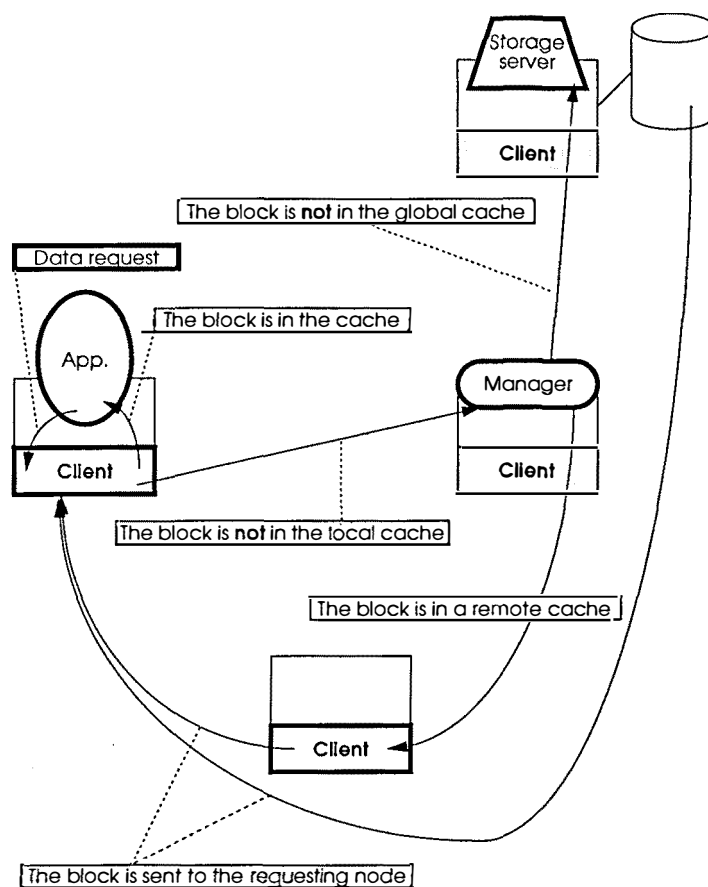


Figure A.2 Steps followed by xFS to fulfill a client request.

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 *Proceedings of the 23rd Annual International Symposium on Fault-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 multiprocesadores 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, Departament 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 Computing 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 massively 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 Asociation, 1992.
- [Dah95] DAHLIN, M. D. *Serverless Network File System*. PhD thesis, University of California at Berkeley, 1995.
- [Dah96] DAHLIN, M. D. Personal communication. 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 multi-processor file system. *ACM SIGARCH*, 15(15):32–39, 1989.
- [DSH⁺94] DRAPEAU, A. L., SHIRRIFF, 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 Paralelos*. 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 efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th Symposium on Operating Systems Principles*. 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. *Flash Memory*, 1994.
- [Jai91] JAIN, R. *The art of computer systems performance analysis*. Wiley Professional Computing, 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 Computer 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 Modeling 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. Sequential program prefetching in memory hierarchies. *IEEE Computer*, pages 7–2, December 1978.

- [Smi85] SMITH, A. J. Disc cache - miss ratio analysis 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 clustering: 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 DRESSER, 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.