11-1996

Galley: A New Parallel File System for Parallel Applications

Nils Nieuwejaar
Dartmouth College

Galley: A New Parallel File System For Scientific Workloads

Nils Nieuwejaar

Department of Computer Science
Dartmouth College, Hanover, NH 03755-3510
nils@cs.dartmouth.edu

A Ph.D thesis from the Department of Computer Science, Dartmouth College.
Available as Technical Report PCS-TR96-300, at URL ftp://ftp.cs.dartmouth.edu/TR/TR96-300.ps.Z

Abstract

Most current multiprocessor file systems are designed to use multiple disks in parallel, using the high aggregate bandwidth to meet the growing I/O requirements of parallel scientific applications. Most multiprocessor file systems provide applications with a conventional Unix-like interface, allowing the application to access those multiple disks transparently. This interface conceals the parallelism within the file system, increasing the ease of programmability, but making it difficult or impossible for sophisticated application and library programmers to use knowledge about their I/O to exploit that parallelism. In addition to providing an insufficient interface, most current multiprocessor file systems are optimized for a different workload than they are being asked to support.

In this work we examine current multiprocessor file systems, as well as how those file systems are used by scientific applications. Contrary to the expectations of the designers of current parallel file systems, the workloads on those systems are dominated by requests to read and write small pieces of data. Furthermore, rather than being accessed sequentially and contiguously, as in uniprocessor and supercomputer workloads, files in multiprocessor file systems are accessed in regular, structured, but non-contiguous patterns.

Based on our observations of multiprocessor workloads, we have designed Galley, a new parallel file system that is intended to efficiently support realistic scientific multiprocessor workloads. In this work, we introduce Galley and discuss its design and implementation. We describe Galley's new three-dimensional file structure and discuss how that structure can be used by parallel applications to achieve higher performance. We introduce several new data-access interfaces, which allow applications to explicitly describe the regular access patterns we found to be common in parallel file system workloads. We show how these new interfaces allow parallel applications to achieve tremendous increases in I/O performance. Finally, we discuss how Galley's new file structure and data-access interfaces can be useful in practice.

Acknowledgments

I'd like to thank NASA Ames Research Center for their generous support throughout my graduate career. They have provided financial support, given me access to their computer facilities, and have allowed me to draw on the expertise of many of their people. Among the individuals at NASA Ames who provided tremendous support were Jeff Becker, Russell Carter, Sam Fineberg, Art Lazanoff, Leigh Ann Tanner, and especially Bill Nitzberg.

During the course of this work, there have been many people with whom I've had extensive discussions, which helped me clarify my ideas, as well as led me to new ideas. Among these people are Mike Best, Carla Ellis, Orran Krieger, and Apratim Purakayastha.

There were a number of people who subjected themselves to the Galley Parallel File System while it was in varying stages of completeness and correctness. Their willingness to assault me with bug reports at any time of the night or day certainly led to Galley being a more stable and useful piece of software. Those people were Matt Carter, Tom Cormen, Melissa Hirschl, Jon Howell, Mark Montague, Joel Thomas, and Sivan Toledo.

I'd like to thank Tom Cormen, John Danskin, and Andrew Grimshaw, who served on my thesis commitee. In addition to reading, commenting on, and signing this thesis, all were free with their time, suggestions, and advice (solicited and otherwise) throughout this whole process. I'd especially like to thank my advisor David Kotz, who helped me find a research area that was challenging and interesting, who gave me direction and guidance when I needed it, and who spent more hours than I care to imagine reading and rereading countless drafts of papers, technical reports and this thesis.

Finally, I'd like to thank my wife Christine for everything else.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

While the speed of most components of massively parallel computers has been steadily increasing for years, the speed of the I/O subsystem has not been keeping pace. Hardware limitations are one reason for the difference in the rates of performance increase, but the slow development of new multiprocessor file systems is also to blame.

The successful design of computer systems (both hardware and software) depends on a thorough understanding of their intended usage. A system's designer optimizes the policies and mechanisms of the system for the cases expected to be most common in the typical workload. In the case of multiprocessor file systems, however, there was little or no information about the nature of a 'typical' workload. As a result, designers were forced to build multiprocessor file systems based only on speculation about how they would be used, extrapolating from characterizations of general-purpose workloads on uniprocessor and distributed file systems, or scientific workloads on vector supercomputer file systems. To fill this gap, we examined how scientific applications on two different multiprocessors use the parallel file systems available to them.

The results of our analyses suggest that the workload for which most multiprocessor file systems were optimized is very different than the workloads they are actually being asked to support. For example, it was generally assumed that scientific applications designed to run on a multiprocessor would behave in the same fashion as scientific applications designed to run on sequential and vector supercomputers: accessing large files in large, consecutive chunks [Pie89, PFDJ89, LIN+93, MK91]. Instead, our observations show that many scientific applications make many small, regular, but non-consecutive requests to the file system.

Using the results from our workload characterizations and from performance evaluations of

existing multiprocessor file systems, we have developed Galley. Galley is a new multiprocessor file system that is designed to deliver high performance to a variety of parallel, scientific applications running on multiprocessors. Rather than attempting to design a file system that is intended to directly meet the specific needs of every user, we have designed a more general system that lends itself to supporting a wide variety of libraries, each of which should be designed to meet the needs of a specific community of users.

Galley introduces a new three-dimensional means of structuring files in a parallel file system. This structure is intended to allow applications to explicitly control the way data is distributed throughout the file system, and to allow applications to explicitly control the amount and type of parallelism in use at any given time.

Galley introduces several new data-access interfaces. These interfaces allow applications to describe to the file system the more complex data access patterns we observed to be common in real parallel scientific workloads. These interfaces are designed to provide the file system with enough information to allow it to deliver higher performance, by performing better disk scheduling, making better use of the limited space available for a buffer cache, and by making more efficient use of the interconnection network. These interfaces may be considered independently of Galley's new three-dimensional file structure; they may be added to existing parallel or sequential file systems, giving applications the opportunity to achieve higher performance, but without sacrificing backwards compatibility. These interfaces have influenced the Scalable I/O Initiative's low-level application programming interface [CPD+96].

Galley's design was deliberately kept simple, to facilitate the task of developing an efficient, high-performance implementation. We will discuss our implementation, and show that we achieved our goal of efficiency and high-performance. Our implementation was designed with the goal of being easily portable to other platforms. We will show that we have succeeded in this respect as well.

We will also discuss several ways in which Galley has been used to solve problems in practice, and the ways in which Galley's features were useful in solving those problems. We will discuss in detail the implementation of one application and one user-level library that we implemented directly on top of Galley. We will also discuss in detail another application that we implemented on top of the aforementioned user-level library. We will also briefly describe several other projects

that have been implemented on top of Galley.

We will discuss several other parallel file systems and examine how Galley is similar or different to those systems. Finally, we will identify several areas that should be explored further.

1.1 Contributions

This work makes two major contributions:

- Our workload characterizations provide an empirically-based understanding of the I/O needs of parallel scientific applications. Rather than relying on guesses and extrapolations, multiprocessor file system implementors may now make design decisions based on the observed behavior of real applications being used in a production environment. Among the particular issues we examined are:
  - What did the job mix look like? How many jobs ran concurrently? How many processors did each job use? How many files did each job use?
  - Were files read, written, or both?
  - What was the distribution of file sizes?
  - What was the distribution of read and write request sizes?
  - How were requests spaced in the file? Were the accesses sequential and, if so, in what way?
  - How much inter-processor file sharing was there? How much inter-job sharing?
  - What forms of locality were there? How might caching be useful?
- We have also designed a new multiprocessor file system that is intended to meet the needs of parallel scientific applications more effectively than current existing parallel file systems. This end is accomplished in three ways:
  - A new way of structuring files in a parallel file system, which allows applications and libraries to add structure to their files and to explicitly control parallelism in file access.
  - Several new file system interfaces, with more expressive power than traditional interfaces, which provide the file system with the information it needs to efficiently handle a variety of access sizes and patterns.
  - A design that is simple and scalable enough to allow an implementation to run well on multiprocessors with dozens or hundreds of nodes.

1.2 Outline

In Chapter 2 we explore both the history and the current state of the art of file systems, both sequential and parallel, to provide sufficient context for the remainder of this thesis. In Chapter 3, we describe the implementation and the low-level results of the workload characterization phase of this work. In Chapter 4, we discuss a more detailed analysis of the file-access patterns observed in our workload characterization.

Chapter 5 discusses the high-level goals of the Galley Parallel File System, as well as the design and implementation of the system. In Chapter 6, we examine the performance and scalability of Galley. In Chapter 7 we show how Galley has been used in practice.

Chapter 8 describes related work in parallel file systems. In Chapter 9, we summarize our results and observations, and draw some overall conclusions.

Chapter 2

Background

To provide a sufficient context for describing our new work, we discuss some of the basic characteristics of file systems in general, and parallel file systems in particular. We will discuss the specific characteristics of many parallel file systems in Chapter 8.

The role of the file system is to provide a simple means for applications to store and retrieve data. The file system relieves the application of the responsibility for low-level management of the storage medium, and ensures that multiple applications do not interfere with one another.

A file system has several different responsibilities. First, it must maintain a mapping between file names and the files they represent. Second, it must keep track of which blocks on the disk are associated with each file and which blocks are available to be used. Finally, the file system must handle requests from applications to move data between the disk and the applications' address spaces.

Although sequential and parallel file systems perform the same basic tasks, they vary in the manner in which those tasks are executed.

2.1 Sequential File Systems

Since most modern file systems, both sequential and parallel, define themselves in relation to Unix, we will briefly discuss some of the characteristics of the Unix file system and its implementations. Under Unix, each file is represented as a linear, addressable stream of bytes [Tho78, RT78]. In other words, files are collections of bytes, arranged in a one-dimensional structure. Each byte within the file may be addressed using a single integer representing that byte's offset, or distance from the

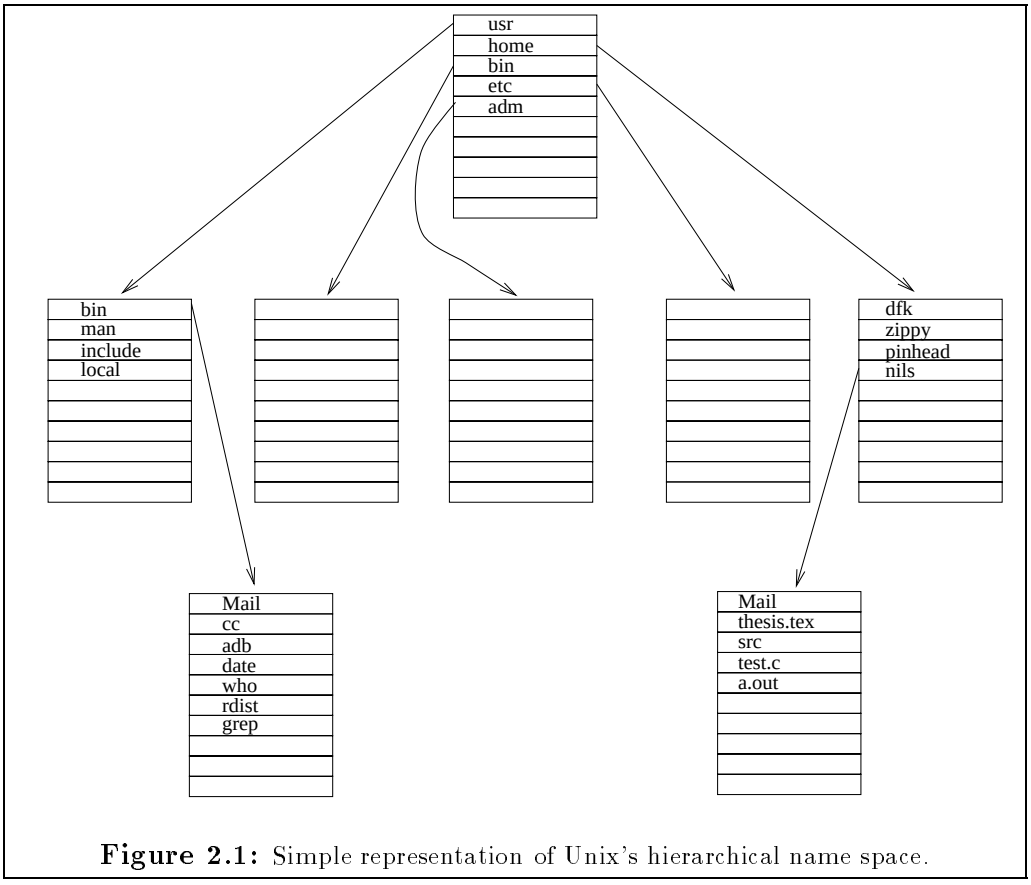

Figure 2.1: Simple representation of Unix's hierarchical name space.

beginning of the file.

2.1.1 Naming

The file namespace in Unix is structured as a tree, such as the one shown in Figure 2.1. Each node in the tree is either a file or a directory. Files contain data and directories contain files and other directories.

On the disk, each directory is stored as one or more *directory blocks*. A directory block contains a series of entries, one for each file in the directory. Each entry contains the name of a file or a directory, as well as the address of the *inode* associated with that file or directory. A file's or directory's inode contains several pieces of metadata: the time the file or directory was created, who created it, who has permission to access it, and so on. For a file, the inode contains additional information, such as the size of the file. Most importantly, the inode also tells us where to find the file's data on disk.

When an application wants to access the data stored in a file, it passes the name to the file system, which then looks up the file's inode. To find the correct inode, the file system begins by searching at the root of the file system, which is always stored at a known location on disk. It then traverses each node in the directory tree, until it reaches the directory or file with the given name. Once it finds the name of the file, it can retrieve the associated inode and can then access the file's data on behalf of the application.

2.1.2 Block Mapping

Files in Unix are mapped to disk blocks using a multi-level mapping scheme, shown in Figure 2.2. Most Unix systems use three levels of mapping: direct, indirect, and doubly-indirect. A direct mapping is a list of blocks containing actual file data. This direct mapping is generally kept right in the inode. In the original Unix file system, the inode contained direct mapping entries for the first 10 blocks of the file [Tho78, RT78]. Since each block in that system was 512 bytes, files that were 5120 bytes or less were mapped completely by the direct entries in the inode. The Berkeley Fast File System (FFS), a higher-performance Unix file system released with BSD 4.2, used a block size of at least 4096 bytes, and the inode typically contained 5 to 13 direct entries [MJLF84]. Thus, FFS is able to access files at least 20 KB in size using only the direct entries in the inode.

If the file is too large to be mapped by the direct entries in the inode, the file system uses an indirect block. The disk address of this indirect block is also stored in the inode. Rather than containing file data, the indirect block contains a list of mapping entries. If the inode contains the mappings for the first 10 blocks in the file, then the first entry in the indirect block is a mapping for the eleventh block of the file. Using 4 bytes for each disk address, the indirect block in the original Unix file system was able to map an additional 128 blocks, or 64 KB. With its 4 KB minimum block size, the FFS is able to map at least an additional 4 MB using an indirect block.

If the file is too large to be mapped using the indirect block, Unix file systems implement a doubly-indirect block, the address of which is also stored in the file's inode. Rather than mapping file blocks to disk blocks, the doubly-indirect block contains pointers to indirect blocks, similar to those described above. Thus, a doubly-indirect block contains pointers to 128 indirect blocks in the original Unix file system, and to at least 1024 indirect blocks in the Fast File System.

The original Unix file system also supports a triply-indirect block, for those files that are larger

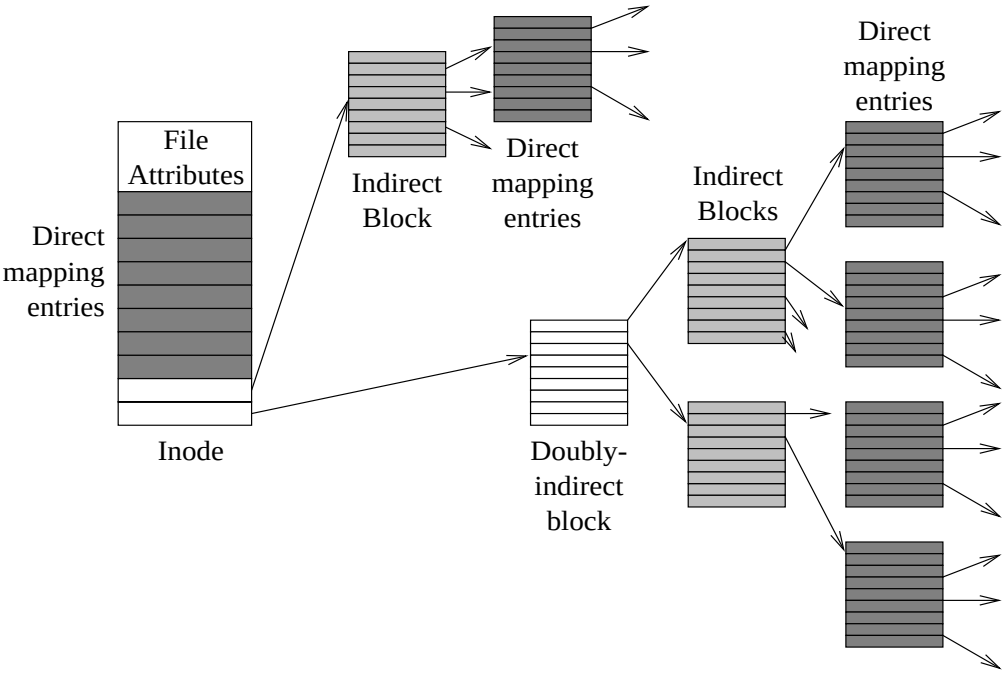

Figure 2.2: Unix's multi-level scheme for mapping file blocks to disk blocks.

than about 8.5 MB. Since the FFS requires at least a 4 KB block size, only a doubly-indirect block is necessary to map files up to 2 GB in size. Until recently, files in Unix could not be larger than 2 GB, so there was no need for FFS to support a triply-indirect block.

Finally, the file system must keep track of which blocks on a disk are in use and which are still available. The first Unix file system maintained a linked list of the free blocks on the disk. This scheme makes it possible to find a free block in constant time, as long as you are not picky about which block you get. However, maintaining the integrity of a linked list on disk in the presence of potential system failures can be a tricky process. Rather than a linked list, the FFS uses a series of bitmaps, with one bit for each block in the system. This structure takes little space and is easy to maintain, but searching for a free block can take longer than constant time. Since the FFS also used a more complex block allocation scheme (e.g., attempting to keep the blocks of a file within the same cylinder group, minimizing rotational latency, etc.), a linked list would also take more than constant time to return a new block that fit the restrictions imposed by the file system.

2.1.3 Data Access

Unix provides applications with a simple data-access interface. Files are linear streams of bytes, and Unix provides the calls `read()` or `write()`, which allow applications to access contiguous regions of those linear streams. The standard Unix file system interface provides only *blocking* semantics. That is, when an application issues a `read()` or `write()` request, the application stops and waits until the call is completed. For a `read()` request, the call is completed when all the requested data has been placed in the application's buffer. For a `write()` request, the call is completed when all the data has been copied out of the application's buffer, and into one of the operating system's internal buffers. Some variants of Unix also provide *non-blocking* I/O calls. These routines return control to the application immediately, and the actual data transfer is carried on behind the scenes as the application continues executing. Non-blocking I/O can allow an application to overlap computation and I/O, reducing overall execution time. Non-blocking I/O tends to be significantly more complicated to use, and the performance varies widely from system to system.

Rather than requiring the application to explicitly specify which region it wants to access on every request, Unix maintains a *file pointer* for each file an application has open. This file pointer indicates the byte following the last byte accessed (before the application first accesses the file, the pointer points to byte 0). On each `read()` or `write()`, the application identifies the number of bytes to access, and the buffer from or to which data should be transferred. The file system then reads or writes the specified number of bytes from the file, beginning at the location indicated by the file pointer. When the access is completed, the file system updates the pointer to point to the byte following the last byte accessed. This interface encourages sequential access, and thus most Unix file systems are optimized for sequential access. The application may tell the file system to move the file pointer to any arbitrary location within the file, without transferring any data, using the `lseek()` call.

2.2 Parallel File Systems

As the sizes of interesting and tractable problems has grown, the amounts of data required to solve these problems has grown as well. An individual disk cannot store enough information, nor can it access it rapidly enough, to solve these problems. Just as we use multiple processors in parallel to

increase computational power, so we can use multiple disks in parallel to increase I/O power.

There are several issues to consider when designing a parallel I/O system for a multiprocessor. First, where in the system are the disks placed? Second, how is the file data distributed among the multiple disks? Finally, how do applications access that data? We will briefly discuss some common ways in which parallel file systems have addressed these issues.

2.2.1 Structure

The first issue to be considered when designing a parallel I/O system is where to place the disks.

One of the most common types of parallel I/O system is the RAID (Redundant Array of Inexpensive Disks) system [PGK88]. In a RAID system multiple disks are clustered together, and the system can deliver high performance by exploiting the high aggregate bandwidth of the multiple disks. A RAID is usually connected to a computer system by a high-speed channel, such as SCSI-2, Fiber Channel, or HiPPI. When the computer accesses the RAID through this channel, the RAID controller hides the parallelism of the underlying system, making the RAID appear as a single high-speed disk, allowing file systems to ignore the complexity of handling multiple disks at a time. Even with a high-speed connection, however, the single channel can become a significant bottleneck for the whole I/O subsystem.

One way to avoid this bottleneck is to eliminate the single channel. In a sequential computer system, this elimination may be accomplished by installing multiple SCSI interfaces, and attaching only a small number of disks to each SCSI bus. In a parallel computer system, this approach can be taken one step further; we can attach disks to entirely different nodes within the system. The disadvantage of distributing the disks among different nodes is that we lose the illusion of a single fast disk, and the file system must explicitly manage the multiple disks. How this management should be done is the primary focus of our research.

Rather than asking each node in a parallel machine to function as both a compute server and an I/O server, most parallel file systems are designed along a client-server model. Some subset of the nodes in the system are designated *I/O processors* (IOPs), and the remainder are designated *compute processors* (CPs). In general, systems that are based on this sort of dichotomy adhere to it strictly. That is, the I/O nodes are used exclusively as I/O servers and the compute nodes are used exclusively for running users' applications. Examples of systems based on this architecture

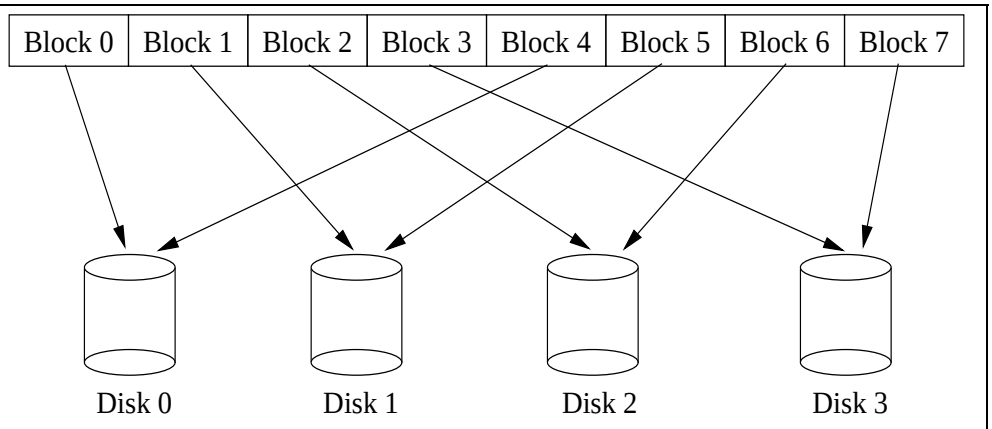

Figure 2.3: Example of striping. The file shown has 8 blocks, which are striped across 4 disks.

are Intel's iPSC/860 and Paragon, and Thinking Machines' CM-5.

2.2.2 Data Distribution

Once the basic architecture of a parallel file system has been determined, the next major issue is the distribution of data within the system. The approach adopted by most parallel file systems is to distribute a file's data across all the disks in the system. This practice is typically called *striping* or *declustering.* This striping is performed by breaking the file into smaller units, typically measured in kilobytes, and assigning these units to disks in round-robin order. A simple example of such a declustering may be seen in Figure 2.3. Kim [Kim86] and Salem [SGM86] were among the earliest researchers to demonstrate the usefulness of transparent striping.

This approach is simple to implement, and it is easy to identify on which disk a given file block is stored, without requiring an expensive lookup operation. Also, for large requests, this distribution can lead to good load balancing among the disks.

2.2.3 Interfaces

Most parallel file systems present applications with an interface similar to that provided by Unix. As discussed above, using a Unix-style interface, applications may access any region of a file, but they may only access data in contiguous chunks.

To support parallel applications, most parallel file systems also provide some facility for describing how the data within the linear file is to be shared among the multiple compute nodes in

the application. In file systems with a linear file model, the most common such facility is the *shared file pointer*. In other words, each file is assigned a single file pointer, which is shared by all the nodes in an application.

File systems that provide shared file pointers also tend to allow applications to choose from a number of *modes*, each of which provides different semantics governing how the file pointer is shared by the nodes in the application. For example, Intel's Concurrent File System provides 4 modes. In Mode 0, the default mode, each node has its own private file pointer. Mode 1 provides atomic-append semantics; there is a single, shared file pointer, and only one node may access the file at a time. Access is granted in a first-come first-serve fashion. Mode 2 again provides a shared file pointer, but access to the file is granted to the compute nodes in round-robin order. Finally, Mode 3 is similar to Mode 2, with the added restriction that each node access the same amount of data on each turn.

Several other types of parallel-application support are discussed in Chapter 8.

2.3 File System Workloads

Although researchers have studied file-system workloads on general-purpose workstations and vector supercomputers, there has never been an extensive study of multiprocessor file-system workloads.

General-purpose uniprocessor workloads.

Uniprocessor file-access patterns have been measured many times. Floyd and Ellis [Flo86, FE89] and Ousterhout et al. [OCH+85] measured isolated Unix machines. Baker et al. studied the workload of a cluster of workstations running the Sprite file system, which is a distributed, Unix-like system [BHK+91]. Ramakrishnan et al. studied access patterns in a commercial computing environment on a VAX/VMS platform [RBK92]. These studies all cover general-purpose (engineering and office) workloads with uniprocessor applications. These studies identify several characteristics that are common among uniprocessor file-system workloads: files tend to be small (only a few kilobytes), they tend to be accessed with small requests, and they tend to be accessed both completely and sequentially (i.e., each byte in the file is accessed in order — from beginning to end).

Scientific vector applications.

Some studies examined scientific workloads on vector machines. In [dC94], del Rosario and Choudhary provide an informal characterization of grand-challenge applications. Powell measured a set of static characteristics (file sizes) of a Cray-1 file system [Pow77]. Miller and Katz traced specific I/O-intensive Cray applications to determine the per-file access patterns [MK91], focusing primarily on access rates. Miller and Katz also measured secondary-tertiary file migration patterns on a Cray [MK93], giving a good picture of long-term, whole-file access patterns. Pasquale and Polyzos studied I/O-intensive Cray applications, focusing on patterns in the I/O rate [PP93, PP94a, PP94b]. All of these studies are limited to single-process applications on vector supercomputers. These studies identify several characteristics that are common among supercomputer file-system workloads. Unlike workstation file-system workloads, files tend to be large (many megabytes or gigabytes) and they tend to be accessed with large requests (megabytes at a time). Like workstation workloads, files are typically accessed both completely and sequentially.

Scientific parallel applications.

Although there has been no other study of the full workload of a multiprocessor file system, people have studied individual applications. Reddy et al. chose five sequential scientific applications from the PERFECT benchmarks and parallelized them for an eight-processor Alliant, finding only sequential file-access patterns [RB90]. This study is interesting, but far from what we need: the sample size is small; the programs are parallelized sequential programs, not parallel programs *per se*; and the I/O itself was not parallelized. Cypher et al. [CHKM93] studied individual parallel scientific applications, measuring temporal patterns in I/O rates. Crandall et al. instrumented three parallel applications to study their file-access activity in detail [CACR95]. Although they found primarily sequential access patterns, the patterns were often cyclical (e.g., applications repeatedly opened and closed the same file, each time accessing it in the same pattern). There was a wide distribution in request sizes, though few were larger than 1 MB, and a wide variation in spatial and temporal access patterns. Baylor et al. performed a similar analysis of several applications on the IBM SP-2 [BW96]. Galbreath et al. present a high-level characterization of multiprocessor file-system workloads based on anecdotal evidence [GGL93]. Crockett [Cro89] hypothesizes about the character of a parallel scientific file-system workload.

In the next two chapters we present our own study of the workloads on two parallel machines in use in a production environment.

Chapter 3

Workload Characterization

Perhaps the most important information needed by the designer of a complex system is a reasonable approximation of the workload that the system will be expected to support. One way of arriving at such an approximation is to examine the workloads supported by existing systems.

Ideally, a workload characterization is an architecture-independent representation of the work generated by a group of users in a particular type of computing environment. However, since the architectures of different parallel I/O subsystems are so diverse, any observed workload will be tied to a particular machine. Although we try to factor out these effects as much as possible in our discussion below, we must note that some care should be taken in generalizing the results.

3.1 Platform

To be useful to a system designer, a workload study must be performed in an environment similar to that in which the new system is expected to be installed. For our purposes, this meant that we had to trace the activity of a multiprocessor file system that was in use for production scientific computing. The Intel iPSC/860 at NASA Ames' Numerical Aerodynamics Simulation (NAS) facility met this criterion.

The iPSC/860 is a distributed-memory, message-passing, MIMD machine. The machine has 128 compute nodes, based on the Intel i860 processor, that are connected by a hypercube network. I/O is handled by 10 dedicated I/O nodes, each of which is connected to one of the compute nodes rather than directly to the hypercube interconnect. The I/O nodes are based on the Intel i386 processor, and each has a single bus for SCSI disk drives. There may also be one or more service

nodes that handle Ethernet connections or interactive shells [NAS93].

Intel's Concurrent File System (CFS) [Pie89, FPD93, Nit92] provides a Unix-like interface to the user with the addition of four *I/O modes*, as discussed in the previous chapter. CFS generally stripes each file across all I/O nodes in 4 KB blocks. CFS allows users to specify that a file only be stored on a subset of the available I/O nodes, but we found no application that use that functionality. In some parallel file systems compute nodes send requests to a server, which is responsible for determining which I/O nodes will be involved in satisfying the request. Under CFS, compute nodes send requests directly to the appropriate I/O node, without the involvement of such a server. Each I/O node maintains a cache of recently used data from its local disk. No caching is done on the compute nodes.

3.2 Data Collection

On the iPSC/860, high-level CFS calls are implemented in a library that is linked with the user's application. We instrumented the library calls to generate an event record each time they were called. The event records were buffered at each compute node and periodically sent to a data collector running on the service node. The collector then wrote the data to the central trace data file, itself on CFS. To avoid skewing the results of our study, the collector's use of CFS was not recorded in the trace data.

For our study, one data file was collected for the entire file system. One other possibility, used in the study discussed in Section 3.5, is to store the trace information for each job in its own file. The trace data files begin with a header record containing enough information to make the file self-descriptive, and continue with a series of event records, one per event.

This work was conducted as part of the CHARISMA project, which we began in June 1993 to CHARacterize I/O in Scientific Multiprocessor Applications from a variety of production parallel computing platforms and sites. The CHARISMA project is unique in recording individual read and write requests in live, multiprogramming, parallel workloads, rather than from selected or non-parallel applications. Since one of the goals of the CHARISMA project was to organize and facilitate a multi-platform file-system tracing effort, we defined a large set of event records suitable for both SIMD and MIMD systems. The records used in the iPSC/860 study are shown in Figure 3.1.

We traced only the I/O that involved the Concurrent File System. This means that any I/O

Notes:

UserID — Unix UID
SystemID — Internet IP address
FileID — (disk, block number) of File Header Block
ClientID — number of node requesting I/O

Header:
- Magic number
- Format version number
- Start date (standard Unix date format)
- System type (iPSC, CM-5, etc.)
- SystemID
- System Configuration (procs, disks, memory)
- Timestamp unit (in seconds, 64-bit float)

Job load:
- record type code
- timestamp
- program name
- path to executable
- UserID
- list of ClientIDs (nodes running the job)

Client completion:
- record type code
- timestamp
- ClientID

Client Open file:
- record type code
- timestamp
- ClientID
- FileID
- file descriptor
- file name
- file size
- file creation time
- open mode (r, w, rw, create, etc.)

Client Close file:
- record type code
- timestamp
- ClientID
- file descriptor
- file size

Read/Write request:
- record type code
- operation type (r/w, sync/async, etc.)
- timestamp
- ClientID
- file descriptor
- file offset
- size of I/O

Truncate/Extend:
(explicit operations only)
- record type code
- timestamp
- ClientID
- file descriptor
- original file size
- new file size

Link/Unlink:
- record type code
- timestamp
- ClientID
- FileID
- new number of links to file

Set I/O mode:
- record type code
- timestamp
- ClientID
- file descriptor
- new access mode

Figure 3.1: The event records used when tracing the file system workload on an iPSC/860.

which was done through standard input and output or to the host file system (all limited to sequential, Ethernet speeds) was not recorded. We collected data for about 156 hours over a period of 3 weeks. Although we did not trace continuously for the whole 3 weeks, we tried to get a realistic picture of the whole workload by tracing at all different times of the day and of the week, including nights and weekends. The period covered by a single trace file ranges from 30 minutes to 22 hours. The longest continuously traced period was about 62.5 hours. Tracing was usually initiated when the machine was idle. For those few cases in which a job was running when we began tracing, the job was not traced. Tracing was stopped in one of two ways: manually or by a system crash. The machine was usually idle when tracing was manually stopped.

Since our instrumentation was almost entirely within a user-level library, there were some jobs whose file accesses were not traced. These included most system programs (e.g., `ls`, `cp`, and `ftp`) as well as user programs that were not relinked during the period we were tracing. Although we were able to record all job starts and ends through a separate mechanism, there was no way to distinguish between a job which was untraced from a job which simply did no CFS I/O, so we do not know precisely how many jobs were traced. While we were tracing, 3016 jobs were run on the compute nodes, of which 2237 were only run on a single node. We actually traced at least 429 of the 779 multi-node jobs and at least 41 of the single-node jobs. As a tremendous number of the single-node jobs were system programs it is not surprising nor necessarily undesirable that so many were untraced. In particular, there was one single-node job that was run periodically, simply to check the status of the machine. That one job accounted for over 800 of the single-node jobs.

One of our primary concerns was to minimize the degree that our measurement perturbed the workload. We identified three ways that our instrumentation might affect the workload.

Our first concern was network contention. We expected users' jobs to generate a great many event records. Had we chosen to send a message to the data collector for each event record, we would certainly have created unreasonable congestion near the collector or perhaps in the overall machine. Since large messages on the iPSC are broken into 4 KB blocks, we chose to create a buffer of that size on each node to hold local event records. This buffer allowed us to reduce the number of messages sent by our instrumentation by over 90% without stealing much memory from user jobs.

The second concern was local CFS overhead. Since we were tracing every I/O operation in a

production environment, it was imperative that the per-call overhead be kept to a minimum to avoid inconveniencing the users. By buffering records on the compute nodes we were able to avoid the cost of message passing on every call to CFS.

Our final concern was that we might increase contention for the I/O subsystem. We tried to minimize this effect by creating a large buffer for the data collector and writing the data to CFS in large sequential blocks. Although we collected about 700 MB of data, our trace files accounted for less than 1% of the total traffic.

Simple benchmarking of the instrumented library revealed that the overhead added by our instrumentation was virtually undetectable in many cases. The worst case we found was a 7% increase in execution time on one run of the NAS NHT-1 Application-I/O Benchmark [CCFN92]. After the instrumented library was put into production use, anecdotal evidence suggests that there was no noticeable performance loss.

3.2.1 Postprocessing

The raw trace files required some simple postprocessing before they could be easily analyzed. This postprocessing included data realignment, clock synchronization, and chronological sorting.

Since each node buffered 4 KB of data before sending it to the central data collector, the raw trace file contained only a partially ordered list of event records. Ordering the records was complicated by the lack of synchronized clocks on the iPSC/860. Each node maintains its own clock; the clocks are synchronized at system startup, but each drifts significantly and differently after that. We partially compensated for the asynchrony by time-stamping each block of records when it left the node and again when it was received at the data collector. From the difference between the two we could approximately adjust the event order to compensate for each node's clock drift relative to the collector's clock. This technique allowed us to get a closer approximation of the event order. Nonetheless, it is still an approximation, so much of our analysis is based on spatial, rather than temporal, information.

3.3 Results

We characterize the workload from the top down, beginning with the number of jobs in the machine and the number and use of files by all jobs. We then examine individual I/O requests by looking

for sequentiality, regularity, and sharing in the access pattern. Finally, we evaluate the effectiveness of caching through trace-driven simulation.

3.3.1 Jobs

As a first look into the details behind Table 3.1, Figure 3.2 shows the amount of time the machine spent running a given number of jobs. For more than a quarter of the traced period, the machine was idle (i.e., zero jobs). For about 35% of the time it was running more than one job, sometimes as many as eight. Although not all jobs use the file system, a file system clearly must provide high-performance access by many concurrent, presumably unrelated, jobs.

Trace	Traced	Megabytes		Number of files				
name	Jobs	Read	Written	Opened	Read	Written	Both	Neither
feb10	73	2977.63	1311.35	3609	2659	573	280	97
feb11	46	129.79	1161.70	5281	41	4185	803	252
feb14p1	9	334.81	395.14	1610	791	819	0	0
feb14p2	15	1701.25	1691.18	783	313	309	147	14
feb14p3	1	40.18	45.92	130	97	33	0	0
feb14p4	12	98.22	121.97	1392	165	919	292	16
feb15	34	18835.90	19265.64	4968	698	3622	442	206
feb16p1	37	12860.40	12593.27	2893	2468	406	2	17
feb16p2	30	32.66	505.09	2159	176	1709	0	274
feb17	20	517.74	398.28	3068	1447	1242	292	87
feb18p1	3	54.78	117.45	735	162	541	0	32
feb18p2	9	196.33	284.34	1248	521	567	0	160
feb18p3	12	28.30	307.49	838	128	676	0	34
feb21	6	114.83	224.47	684	198	294	0	192
feb22p1	37	325.78	386.63	3679	534	3025	1	119
feb22p2	14	16.71	228.49	3500	188	3269	0	43
feb22p3	7	21.44	79.44	2573	247	2217	0	109
feb23p1	17	96.64	3698.34	8168	688	7440	0	40
feb23p2	63	216.51	381.98	9512	1166	7680	0	666
feb24	5	142.96	1261.17	1751	702	981	0	68
mar1	30	69.54	265.96	5198	1151	3993	0	54
Totals	470	38812.40	44725.29	63779	14540	44500	2259	2480
				100.0%	22.8%	69.8%	3.5%	3.9%

Table 3.1: Summary statistics of the trace data. Only those jobs whose file accesses were caught by our library are included here.

Of course, some of the jobs in Figure 3.2 were small, single-node jobs, and some were large

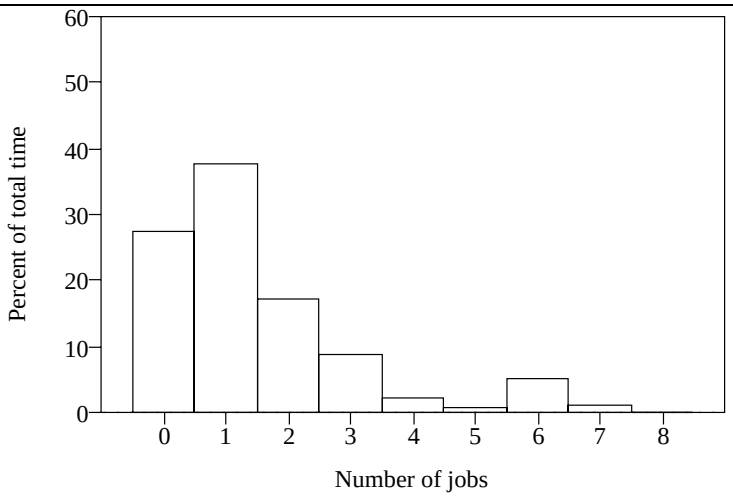

Figure 3.2: Amount of time the machine spent with the given number of jobs running. This data includes all jobs, even if their file access could not be traced.

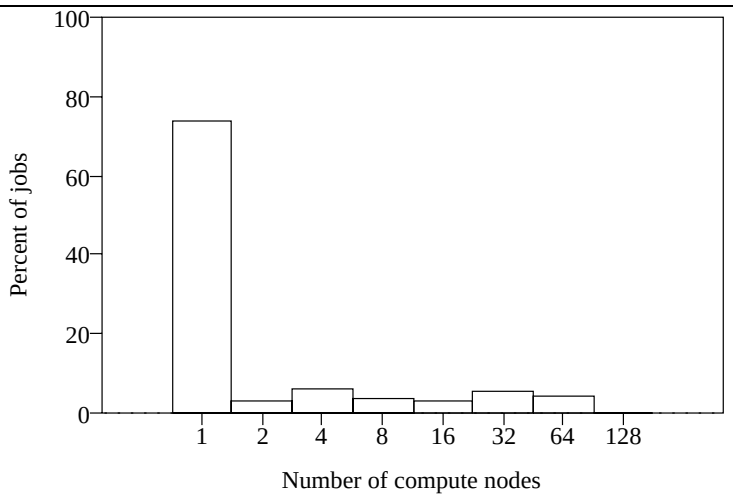

Figure 3.3: Distribution of the number of compute nodes used by jobs in our workload (even those whose file access could not be traced). The iPSC limits the choice to powers of 2.

parallel jobs. Figure 3.3 shows the distribution of the number of compute nodes used by each job. One-node jobs dominated the job population, although large parallel jobs dominated node usage. A successful file system must allow both small, sequential jobs and large, highly parallel jobs access to the same files under a variety of conditions and system loads.

3.3.2 Files

In Table 3.1 above, files are classified by how they were actually used rather than by the mode in which they were opened. Note that many more files were written than were read (indeed, more than three times as many). We found that the programmers of traced applications often found it easier to open a separate output file for each compute node, rather than coordinating writes to a common output file. This tendency may have contributed to the substantially smaller average number of bytes written per file (1.2 MB) than average bytes read per file (3.3 MB). Note also that there were very few files that were read and written in the same open. This latter behavior is common in Unix file systems [Flo86] and may be accentuated here by the difficulty in coordinating concurrent reads and writes to the same file (note that the CFS file-access modes are of little help for read-write access). We suspect that most of those files that were opened, but not accessed at all, were opened by applications that terminated prematurely.

Table 3.2 shows that most jobs opened only a few files over the course of their execution, although a few opened many files. The maximum was one job that opened 2217 files. Some of the jobs that opened a large number of files were opening one file per node. Although not all files were open concurrently, file-system designers must optimize access to several files within the same job.

Number of Files	Number of Jobs
1	71
2	15
3	24
4	120
5+	240

Table 3.2: Among traced jobs, the number of files opened by jobs was often small (1–4).

We found that only 0.61% of all opens were to "temporary" files, which we defined to be any file deleted by the same job that created it. Nearly all of those temporary files may have been from one application. The rarity of temporary files and of files that were both read and written indicates that few applications chose to use files as an extension of memory for out-of-core solutions. Many of the Ames applications are computational fluid dynamics (CFD) codes, for which they have found that out-of-core methods are in general too slow.

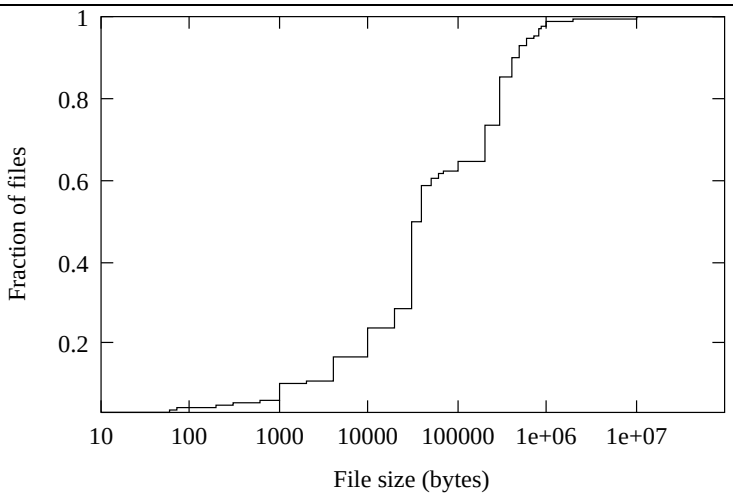

Figure 3.4: Cumulative distribution function (CDF) of the number of files of each size at close. For a file size x, CDF(x) represents the fraction of all files that had x or fewer bytes.

Figure 3.4 shows that most of the files accessed were large (10 KB to 1 MB).[1] It is important to note that each of the largest jumps is primarily due to one or two applications, so undue emphasis should not be placed on the specific numbers as opposed to the general tendency towards larger files. Although these files were larger than those in a general-purpose file system [BHK+91], they were smaller than we would expect to see in a scientific supercomputing environment [MK91]. We suspect that users limited their file sizes due to the small disk capacity (7.2 GB) and limited disk bandwidth (10 MB/s peak).

3.3.3 I/O Request Sizes

Figures 3.5 and 3.6 show that the vast majority of reads are small, but that most bytes are transferred through large reads.

Indeed, 96% of all reads were for fewer than 100 bytes, but those reads transferred only about 2% of all data read. Similarly, 89% of all writes were for fewer than 100 bytes, but those writes transferred only about 2.5% of all data written. The number of small requests is surprising due to their poor performance in CFS [Nit92]. The jump at 4 KB indicates that some users have optimized

[1] As there was a large number of small files as well as a number of distinct peaks across the whole range of sizes, there was no constant granularity that captured the detail we felt was important in a histogram. We chose to plot the file sizes on a logarithmic scale with pseudo-logarithmic bucket sizes: the bucket size between 10 and 100 bytes is 10 bytes, the bucket size between 100 and 1000 is 100 bytes, and so on.

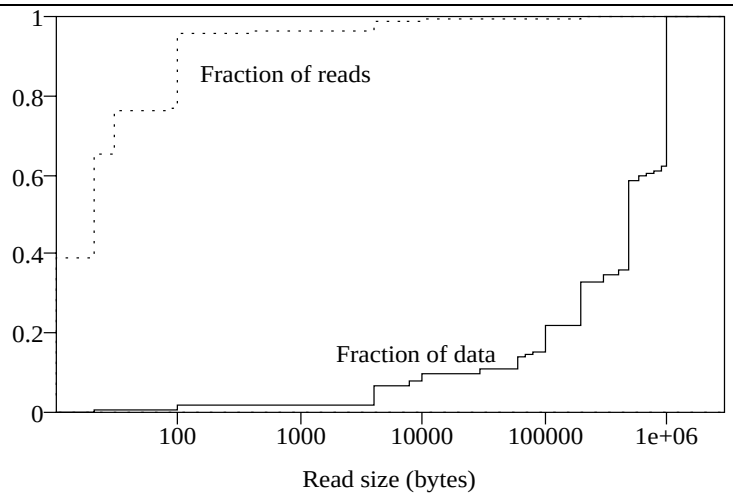

Figure 3.5: CDF of the number of reads by request size and of the amount of data transferred by request size.

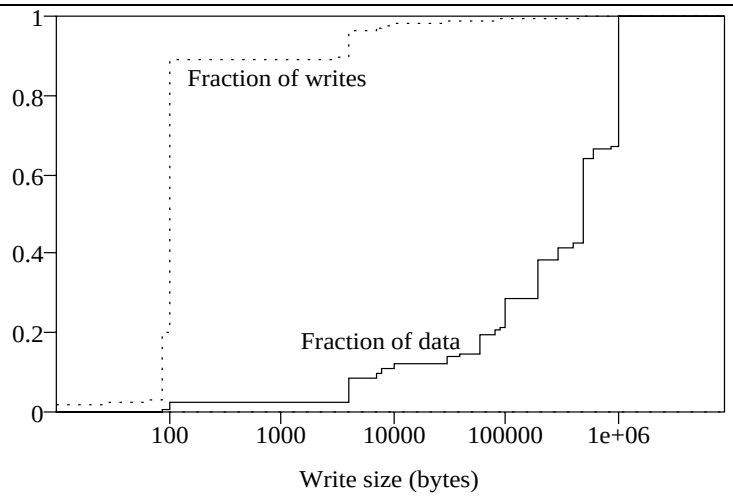

Figure 3.6: CDF of the number of writes by request size and of the amount of data transferred by request size.

for the file-system block size, but it appears that most users preferred ease of programming over performance.

Figures 3.5 and 3.6 show spikes in the number of small requests as well as in the data transferred by 1 MB requests. Although the spikes of small requests occurred throughout the tracing period, one trace alone (probably one application alone) contributed the large spike at 1 MB. Although the specific position of the spikes is likely due to the effect of individual applications, we believe

that the preponderance of small request sizes is the natural result of parallelization by distributing file data across many processors, and would be found in other workloads using a similar file-system interface.

3.3.4 Sequentiality

We define a *sequential* request to be one that begins at a higher file offset than the previous request from the same compute node. We define a *consecutive* request to be a sequential request that begins precisely one byte beyond where the previous request ended. A common characteristic of file workloads, particularly scientific workloads, is that files are accessed consecutively [OCH+85, BHK+91, MK91]. Frequently, files are also accessed in their entirety. Figures 3.7 and 3.8 show the amount of sequential and consecutive access (on a per-node basis) to files with more than one request in our workload.

The spikes at 100% in the two figures show that most write-only files were accessed strictly consecutively (and hence strictly sequentially as well). This behavior was likely due to the fact that most write-only files were written only by one processor. Most read-only files were also accessed strictly sequentially. Unlike the write-only files, however, nearly 70% of read-only files were accessed with non-consecutive requests, as illustrated by the spike at 0% in Figure 3.8. These sequential, but non-consecutive, patterns were the result of interleaved access, where successive records of the file are accessed by different nodes. In an interleaved access pattern, from the perspective of an individual node, some bytes must be skipped between one request and the next. Not surprisingly, access to read-write files was primarily non-sequential.

3.3.5 I/O-Request Intervals

We define the number of bytes skipped between one request and the next to be the *interval size*. Consecutive accesses have interval size 0. The number of *different* interval sizes used in each file, across all nodes that access that file, is shown in Table 3.3. A surprising number of files were read or written in one request per node, so there were no intervals. Over 99% of the 1-interval-size files were consecutive accesses; the one interval size was 0. The remainder of 1-interval-size files, along with the 2-interval-size files, represent 5% of all files, and indicate another form of highly regular access pattern. Only 1.2% of all files had 3 or more different interval sizes, and their regularity, if

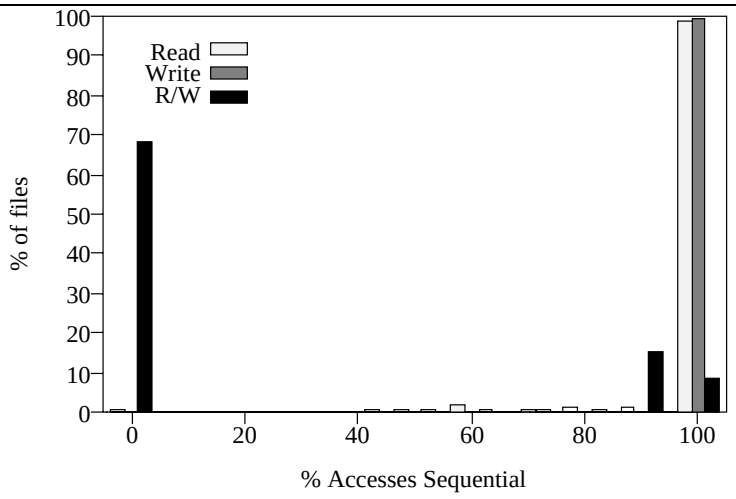

Figure 3.7: Sequential access to files on a per-node basis. Most read-only and write-only files were accessed strictly sequentially.

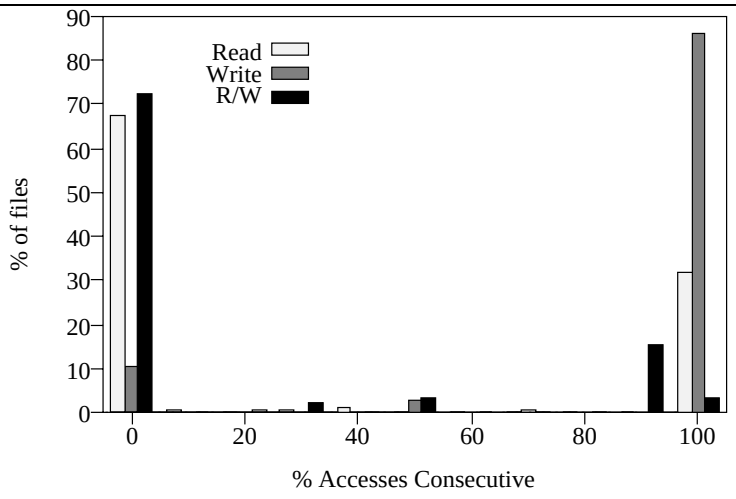

Figure 3.8: Consecutive access to files on a per-node basis. Most write-only files were accessed strictly consecutively.

any, was more complex.

Number of different interval sizes	Number of files	Percent of total files
0	20078	38.2
1	29451	56.1
2	2363	4.5
3	48	0.1
4+	596	1.1

Table 3.3: The number of different interval sizes used in each file across all participating nodes. Zero represents those cases where only one access was made to a file, per node.

To get a better feel for this regularity, we also counted the number of different *request sizes* used in each file, as shown in Table 3.4. Over 90% of the files were accessed with only one or two request sizes. Combining the regularity of request sizes with the regularity of interval sizes, many applications clearly used regular, structured access patterns. These patterns are explored in more detail in the next chapter.

Number of different request sizes	Number of files	Percent of total files
0	2480	3.9
1	25523	40.0
2	32779	51.4
3	2510	3.9
4+	487	0.8

Table 3.4: The number of different request sizes used in each file across all compute nodes. Files with zero different sizes were opened and closed without being accessed.

3.3.6 Synchronization

Given the regular request sizes and interval sizes shown in Tables 3.3 and 3.4, Intel's I/O modes (discussed in Chapter 2) would seem to be helpful. Our traces show, however, that over 99% of the files were accessed using mode 0, so fewer than 1% of the files were accessed using modes 1, 2, or 3. Tables 3.3 and 3.4 may give one hint as to why: although there were few different request sizes and interval sizes, there were often more than one, something not easily supported by the automatic file modes. Anecdotal evidence also suggests that programmers chose not to use modes 1 to 3 because

they were significantly slower than mode 0.

3.3.7 Sharing

We call a file *shared* if more than one process opens it. It is *concurrently shared* if the opens overlap in time. It is *write-shared* if at least one of the processors writes to the file. In uniprocessor and distributed-system workloads, concurrent sharing is known to be rare [BHK+91]. In a parallel file system, concurrent file sharing among processes within a job is presumably the norm, while concurrent file sharing between jobs is likely to be rare. Indeed, in our traces we saw a great deal of file sharing within jobs, and *no* concurrent file sharing between jobs. The interesting question is *how* the individual bytes and blocks of the files were shared. A block, in this case, refers to one of CFS's 4 KB file blocks.

Figure 3.9 shows the percentage of files with varying amounts of byte- and block-sharing. Since it is impossible for files opened by only a single node to have any amount of sharing, this figure includes only those files that were concurrently opened by multiple nodes. There was more sharing for read-only files than for write-only or read-write files, which is not surprising given the complexity of coordinating write sharing. Indeed, although 70% of read-only files had 100% of their bytes shared, 90% of write-only files had no bytes shared at all. While a half of all read-write files (not shown in Figure 3.9) were 100% byte-shared, 93% of them were 100% block-shared, which would stress a cache consistency protocol, if present. Overall, the amount of block sharing implies strong *interprocess* spatial locality, and suggests that caching may be successful.

3.4 Caching

Buffering and caching are common in traditional file systems, and with the right policies can be successful in multiprocessor file systems [KE93b, KE93a]. One advantage of caching is that multiple small requests (which were common in this workload) may be combined into a few larger requests that can be more efficiently served by disk hardware. Indeed, with RAID disk arrays commonly seen on today's multiprocessors (such as the Intel Paragon and the KSR-2) it is even more important to avoid small requests at the disk level. Fortunately, the small requests seen in Figures 3.5 and 3.6, when coupled with small interval size, lead to spatial locality. Other potential benefits may come from temporal or interprocess locality in the access pattern.

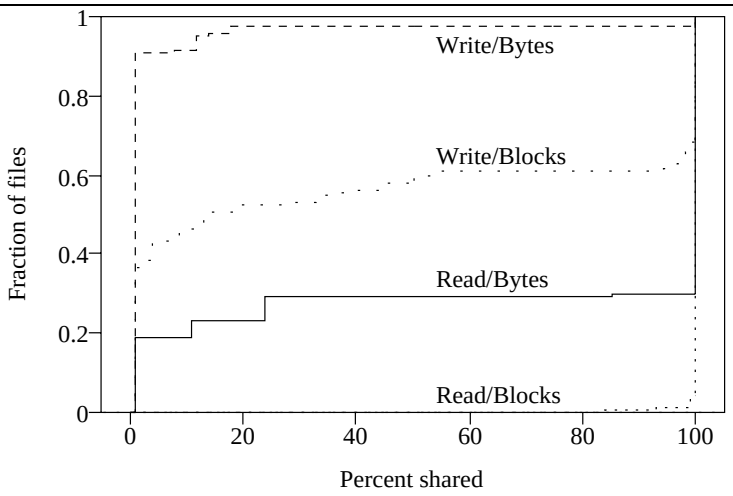

Figure 3.9: CDF of file sharing between nodes in Read-Only and Write-Only files at byte and block granularity.

In a distributed-memory machine, it is possible to place a buffer cache at the compute nodes, at the I/O nodes, or both. We evaluated all three options with trace-driven simulation.

3.4.1 Compute-node Caching

The amount of block sharing in write-only and read-write files show that any attempt to maintain write-buffers at the compute nodes would necessitate a cache consistency protocol, so we restricted our effort to read-only files. Other people have examined the possibility of caching data for write-only files at the compute node [PEK96]. The results of a simple trace-driven simulation of read-only compute-node caching, with 4 KB (one block) buffers and LRU replacement, are shown in Figure 3.10. We consider a *hit* to be any request that was *fully* satisfied from the local buffer (i.e., with no request sent to an I/O node).

Caching success, as indicated by a high hit rate, was limited to a subset of the jobs: 40% of the jobs had a greater than 75% hit rate, but 30% of the jobs had a 0% hit rate. Further, for those jobs where a cache was beneficial, a single one-block buffer per compute node was usually sufficient. A single buffer could maintain a high hit rate in patterns with a small request size and a short (perhaps zero) interval size. Clearly there was spatial locality in our workload, and not much temporal locality, or multiple buffers would have helped more[2]. In short, it appears that a one-

[2]Multiple buffers were useful in a very few jobs, apparently those which were interspersing reads from more than

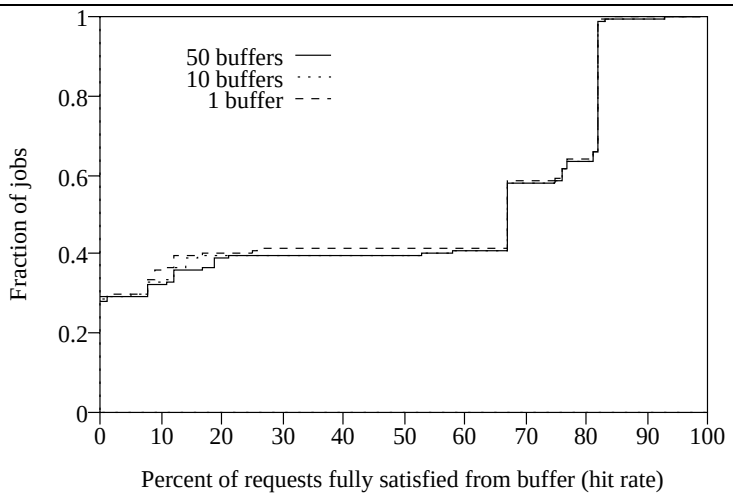

Figure 3.10: Results of compute-node caching simulation. Hit rates differed from job to job, with three distinct clumps, indicating that the cache either helped or did not. One buffer was as good as many buffers.

block buffer per compute node, per file, may be useful for read-only files, but a careful performance analysis is still necessary.

3.4.2 I/O-node Caching

Given the apparent interprocess locality, I/O-node caching should also be successful. To find out whether that was the case, we ran a trace-driven simulation of I/O-node caches, with 4-KB buffers managed by either an LRU or FIFO replacement policy. The number of compute nodes was held constant at 128 (the actual number of compute nodes in the traced system), and the number of I/O nodes varied from 1 to 20. The actual number of I/O nodes on the traced system was 10. These I/O-node caches served all compute nodes, all files, and all jobs, according to our best guess of the event ordering within our traces as described in Section 3.2. We assumed that the files were striped in a round-robin fashion at a one-block granularity. No compute-node cache was used.

Figure 3.11 shows the results of the simulation. With LRU replacement, a small cache (4000 4-KB buffers over all I/O nodes) was sufficient to reach a 90% hit rate. With FIFO replacement, nearly 20000 buffers were needed to obtain a 90% hit rate, since FIFO does not give preference to blocks with high locality. It made little difference whether the buffers were focused on a few I/O

one file. In those cases a single buffer *per file* would have been appropriate.

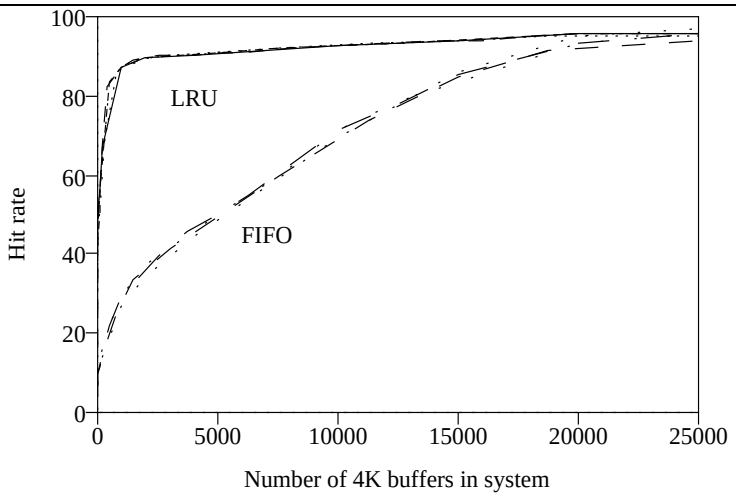

Figure 3.11: Results of I/O-node caching simulation. Each line represents a complete run of the simulation with a fixed number of I/O nodes. Although they are hard to distinguish, there are lines for 1, 2, 5, 10, 15, and 20 I/O nodes.

nodes or spread over many I/O nodes. That is, the hit rates were similar; performance is another issue. The success of such a small cache, coupled with the apparent lack of intraprocess locality in many jobs (Figure 3.10), reconfirms the presence of a high degree of interprocess spatial locality.

As a final test, we simulated the combination of a single buffer per compute node and a cache at each of 10 I/O nodes. The result was only about a 3% reduction in the I/O-node hit rate when each I/O node had a small cache of 50 buffers. This further suggests that when caching was limited to the I/O nodes, most of the hits were indeed a result of interprocess locality because, as Figure 3.10 shows, the limited intraprocess locality was filtered out by the compute-node cache.

Note the contrast with Miller and Katz's tracing study [MK91], which found little benefit from caching. They did notice a benefit from prefetching and write-behind. Both their workload and ours involve sequential access patterns; the difference is that the small requests in our access pattern lead to intraprocess spatial locality, and the distribution of a sequential pattern across parallel compute nodes leads to interprocess spatial locality, both of which could be successfully captured by caching.

3.5 Workload Characterization of a CM-5

Working with researchers from Duke University and Thinking Machines Corporation, we performed a similar workload characterization on a Thinking Machines CM-5 at the National Center for Supercomputing Applications [PEK+95, Pur96]. The CM-5 had 512 compute nodes and a much more powerful I/O subsystem than the iPSC/860. While the iPSC/860 was used primarily by control-parallel applications written in Fortran or C, most of the applications on the CM-5 were data-parallel programs written in CMF, a data-parallel dialect of Fortran. The CM-5 also had a number of control-parallel applications written in Fortran or C, which used the CMMD library for message-passing and I/O. The iPSC/860 was used mostly for computational fluid dynamics applications, but the CM-5 at NCSA was used for a wide variety of scientific applications.

Despite the great differences between the machines and the environments, we found that the file system workloads had broad areas of similarity.

While the number of write-only files outnumbered the read-only files by 3 to 1 on CFS, the ratio was less than 2 to 1 on the CM-5. We hypothesize that the number of write-only files on CFS was so high because many applications found it easier to manage a separate output file for each compute node, and then coalescing them during postprocessing. Since the applications on the CM-5 were predominantly data-parallel, this file-per-CP model is less natural. This hypothesis is further supported by the total number of files opened per application: on the iPSC, most applications opened 4 or more files; on the CM-5, most opened only 1 or 2.

On the CM-5 there was little difference between the amount of data written to write-only files and read from read-only files. In both cases, the amount of data transferred was much greater than under CFS. Since the CM-5 had a more powerful I/O subsystem, as well as many times the amount of disk space, this result is also not surprising. Like the iPSC, the CM-5 workload had few temporary files.

Individual requests on the CM-5 were larger than we observed on the iPSC, but since the vast majority were under 1000 bytes, they were still far smaller than conventional wisdom would have led us to expect. Like the iPSC, there was a great deal of regularity in the interval sizes; almost all files had only 1 or 2 intervals. There was less regularity of request sizes on the CM-5, but still fewer than 20% of the files had 4 or more different request sizes.

Like the iPSC, we observed a great deal of byte and block sharing. The sharing in control-parallel (CMMD) programs was on the same order as the sharing on the iPSC: 61% of the read-only files had all of their bytes shared and 93% of the write-only files had none of their bytes shared. The data-parallel (CMF) programs had much less byte sharing; 60% of the read-only files had 1% or fewer of their bytes shared by multiple processors. This lack of sharing probably reflects CMF's data-distribution model: each processor is statically assigned a disjoint subsection of a matrix.

Although there were differences in the workloads, there were significant similarities as well. Perhaps most importantly, it appears that there is a natural tendency for parallel programs to access files in regular patterns composed of small, non-contiguous chunks.

Chapter 4

Access Pattern Analysis

To better understand some of the results from the previous chapter, and the cause of those results, we performed a more detailed analysis of the patterns in which individual nodes accessed data in individual files.

4.1 Access Patterns

Most parallel file systems have been optimized to support large (many kilobyte) file accesses. The workload study described in the previous chapter shows that while some parallel scientific applications do issue a relatively small number of large requests, there are many applications that issue thousands or millions of very small requests, putting a great deal of stress on current file systems.

A common characteristic of many file-system workloads, particularly scientific file-system workloads, is that files are accessed consecutively [OCH+85, BHK+91, MK91]. In the parallel file-system workload, we found that while almost 93% of all files were accessed sequentially, consecutive access was primarily limited to those files that were only opened by one compute node. When files were opened by just a single node, 93% of those files were accessed *strictly consecutively* (i.e., every access began immediately after the previous access), but when files were opened by multiple nodes, only 15% of those nodes accessed the file strictly consecutively.

Recall that an *interval* is the distance between the end of one access and the beginning of the next. Although we found that almost 99% of all files were accessed with fewer than 3 different intervals, that finding made no distinction between files accessed by a single node and files accessed by multiple nodes. Looking more closely, we found that while 51% of all multi-node files were

accessed at most once by each node (i.e., there were no intervals at all) and 16% of all multi-node files had only 1 interval size, over 26% of multi-node files had 5 or more different interval sizes. Since previous studies have shown that scientific applications rarely access files randomly [MK91], the fact that a large number of multi-node files have many different interval sizes suggests that these files are being accessed in some complex, but possibly regular, pattern.

4.1.1 Strided Accesses

Although files may be opened by multiple nodes simultaneously, we are only interested here in the accesses generated by individual nodes. When necessary to avoid confusion, we use the term *node-file* to discuss a single node's usage of a file. We refer to a series of requests to a node-file as a *simple-strided* access pattern if each request is the same size and if the file pointer is incremented by the same amount between each request. One simple way this pattern could occur in practice is if each process in a parallel application reads a single column of data from a two-dimensional matrix stored on disk in row-major order. Another possibility is shown in Figure 4.1. In this example, an application has distributed the columns of a two-dimensional matrix across its processors in a cyclic pattern. Note that the number of columns must be evenly divisible by the number of compute nodes. Otherwise, the distance between columns within a row would be different than the distance between the last column in one row and the first column in the next row, breaking the simple-strided pattern.

Since a strided pattern was less likely to occur in single-node files, and since it could not occur in files that had only one or two accesses, we looked only at those files that had three or more requests by multiple nodes.[1] Figure 4.2 shows that many of the accesses to these files appeared to be part of a simple-strided access pattern. Although consecutive access was far more common in single-node files, it does occur in multi-node files. Since consecutive access could be considered a simple form of strided access (with an interval of 0), Figure 4.2 shows the frequency of strided accesses with and without consecutive accesses included. In either case, over 80% of all the files we examined were apparently accessed entirely with a strided pattern.

We define a *strided segment* to be a group of requests that appear to be part of a simple-strided

[1] Although we only looked at a restrictive subset of files, they account for over 93% of the I/O requests in the entire traced workload.

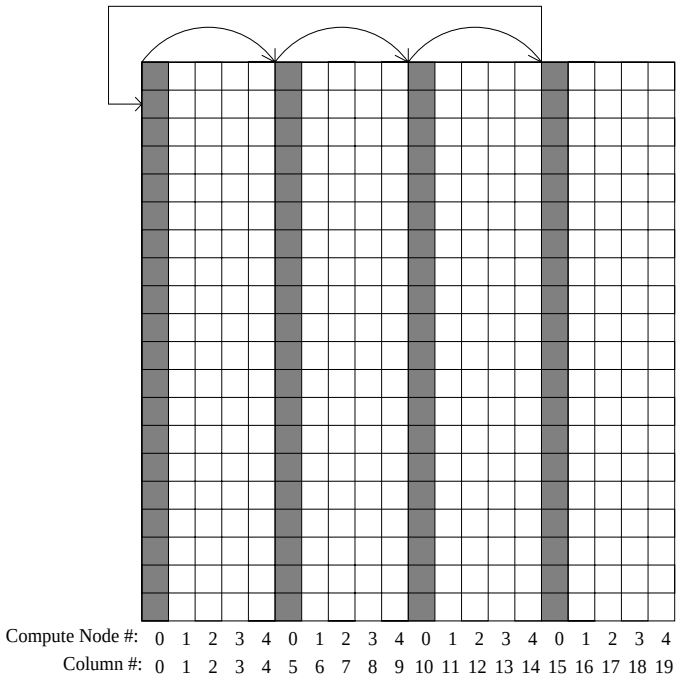

Figure 4.1: The columns of this 20x20 matrix have been distributed cyclically across the 4 compute nodes in an application. The columns assigned to node 0 are highlighted. If the matrix were composed of 8-byte doubles and stored on disk in row-major order, the I/O pattern would have an stride of of 32 (4*8) bytes.

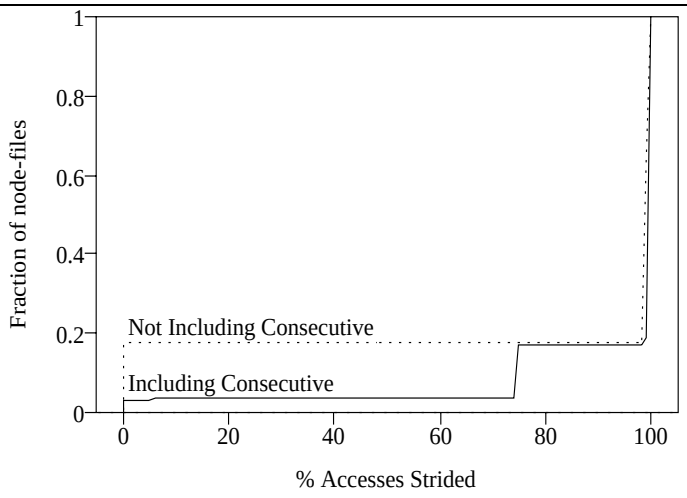

Figure 4.2: Cumulative distribution of node-files according to the fraction of accesses that were involved in a simple-strided pattern. This graph covers both the case where consecutive accesses are counted as strided (with an interval of 0) and the case where they are not.

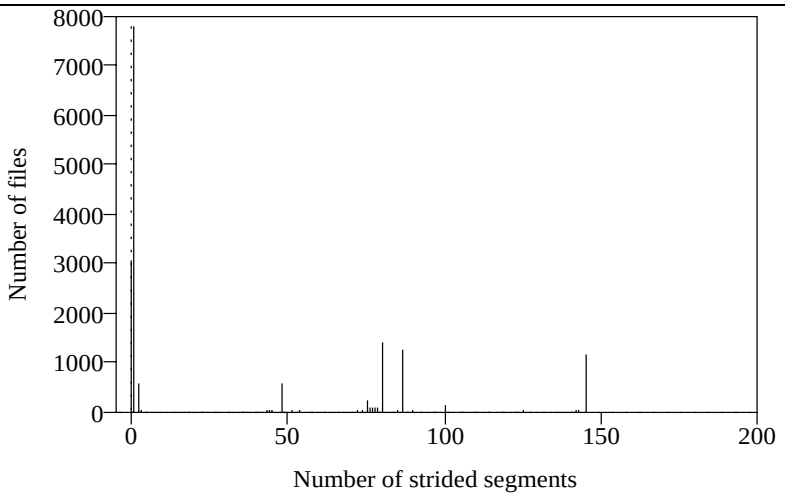

Figure 4.3: The number of different strided segments in each node-file. We have ignored segments of fewer than 10 accesses.

pattern. A segment's *length* is the number of requests that comprise that segment. Figure 4.2 only shows the percentage of requests that were involved in some strided segment; it does not tell us whether the requests are all part of a single strided segment that spans the whole file, or if each file had many segments with only a few requests in each. Figure 4.3 shows that most files had only a few strided segments, it was still common for a node-file to be accessed in many strided segments. Since we were only interested in those cases where a file was clearly being accessed in a strided pattern, this figure does not include short segments (fewer than 10 accesses) that may appear to be strided. Furthermore, in this graph we did not consider consecutive access to be strided. Despite using these fairly restrictive criteria for 'strided access', we still found that it occurred frequently. Although Figure 4.4 indicates that most segments fell into the range of 20 to 30 requests, Figure 4.5 shows that there were quite a few long segments as well.

Although the existence of these simple-strided patterns is interesting and potentially useful, the fact that many files were accessed in multiple short segments suggests that there was a level of structure beyond that described by a simple-strided pattern.

4.1.2 Nested-strided Accesses

A *nested-strided* access pattern is similar to a simple-strided access pattern but rather than being composed of simple requests separated by regular strides in the file, it is composed of strided

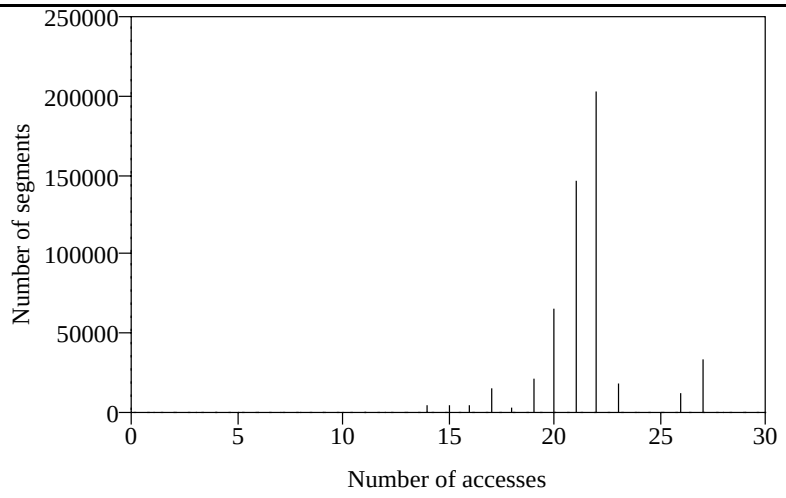

Figure 4.4: The number of segments of a given length (including 'short' segments of 10 or fewer accesses). By far, most segments have between 20 and 30 accesses.

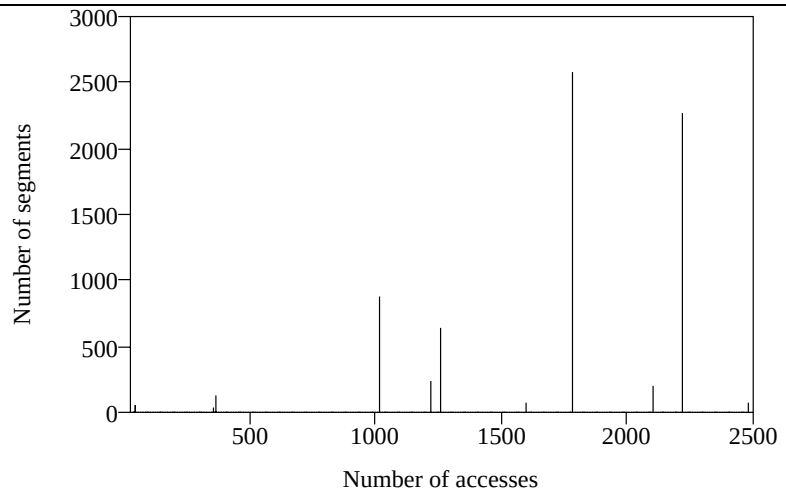

Figure 4.5: The tail of the segment length distribution shown in the previous figure. There are quite a few very long strided segments.

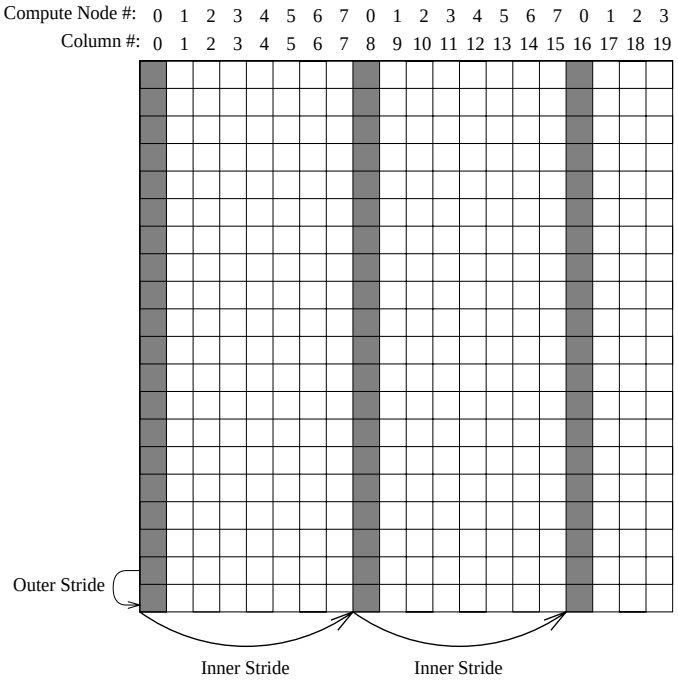

Figure 4.6: The columns of this 20x20 matrix have been distributed cyclically across the 8 compute nodes in an application. The columns assigned to node 0 are highlighted. If the matrix were composed of 8-byte doubles and stored on disk in row-major order, the I/O pattern would have an inner stride of 64 (8*8) bytes and an outer stride of 160 (20*8) bytes.

segments separated by regular strides in the file. A singly-nested pattern is the same as a simple-strided pattern. A doubly-nested pattern could correspond to the pattern generated by an application that distributed the columns of a matrix stored in row-major order across its processors in a cyclic pattern, if the columns could not be distributed evenly across the processors (Figure 4.6). The simple-strided sub-pattern corresponds to the requests generated within each row of the matrix, while the top-level pattern corresponds to the distance between one row and the next. This access pattern could also be generated by an application that was reading a single column of data from a three-dimensional matrix. Higher levels of nesting could occur if an application mapped a multidimensional matrix onto a set of processors.

Table 4.1 shows how frequently nested patterns occurred. Files with zero levels of nesting had no strided accesses, and those with one level had only simple-strided accesses. Interestingly, it was far more common for files to exhibit three levels of nesting than two. This tendency suggests that

Maximum Level of Nesting	Number of node-files
0	469
1	10945
2	747
3	5151
4+	0

Table 4.1: The number of node-files that use a given maximum level of nesting.

many of the applications in this environment were using multidimensional matrices.

4.1.3 CM-5

We performed a similar analysis of the workload on the CM-5 discussed in the previous chapter. Those applications written in the data-parallel CM Fortran had a single file pointer for the whole application. Conceptually, individual CPs did not access files; the application as a whole performed all file accesses. Consequently, we found that most of those data-parallel applications exhibited little or no explicitly strided access. Since we hypothesize that most of the strided access patterns are the result of distributing data across multiple CPs in an application, we would not expect to see this form of access in data-parallel applications, where there is no notion of a per-CP file access. Of course, the actual movement of data between CPs and IOPs is likely to be similar to that seen on the iPSC, but that data movement is transparent to the application programmer. Those CM-5 programs that were written in the control-parallel paradigm, with one file pointer for each CP, did exhibit a high degree of strided access. Like the iPSC, most of the applications had a single level of nesting, and there were more with three levels than with two levels. Unlike the iPSC, there were a small number of applications with more than 3 levels of nesting.

Chapter 5

Design and Implementation of Galley

Galley is a new multiprocessor file system, which we designed in response to the results of our workload studies.

5.1 Design Goals

Most current multiprocessor file system designs are based primarily on hypotheses about how parallel scientific applications would use that file system. Before we began the detailed design of Galley, we laid out a number of high-level design goals, which are the result of examining how parallel scientific applications actually use existing file systems:

- allow applications and libraries to explicitly control parallelism in file access,
- efficiently handle a variety of access sizes and patterns,
- be flexible enough to support a wide variety of interfaces and policies, implemented in libraries,
- allow easy and efficient implementations of libraries,
- be scalable enough to run well on multiprocessors with tens or hundreds of nodes,
- minimize memory and performance overhead.

5.2 File Structure

Most parallel file systems are based on a Unix-like, linear file model [BGST93, LIN+93, Pie89]. Under this model, a file is seen as an addressable, linear sequence of bytes. Applications can issue

requests to read or write contiguous subranges of that sequence of bytes. A parallel file system typically declusters the data within those files (i.e., scatters the blocks of each file across multiple disks), allowing parallel access to the file. This parallel access reduces the effect of the bottleneck imposed by the relatively slow disk speed. Although the file is actually scattered across many disks, the underlying parallel structure of the file is hidden from the application.

Galley uses a more complex file model that allows greater flexibility, and which should lead to higher performance. As we show in Chapter 7, it is possible to support the traditional linear file model on top of Galley's more complex structure.

5.2.1 Subfiles

The linear file model offered by most multiprocessor file systems can give good performance when the request size generated by the application is significantly larger than the declustering unit size, as a single request will involve data from multiple disks. Under these conditions, the file system can access multiple disks in parallel, delivering higher bandwidth to the application, and possibly hiding any latency caused by disk seeks. The drawback of this approach is that most multiprocessor file systems use a declustering unit size measured in kilobytes (e.g., 4 KB in Intel's CFS), but our workload characterization studies show that the typical request size in a parallel application is much smaller: frequently under 200 bytes. This disparity between the request size and the declustering unit size means that most of the individual requests generated by parallel applications are not being executed in parallel. In the worst case, the compute processors in a parallel application may issue their requests in such a way that all of an application's processes may first attempt to access disk 0 simultaneously, then all attempt to access disk 1 simultaneously, and so on.

Another drawback of the linear file model is that a dataset may have an efficient, parallel mapping onto multiple disks that is not easily captured by the standard declustering scheme. One such example is the two-dimensional, cyclically-shifted block layout scheme for matrices, shown in Figure 5.1, which was designed for SOLAR, a portable, out-of-core linear-algebra library [TG96]. This data layout is intended to efficiently support a wide variety of out-of-core algorithms. In particular, it allows blocks of rows and columns to be transferred efficiently, with a high degree of I/O parallelism, as well as square or nearly-square sub-matrices.

To avoid the limitations of the linear file model, Galley does not impose a declustering strategy

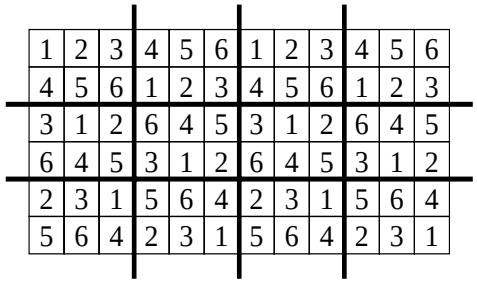

Figure 5.1: An example of a 2-dimensional, cyclically-shifted block layout, as described in [TG96]. In this example there are 6 disks, logically arranged into a 2-by-3 grid, and a 6-by-12 block matrix. The number in each square indicates the disk on which that block is stored.

on an application's data. Instead, Galley provides applications with the ability to fully control this declustering according to their own needs. This control is particularly important when implementing I/O-optimal algorithms [CK93]. Applications are also able to explicitly indicate which disks they wish to access in each request. To allow this behavior, files are composed of one or more *subfiles*, which may be directly addressed by the application. Each subfile resides entirely on a single disk, and no disk contains more than one subfile from any file. By default a file will contain one subfile on every disk in the system, but the application may also choose how many subfiles a file contains when the file is created. The application may even choose on which disks the file will be created. The number of subfiles remains fixed throughout the life of the file.

The use of subfiles gives applications the ability both to control how the data is distributed across the disks, and to control the degree of parallelism exercised on every subsequent access. Of course, many application programmers will not want to handle the low-level details of data declustering, so we anticipate that most end users will use a user-level library (such as SOLAR) that provides an appropriate declustering strategy, but hides the details of that strategy from the end user.

5.2.2 Forks

Each subfile in Galley is structured as a collection of one or more independent *forks*. A fork is a named, addressable, linear sequence of bytes, similar to a traditional Unix file. Unlike the number of subfiles in a file, the number of forks in a subfile is not fixed; libraries and applications may add

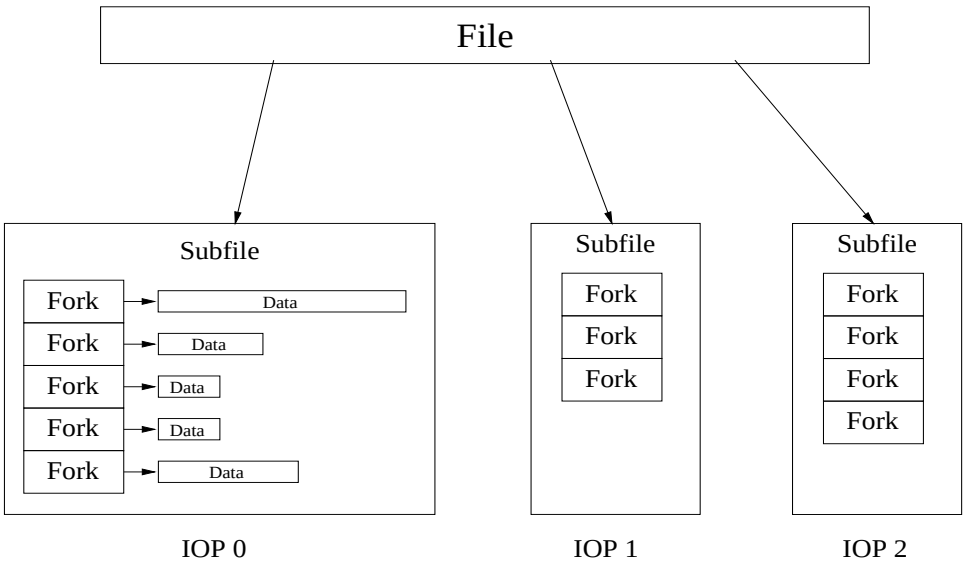

Figure 5.2: Three dimensional structure of files in the Galley File System. The portion of the file residing on disk 0 is shown in greater detail than the portions on the other two disks.

forks to, or remove forks from, a subfile at any time. There is no requirement that all subfiles have the same number of forks, or that all forks have the same size. The final, three-dimensional file structure is illustrated in Figure 5.2.

The use of forks allows further application-defined structuring. For example, if an application represents a physical space with two matrices, one containing temperatures and other pressures, the matrices could be stored in the same file (perhaps declustered across multiple subfiles) but in different forks. In this way, the related information is stored logically together but each matrix may be accessed independently.

While typical application programmers may find forks helpful, they are most likely to be useful when implementing libraries. In addition to storing data in the traditional sense, many libraries also need to store persistent, library-specific "metadata" independently of the data proper. One example of such a library would be a compression library similar to that described in [SW95a]. Rather than compressing the whole file at once, making it difficult to modify or extract data in the middle of the file, the file is broken into a series of chunks, which are then compressed independently. With Galley, such a library could store the compressed data chunks in one fork and the necessary index information about those chunks in another.

Another instance where this type of file structure may be useful is in the problem of genome-sequence comparison. This problem requires searching a large database to find approximate matches between strings [Are91]. The raw database used in [Are91] contains thousands of genetic sequences, each of which is composed of hundreds or thousands of bases. To reduce the amount of time required to identify potential matches, the authors constructed an index of the database that was specific to their needs. Under Galley, this index could be stored in one fork, while the database itself could be stored in a second fork.

A final example of the potential use of forks is Stream*, a parallel file abstraction for the data-parallel language, C* [MHQ96]. Stream* divides a file into three distinct segments, each of which corresponds to a particular set of access semantics. Although the current implementation of Stream* stores all the segments in a single file, one could use a different fork for each segment. In addition to the raw data, Stream* maintains several kinds of metadata, which are currently stored in three different files: `.meta`, `.first`, and `.dir`. In a Galley-based implementation of Stream*, it would be natural to store this metadata in separate forks rather than separate files.

5.2.3 Namespace

Efficiently supporting a mature, useful naming system in a scalable parallel file system is a complex problem. Given this complexity, and the fact that it was not the focus of our research, we have avoided most of the issues involved by limiting Galley to a much simpler naming structure than most sequential or parallel file systems. Rather than providing a hierarchical structure, Galley's namespace is flat. Conceptually, this means that every file is stored in the root directory, and there are no subdirectories.

This simple naming structure is one of the more serious shortcomings that would need to be addressed before Galley could be considered for practical use.

5.3 Compute Processors (CPs)

Beyond the assumption that the file system contains multiple disks, the file model discussed in the previous section does not depend on any specific file system structure. As we discuss below, Galley is based on the client-server model.

A client in Galley is simply any user application that has been linked with the Galley run-time

library, and which runs on a compute processor. The run-time library receives file-system requests from the application, translates them into lower-level requests, and passes them, as messages, directly to the appropriate servers, running on I/O processors. The run-time library then handles the transfer of data between the I/O processors and the compute node's memory.

As far as Galley is concerned, every compute processor in an application is completely independent of every other compute processor. Indeed, Galley does not assume that one compute processor is even aware of the existence of other compute processors. This independence means that Galley does not impose any communication requirements on a user's application, so applications may use whichever communication software (e.g., MPI, PVM, P4) is most suitable to the given problem. Indeed, applications are not limited to the control-parallel, message-passing paradigm; data-parallel applications can be, and have been, implemented on top of Galley.

Like most multiprocessor file systems, Galley offers both blocking and non-blocking I/O. To simplify the implementation, and to avoid binding Galley too tightly to a single architecture, Galley originally used multi-threading to implement non-blocking I/O. Unfortunately, most of the major communications packages cannot function in a multi-threaded environment. Since we cannot rely on the availability of thread support, Galley is designed to use Unix signals to implement non-blocking I/O, which in turn requires that we use a TCP/IP communications substrate. Clearly, Unix signals were not intended to support high-performance I/O, so this approach may affect the performance of some applications. The application programmer will have to decide if the performance gained by overlapping I/O and computation outweighs the performance impact of our signal-based non-blocking I/O. If support for multi-threaded applications ever becomes commonplace in message-passing packages, our original approach would likely be preferable.

5.4 I/O Processors (IOPs)

Galley's I/O servers, illustrated in Figure 5.3, are composed of several units, which are described in detail below. Galley's IOPs do not communicate between themselves and they use only TCP/IP to communicate with CPs. Since IOPs do not interact with any high-level message-passing library, we were able to implement each unit as a separate thread. Each IOP also has one thread designated to handle incoming I/O requests for each compute processor. When an IOP receives a request from a CP, the appropriate CP thread interprets the request, passes it on to the appropriate worker

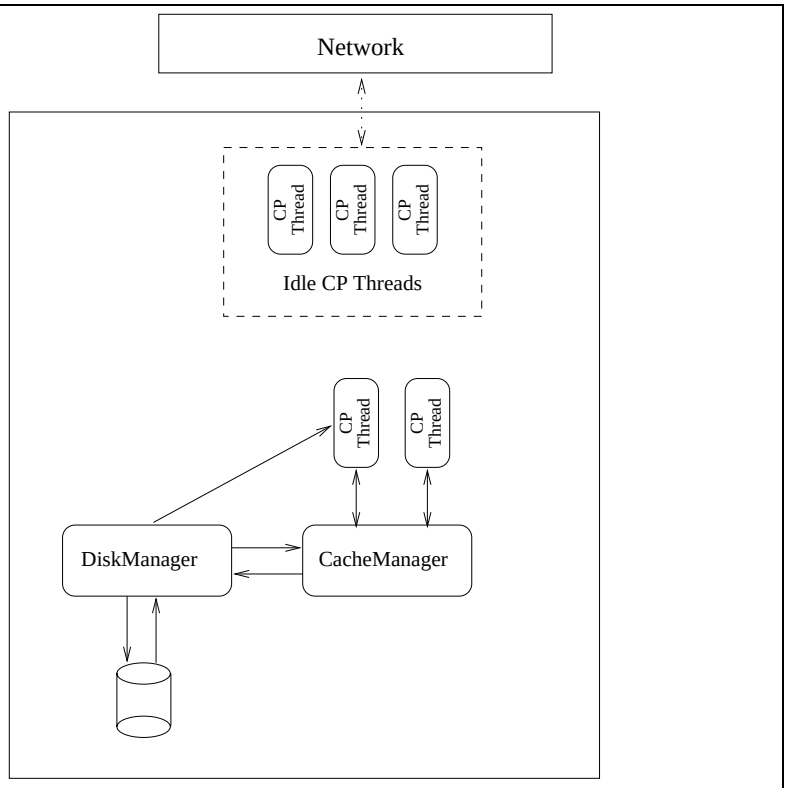

Figure 5.3: Internal structure of a Galley I/O Processor, showing two active data requests waiting for blocks from the CacheManager and/or the DiskManager, and three idle CP threads. All threads execute in the same address space.

thread, and then handles the transfer of data between the IOP and the CP. This multi-threading makes it easy for an IOP to service requests from many clients simultaneously.

While one potential concern is that this thread-per-CP design may limit the scalability of the system, we have not observed such a limitation in the practice. One may reasonably assume that a thread that is idle (i.e., not actively handling a request) is not likely to noticeably affect the performance of an IOP. By the time the number of active threads on a single IOP becomes great enough to hinder performance, the IOP will most likely be overloaded at the disk, the network interface, the memory, or the buffer cache, and the effect of the number of threads will be minor relative to these other factors.

5.4.1 CP Threads

CP threads remain idle until a request arrives from the corresponding CP. After being awakened to service a new request, a CP thread creates a list of all the disk blocks that will be required to satisfy the request. The CP thread then passes the full list of blocks to the CacheManager, and waits on a queue of buffers returned by the CacheManager and DiskManager. As buffers arrive on a CP thread's queue, the thread moves the data between those buffers and the thread's CP (possibly using an intermediate buffer for packing and unpacking small records). When a CP thread finishes all the data movement for a buffer, it decreases that buffer's reference count (discussed more in the next section), and handles the next buffer in the queue. When the whole request has been satisfied, or if it fails in the middle, the thread passes a success or failure message back to its CP, and idles until another request arrives.

The order in which a fork's blocks are placed on the CP thread's buffer queue is determined by which blocks are present in the buffer cache and the order in which that fork's blocks are laid out on disk. Therefore, it is not possible for Galley's client-side run-time library to know in advance the order in which an IOP will satisfy the individual pieces of a request. So, when reading, before the IOP can send data to the CP, it must first send a message indicating what data will be sent. Similarly, when writing, the IOP must send a message to the CP indicating which portion of the data the IOP is ready to receive. When writing, this approach is somewhat unusual in that the IOP is essentially 'pulling' the data from the CP, rather than the traditional model, where the CP 'pushes' the data to the IOP.

There is a further complication in transferring data between CPs and IOPs: packing. Rather than sending lots of small packets across the network, when possible Galley packs multiple small chunks of data into a larger buffer, and sends the larger buffer when it is full. This packing reduces the aggregate latency, and increases the effective data-transfer bandwidth. In the current implementation, the list of data chunks is precomputed on the CP and the whole list is sent to the IOP.[1] On our testbed systems, the speed of the network relative to the speed of the processors is high enough that sending the list across the network makes more sense than computing the list on the CPs and the IOPs.

[1] Note, this list is specified in logical fork-level chunks, not low-level disk-block chunks.

For simplicity, within a single packet the IOP will only pack chunks in the order they appear in the chunk list. If an out-of-order block is placed on a CP thread's queue, the current buffer is flushed, even if it is not full, and a new pack buffer is started. An early implementation of Galley supported out-of-order packing within a packet, but that approach required that a fairly large packet of 'control' data be sent to the CP with each flushed buffer. The current implementation is less flexible, but appears to have higher performance on our testbeds. On a system with a higher-bandwidth, lower-latency network, out-of-order packing might be more efficient, as the cost of the extra control data would be reduced.

5.4.2 CacheManager

Each IOP has a buffer cache that is maintained by the CacheManager. Each buffer has an associated reference count, which indicates how many CP threads are waiting to use data stored in that buffer. In addition to deciding which blocks are kept in the buffer, the CacheManager does all the work involved in locating blocks in the buffer cache for CP threads. To perform these lookups, the CacheManager maintains a separate list of disk blocks requested by each thread. When the CacheManager has outstanding request lists from multiple threads, it services requests from each list in round-robin order. This round-robin approach is an attempt to provide fair service to each requesting CP.

The CacheManager maintains a global LRU list of all the blocks resident in the cache. When a new block is to be brought into the cache, this list is used to determine which block is to be replaced. Providing applications with more control over cache policies is one possible area for future research.

Rather than performing lookups by scanning through the entire LRU list, for efficiency the CacheManager also maintains a hash table, containing pointers to all the blocks in the cache. For each disk block requested, the CacheManager searches its hash table of resident blocks. If the block is found, its reference count is increased, a pointer to that buffer is added to the requesting thread's ready queue, and the block is moved to the most-recently-used end of the LRU list. If the block is not resident in the cache, the CacheManager finds the first block in the LRU list with a reference count of 0, and schedules it to be replaced by the requested block. The buffer is then marked 'not ready', and a request is issued to the DiskManager to write out the old block (if necessary) and to read the new block into the buffer. Once a block has been scheduled for eviction, it cannot be

"recalled"; if another request arrives for that block, it will have to be reread from disk.

5.4.3 DiskManager

The DiskManager logically partitions a disk into 32 KB blocks, and accepts requests from the CacheManager to read or write those blocks. The DiskManager maintains a list of pending block requests. As new requests arrive from the CacheManager, they are placed into the list according to the disk scheduling algorithm. The DiskManager currently uses a Cyclical Scan algorithm [SCO90]. When a block has been read from disk, the DiskManager updates the cache status of that block's buffer from 'not ready' to 'ready' and adds it to the requesting threads' ready queues.

The DiskManager is also responsible for keeping track of which blocks on the disk are not assigned to any fork, and for allocating new blocks to forks as they grow. Like FFS, and unlike the original Unix file system, Galley uses a bitmap to keep track of the empty blocks on a disk. Given Galley's 32 KB block size, a single bitmap block contains 262144 bits, which is sufficient to maintain information about 8 GB of disk space. For a large disk or RAID, multiple bitmap blocks would be used. These bitmap blocks are stored near the "front" of the disk, and are flushed every time the disk scheduler reaches the front of its cycle. Although this regular flushing provides some measure of safety, it is possible for the file system to be left in a highly inconsistent state following a crash. This problem would certainly need to be addressed before Galley could be considered ready for production use.

Many file systems perform some kind of prefetching; data is read from the disk into the buffer cache before any process actually requests it. Prefetching is an attempt to reduce the latency of file access perceived by the process. Galley's DiskManager does not attempt to prefetch data for two reasons. First, indiscriminate prefetching can cause thrashing in the buffer cache [Nit92]. Second, prefetching is based on the assumption that the system can intelligently guess what an application is going to request next. Using the higher-level requests described below, there is frequently no need for Galley to make guesses about an application's behavior; the application is able to explicitly provide that information to each IOP.

To increase portability, Galley does not use a system-specific low-level driver to directly access the disk. Instead, Galley relies on the underlying operating system (presumably Unix) to provide such services. Galley's DiskManager has been implemented to use raw devices, Unix files, or simu-

lated devices as "disks". Galley's disk-handling primitives are sufficiently simple that modifying the DiskManager to access a device directly through a low-level device driver for higher performance is likely to be a trivial task.

5.4.4 Metadata

The preceding sections discuss the functional units that move the data between the disk, the cache, and the CPs. We now discuss some of the details of how that data is stored and retrieved. Figure 5.4 shows how the pieces discussed below all tie together.

Naming

When an application wishes to access the data in a file, it must first locate that file's metadata, or the information that describes where the actual data is stored. Each file in Galley has a single *file header block* associated with it. The file header block contains the number of subfiles in the file, and a list of *subfile header IDs*. A subfile header ID is an `(IOP, block_num)` pair, which indicates on which IOP a subfile is stored, and which disk block on that IOP contains the subfile's header block.

Rather than storing all the file header blocks on a single IOP, which would create a single point of congestion that could limit the system's scalability, the file header blocks are distributed across all the IOPs in the system. To find the IOP that manages a given file's header block, a simple hash function is applied to the file name. Given the simplicity of Galley's naming system, this approach is reasonably simple to implement efficiently. Vesta uses a similar hashing scheme for their naming system [CF96].

A subfile header block contains information about all the forks within the subfile. In particular, a fork's entry in the subfile header block contains its name, its size, and the disk addresses of its *mapping blocks*, which are discussed below. The current implementation of Galley allows each subfile to have only a single, 32 KB header block, limiting subfiles to 256 forks. If this limit should prove problematic in practice, extending or eliminating it would require little work.

Block mapping

Although Galley's forks resemble traditional Unix files, Galley does not adopt Unix's method of mapping these files to blocks. Rather than using a hierarchical structure of direct, indirect, and

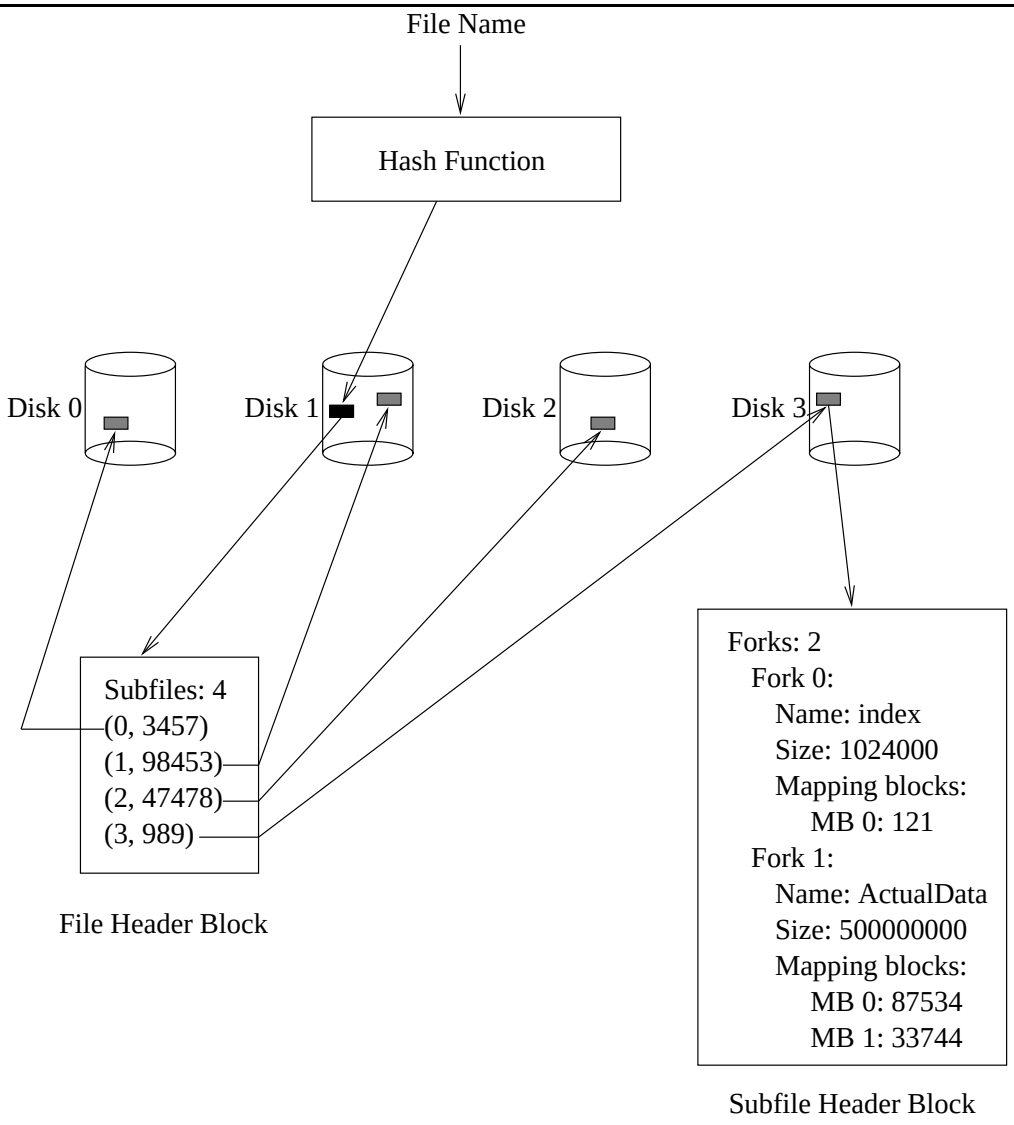

Figure 5.4: Diagram of Galley's metadata structures from the file level down to the fork level.

doubly-indirect blocks to map disk block to forks, Galley uses a one-level collection of *mapping blocks*. Each of these mapping blocks is similar in nature to a direct block in the Unix mapping system. That is, each mapping block contains a series of block numbers, each of which maps a single block in the fork to a single block on disk. This approach is simpler than the multi-level approach used by Unix file systems, and allows a simpler implementation.

Since Galley's blocks are larger than those on most Unix file systems, we need fewer mapping entries to map the same amount of data onto disk. For example, a Unix file system that used a 4 KB block size would need 8 times as many mapping entries to map a file onto disk as Galley would to map the same file to disk. Furthermore, since Galley's blocks are larger, each mapping block contains more entries: again, 8 times as many as a file system with a 4 KB block size. Under Galley, a single mapping block contains 8192 mapping entries, enough to map 256 megabytes of data. The current implementation limits a single fork to 4 GB of data. Note that this limitation allows us to specify offsets into a fork with a single 32-bit integer.

One special case of block mapping occurs when a fork contains a *hole*. For example, if an application wrote data only in block 0 and block 2, then block 1 will not contain any data. As in Unix, we treat such a block as if it were filled with zeros. While we could certainly assign a disk block to each such empty block, that would consume unnecessary disk space, and require disk accesses that could be avoided. Instead, Galley maps these empty blocks to disk block 0. Since Galley uses disk block 0 to store metadata about the file system itself, we know that no ordinary file will ever have an actual block assigned to disk block 0. If an application ever attempts to write to a block that is mapped to 0, Galley assigns an actual disk block to replace the empty block. If an application asks to read data from an empty block, Galley sends a short message to the requesting CP, notifying it that the block is empty. The CP then fills the appropriate buffer with zeros. Not only does this approach avoid a disk access, it avoids sending a block of data across the network.

The Unix approach is optimized for small files. The workload studies discussed in [OCH+85, BHK+91] show that most files in a Unix workload can be addressed using just the direct-mapping entries in the inode. Few files will require an indirect block and fewer still will require a doubly-indirect block. Since we have seen that files in a scientific environment (both vector supercomputer and parallel) tend to be much larger than a Unix environment, and tend to have greater variability in size, Galley adopts a simpler approach to block mapping that does not favor any particular file

size.

5.5 Application Interface

Given the new file model implemented by Galley, and the observed frequency of regular access patterns in multiprocessor file system workloads, it was not sufficient to simply provide applications with a traditional Unix-like interface. Thus, Galley provides a new interface that is intended to better meet the needs of scientific applications. Although applications may certainly be written directly to Galley's interface, it is intended primarily to allow the easy implementation of libraries. We anticipate that these libraries will provide the higher-level functionality needed by most users.

5.5.1 File Operations

Files in Galley are created using the `gfs_create_file()` call. In addition to specifying a file name, an application may specify on how many IOPs, and even on which IOPs, the file is to be created. A `gfs_create_file()` call is completed in three steps. The first step is to verify that the name chosen for the file is not already in use, and to *reserve* the name if it is available. This step requires that a single message be sent to the IOP that will be responsible for maintaining the metadata for the new file. The responsible IOP is chosen by applying a hash function to the file name, as discussed earlier. The second step is to create subfiles on each of the appropriate IOPs. This step requires that a message be sent to each IOP, asking that a subfile header block be assigned to the file. Each IOP returns either the ID of the assigned header block, or an error code. If this step fails on any IOP (e.g., if it is out of disk space), then each IOP is instructed to release the newly assigned header blocks, the reserved file name is released, and the appropriate error code is returned to the application. The final step of a successful file-creation process is to store the file name, along with all the subfile header block IDs, on disk at the responsible IOP and to return a success code to the application. Note that after the file is created, all the subfiles are empty; no forks are created as part of the file-creation process. Also, the file is not opened as part of this process.

As far as Galley is concerned, each compute node in an application is a completely independent entity. Therefore, Galley has no notion of a *leader*, a node that can issue requests on behalf of other processors. Thus, each node in an application that wishes to use a file in Galley must explicitly open that file using the `gfs_open_file()` call. Although it is conceivable that requiring every

node to open a file independently could lead to performance bottlenecks, we have not observed any such problems in practice. When an application issues a `gfs_open_file()` call, the run-time library sends a request to the appropriate metadata server (again, determined by hashing the file name). If the file exists, the metadata server returns a list of all the subfile header block IDs to the requesting CP. The run-time library assigns the open file a *file ID*, and caches the list of header block IDs in an *open-file table* to avoid repeated requests to the metadata server. Since these IDs do not change during the course of the file's lifetime, we do not have to be concerned that the cached IDs will become inconsistent with the IDs stored at the metadata server. The run-time library then sends messages to each of the IOPs on which the file has a subfile, notifying the IOP that the subfile has been opened. The IOP then either sets up a small amount of state, or increases a reference count if another CP has already opened the subfile. The reference count is decreased when a CP closes the file, or if an application that has it open completes or crashes.

The metadata server maintains no information about which CPs open a file, or even that the file has been opened. This lack of state at the metadata server means that it is possible for one compute processor to ask that a file be deleted (using `gfs_delete_file()`) while another CP is still using the file. Deleting a file in Galley is a two-step process. The first step simply involves removing some indexing information: the name and ID list stored at the metadata server. Since each CP that opens a file maintains a local cache of header block IDs, CPs that have already opened a file are not affected by the removal of that indexing information. The second step is asking each IOP on which the file was created to delete its subfile. If the reference count for that subfile is 0, it (and all of its forks) are actually deleted. If the reference count for that subfile is greater than 0, it is marked for deletion, and will be deleted when the reference count reaches 0. Thus, even if CP A requests that a file be deleted, while CP B is using the file, CP B will still be able to access the file's data until it closes the file.

5.5.2 Fork Operations

Forks are created using the `gfs_create_fork()` call, which takes as parameters the ID of an open file, the subfile in which the fork is to be created, and a name for the new fork. Galley's run-time library looks up the ID of the appropriate subfile header block in its cached list, and sends both the subfile header ID and the fork name to that subfile's IOP. By sending the subfile header ID

to the IOP, there is no need for an extra indexing operation to take place at the IOP; the IOP is able to retrieve the appropriate subfile header block immediately. The IOP adds the name of the fork to the subfile header block, and returns a success or error code to the CP. For the convenience of application programmers, Galley also provides a `gfs_all_create()` call, which creates a fork of the given name in each of the file's subfiles.

As with files, each process in an application that intends to access a fork's data must explicitly open that fork. Once again, while this requirement theoretically has the potential for reducing system performance, such behavior has not been observed in practice. Forks are opened using the `gfs_open_fork()` call, which takes the same parameters as the fork-creation call. If the fork-open request is successful, Galley returns a *fork ID*, which is used in subsequent calls, much like a file descriptor is used in Unix. Forks are closed with `gfs_close_fork()` and deleted with `gfs_delete_fork()`. As with files, if a CP attempts to delete a fork that has a non-zero reference count, that fork is marked for deletion, but is not actually deleted until its reference count reaches 0. For convenience, there are `gfs_all_open()`, `gfs_all_close()`, and `gfs_all_delete()` calls as well.

5.5.3 Data Access Interface

The standard Unix interface provides only simple primitives for accessing the data in files. These primitives are limited to `read()`ing and `write()`ing consecutive regions of a file. As discussed in the previous chapter, we have found that these primitives are not sufficient to meet the needs of many parallel applications. Specifically, parallel scientific applications frequently make many small requests to a file, with *strided* access patterns.

In addition to a simple, Unix-style interface, Galley provides three interfaces that allow applications to explicitly request data in regular, structured patterns, such as those described in Chapter 4, as well as one interface for complex, but unstructured requests. These interfaces allow the file system to combine many small requests into a single, larger request, which can lead to improved performance in several ways. First, reducing the number of requests can lower the aggregate latency costs, particularly for those applications that issue thousands or millions of tiny requests. Second, providing the file system with this level of information allows the IOPs to make intelligent disk-scheduling decisions, leading to fewer disk-head seeks, and to better utilization of the disks'

internal caches. Finally, the file systems can reduce the total number of messages transmitted between the CP and the IOP.

The data-access interfaces offered by Galley are summarized below. Note that each request accesses data from a single fork; Galley has no notion of a file-level read or write request.

Traditional requests

```
int gfs_read(int fid, void *buf, long offset, long size)
```

Beginning at `offset` in the open fork indicated by `fid`, the file system will read `size` bytes, and store them in memory at `buf`. The call returns the number of bytes transferred.

Naturally, there is a corresponding `gfs_write()` call.

Simple-strided requests

```
int gfs_read_strided(int fid, void *buf, long offset, long size,
                     long f_stride, long m_stride, int quant)
```

Beginning at `offset` in the open fork indicated by `fid`, the file system will read `quant` records, of `size` bytes each. The offset of each record is `f_stride` bytes greater than that of the previous record. The records are stored in memory beginning at `buf`, and the offset into the buffer is changed by `m_stride` bytes after each record is transferred. The call returns the total number of bytes transferred.

When `m_stride` is equal to `size`, data will be *gathered* from disk, and stored contiguously in memory. When `f_stride` is equal to `size`, data will be read from a contiguous region of a file, and *scattered* in memory. It is also possible for both `m_stride` and `f_stride` to be different than `size`, and possibly different than each other. Either the file stride (`f_stride`), the memory stride (`m_stride`), or both may be negative. Note that, when reading, if the memory stride is negative, but of a smaller magnitude than the record size, the records will overlap in memory. For example, if the memory stride is -5 and the record size is 10, the first record covers bytes 0 to 10, the second record covers bytes -5 to 5, and so on. Similarly, when writing, if the file stride is negative, it is possible for records to overlap in the file. Galley does not guarantee the order in which records will be transferred, so the system's behavior in such situations is undefined. Furthermore, the behavior is likely to be unrepeatable as well.

Nested-strided requests

```
int gfs_read_nested(int fid, void *buf, long offset, long size,
                    struct gfs_stride *vec, int levels)
```

The `vec` is a pointer to an array of `(f_stride, m_stride, quantity)` triples listed from the innermost level of nesting to the outermost. The number of levels of nesting is indicated by `levels`.

An example of the use of the nested-strided interface is shown in Figure 5.5. This example contains the code to allow a node to read its portion of a three-dimensional $M * M * M$ matrix. The matrix is stored in a single fork, in standard "C" order, and the matrix is to be distributed across the processors in a (BLOCK, BLOCK, BLOCK) fashion. For simplicity, we have assumed that we have the proper number of processors to distribute the data evenly. In this case that means we have $N * N * N$ processors, which are logically arranged in a cube. The processors are assigned ranks from 0 to $N^3 - 1$, starting at the top, left, front corner and proceeding to the back, right, bottom corner. So, processor 0 is at the top left of the front of the cube, processor $N^2 - 1$ is at the bottom right of the front of the cube, and processor N^2 is at the top left of the second plane of the cube. Using a traditional read/write interface, each node would have to issue $(M/N)^2$ requests to read its entire subcube. The nested-strided request reduces the number of requests issued by each node to one.

Although this code fragment looks complicated, it should be noted that it is essentially a proper subset of the code necessary to request each chunk individually, as would be necessary with a traditional interface. Furthermore, it is no more complex than in any other general-purpose, structured interface, such as MPI-IO or Vesta (which are discussed in Chapter 8).

Nested-batched requests

```
int gfs_read_batched(int fid, void *buf, struct gfs_batch *vec,
                     int quant)
```

Although we found that most of the small requests in the observed workloads were part of either simple-strided or nested-strided patterns, there may well be applications that could benefit from some form of high-level, regular request, but would find the nested-strided interface too restrictive. One example of such an application is given in Chapter 7. For those applications, we provide a

```
#define Q (M/N) /* Elements in each dimension assigned to a node */
#define ELT_SIZE sizeof(double)
#define ROW_SIZE (M * ELT_SIZE)
#define PLANE_SIZE (M * M * ELT_SIZE)

int
read_my_block(rank, kid, a)
    int rank;      /* This node's rank in the application */
    int kid;       /* The ID of the open fork with the data */
    double a[]);   /* Where the data will be stored */
{
    struct gfs_stride vec[2];
    u_long offset;
    u_long bytes;
    int x, y, z;     /* This node's location in the logical cube
                        of processors. */

    x = rank % N;
    y = (rank % (N*N))/N;
    z = rank / (N*N);

    offset = Q * ((x*ELT_SIZE) + (y*ROW_SIZE) + (z*PLANE_SIZE));

    /* Inner stride: Captures the data from one plane */
    vec[0].f_stride = ROW_SIZE;      /* Distance between two rows */
    vec[0].m_stride = Q * ELT_SIZE;  /* Size of data from one row */
    vec[0].quantity = Q;             /* Number of rows to read */

    /* Outer stride: Captures the data from all the planes */
    vec[1].f_stride = PLANE_SIZE;    /* Distance between two planes */
    vec[1].stride = Q*Q * ELT_SIZE;  /* Size of data from one plane */
    vec[1].quantity = Q;             /* Number of planes to read */

    bytes = gfs_read_nested(kid, a, offset, (Q * ELT_SIZE), vec, 2);
    return (bytes == (Q*Q*Q * ELT_SIZE));
}
```

Figure 5.5: Example of a nested-strided request. For simplicity, we assume that the data can be distributed evenly across the processors..

nested-batched interface. The data structure involved in a nested-batched I/O request is called a *request vector*, and is shown in Figure 5.6.

A single instance of this data structure essentially represents a single level in a nested-strided request. That is, with one `gfs_batch` structure, you can represent a "standard" request, a simple-strided request, or one level of nesting in a nested-strided request. Galley's batched interface allows an application to submit a vector of batched requests, which allows an application to submit a list of strided requests, a list of standard requests, a list of nested-strided requests, or arbitrarily complex combinations of those requests.

```
struct gfs_batch {
      int32 f_off;           /* File offset */
      int32 m_off;           /* Memory offset */
      char  f_absolute;      /* Is the file offset absolute? */
      char  m_absolute;      /* Is the memory offset absolute? */
      char  sub_vector;      /* Is the sub-request a vector? */
      int32 quant;           /* Number of repetitions */
      int32 f_stride;        /* File stride between repetitions */
      int32 m_stride;        /* Memory stride between repetitions */
      int   subvec_len;      /* Number of elements in subvec */
      union {
            int32 size;                /* Size for simple request */
            struct gfs_batch *subvec;  /* Vector of batch requests */
      } sub;
};
```

Figure 5.6: Data structure involved in a nested-batched I/O request.

As with a nested-strided request, a batched request allows an application to specify that a particular pattern will be repeated a number of times, with a regular stride between each instance of the pattern. However, a nested-strided request requires that the repeated pattern be either a simple-strided or a nested-strided request. The batched interface allows applications to repeat batched requests with a regular stride between them. Hence the name "nested-batched". This capability allows applications to repeat arbitrary access patterns with a regular stride.

A full `gfs_read_batched()` or `gfs_write_batched()` request will typically combine multiple `gfs_batch` structures into vectors, trees, vectors of trees, trees of vectors, and so on. For example, a doubly-nested-strided request would be a two-level tree. The root of the tree would describe the outer level of striding, and that node's child would describe the inner level of striding. An application with two such strided requests could combine them into a single batched request. In that case, there would be a vector of two trees, and each tree would have two levels.

The first two elements in the data structure contain the initial file and memory offsets of the request. The second two elements of the data structure indicate whether these offsets are specified absolutely (as is done with all other Galley requests), or relatively. If the offsets are relative, then if the request is the first element in a new vector, these offsets are specified relative to the offset of that vector's parent. Otherwise, a relative offset is specified relative to the offset of the previous element in the vector.

The fifth element in the structure (`sub_vector`) indicates whether the pattern to be repeated

is a simple data request or another batch vector. The sixth element (`quant`) indicates how many times the pattern should be repeated. The next two elements contain the strides that should be applied to the file and memory offsets between repetitions of the pattern. The ninth element in the structure applies only when the pattern to be repeated is a batched request. In that case, it indicates how many elements are in the sub-request.

Finally, the sub-request is described. The sub-request can be a simple data transfer (in the case of a standard or a simple-strided request), or it can be a vector of `gfs_batch` structures (in the case of a nested-strided, or more complex request).

List requests

```
int gfs_read_listio(int fid, void *buf, struct gfs_list *vec,
                    int quant)
```

Finally, in addition to these structured operations, Galley provides a *list* interface, which has functionality similar to the POSIX `lio_listio()` interface [IBM94]. This interface allows an application to simply specify an array of (file offset, memory offset, size) triples that it would like transferred between memory and disk. Although the nested-batched interface may be used to achieve the same functionality, the list interface is far simpler for those applications with access patterns that do not have any inherently regular structure. While this interface essentially functions as a series of simple reads or writes, it provides the file system with enough information to make intelligent disk-scheduling decisions, as well as the ability to coalesce many small pieces of data into larger messages for transferring between CPs and IOPs.

5.5.4 Non-blocking I/O

All of the calls discussed above are blocking calls. Galley also allows applications to make non-blocking data-transfer calls. The syntax of the non-blocking calls is similar to that of the blocking calls, with the addition of one extra parameter: a *handle*.

A handle is essentially a pointer to an internal Galley data structure. Each non-blocking call has an associated handle, and each handle may only be associated with a single non-blocking call at a time.

Galley uses the handle data structure to maintain information about the current status of the outstanding I/O request. Applications pass the handle to Galley to check the status of the request.

In particular, `gfs_test(handle)` returns 1 if the non-blocking call has completed and 0 if there is still data to be transferred. After calling `gfs_wait(handle)`, the application will block until the non-blocking I/O request has completed. `gfs_wait()` returns with the exit status of the non-blocking request (i.e., either the total number of bytes transferred or an error code). `gfs_wait()` is also used to *clear* a handle. That is, a handle may be reused, but the application must first inform Galley (using `gfs_wait()`) that it has collected the result of the earlier call associated with the handle. Calling `gfs_test()` does not clear the handle, even if the call has completed, because the application may wish to check how much data was transferred – information not returned by `gfs_test()`.

As with the blocking calls, Galley makes no guarantees about the order in which non-blocking requests will be satisfied. Thus, if a node issues multiple non-blocking reads for the same location in memory, the state of that location when the reads have completed is undefined. Similarly, if a node issues two non-blocking writes for overlapping regions of a single fork, the contents of that region when the writes complete are also undefined.

5.6 Portability

Galley was originally implemented on a cluster of IBM RS/6000s and an IBM SP-2, all running AIX 4.1.3. It has been in daily use on those systems for over a year, mostly by graduate and undergraduate students. Galley has been ported to DEC Alpha workstations running Digital Unix and x86-based PCs running Linux and FreeBSD.

The first port, from the RS/6000 to the DEC Alpha, took several days. Most of the difficulties we encountered during this port were related to the 32-bit nature of the RS/6000 and the 64-bit nature of the Alpha. After the first port was completed, the subsequent ports took only a few hours each. The ease of the subsequent ports suggests that most of the machine-dependent elements of the implementation have been identified and have either been removed or clearly noted in the code.

The portion of the code that requires the most change when porting to a new architecture is that part which implements Galley's thread interface. Although all of our target systems have ostensibly provided a POSIX-compliant *pthreads* interface, those interfaces are all based on different drafts of the POSIX standard. Thus, while the interfaces are basically the same, there are many

small incompatibilities. Rather than relying directly on the pthreads interface throughout the code, Galley implements its own simple interface to the underlying thread system. So, porting Galley to a new thread interface only requires that a small, isolated section of the code be modified.

Chapter 6

Performance of Galley

Most studies of multiprocessor file systems have focused primarily on the systems' performance on large, sequential requests. Indeed, most do not even examine the performance of requests of fewer than many kilobytes [Nit92, BBH95, KR94]. As discussed earlier, multiprocessor file-system workloads frequently include many small requests. This disparity between the observed and benchmarked workloads means that most performance studies actually fail to examine how a file system can be expected to perform when running real applications in a production environment.

6.1 Access Patterns

We examine the performance of Galley under several different access patterns, shown in Figure 6.1. Each pattern is composed of a series of requests for fixed-size pieces of data, or records. Although these patterns do not directly correspond to a particular 'real world' application, they are representative of the general patterns we observed to be most common in production multiprocessor systems, as described earlier. Our experiments used a file that contained a subfile on each IOP, and a single fork within each subfile. To allow us to better understand the system's performance, each fork was laid out contiguously on disk. The patterns shown in Figure 6.1 reflect the patterns that we access *from each fork*, and hence, from each IOP. The correspondence between the file-level patterns observed in actual applications, and the IOP-level access patterns used in this study, is discussed below.

The simplest access pattern is called *broadcast*. With this access pattern every compute node reads the whole file. In other words, the IOPs *broadcast* the whole file to all the CPs. This access

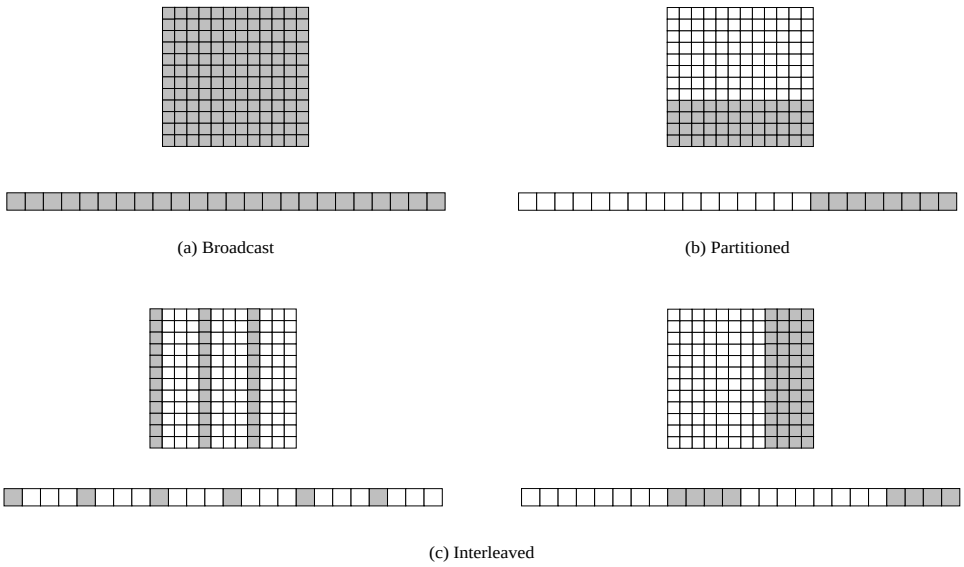

Figure 6.1: The three access patterns examined in this study. Two views of each pattern are displayed: the pattern as applied to a linear fork on a single IOP, and matrix distributions that could give rise to the pattern. For these examples, we assume that the matrices are stored in row-major order. Each square corresponds to a single *record* in the file, and the highlighted squares represent the records accessed by a single CP in a group of four.

pattern models the series of requests we would expect to see when all the nodes in an application read a shared file, such as the initial state for a simulation. Since, to read all the data in a file, an application must read all the data in every subfile, a broadcast pattern at the file level clearly corresponds to a broadcast pattern at each subfile. Although it may seem counterintuitive for an application to access large, contiguous regions of a file in small chunks, we observed such behavior in practice (recall the files with interval-size 0 in Chapter 3). One likely reason that data would be accessed in this fashion is that records stored contiguously on disk are to be stored non-contiguously in memory. Another possible cause for such behavior is that the I/O was added to an existing loop as an afterthought. Since it seems unlikely that an application would want every node to rewrite the entire file, we did not measure the performance of the broadcast-write case.

Under a *partitioned* pattern, each compute node accesses a distinct, contiguous region of each file. This pattern could represent either a one-dimensional partitioning of data or the series of accesses we would expect to see if a two-dimensional matrix were stored on disk in row-major order, and the application distributed the rows of the matrix across the compute nodes in a BLOCK

fashion. There are two different ways a partitioned access pattern at the file level can map onto access patterns at the IOP level. The simpler mapping, which is not shown in the figure, occurs if the file is distributed across the disks in a BLOCK fashion; that is the first $1/n$ of the file bytes in the file are mapped onto the first of the n IOPs, and so forth. For each IOP, this mapping results in an access pattern similar to a broadcast pattern with 1 compute processor. The other mapping, shown in the figure above, distributes blocks of data across the disks in a CYCLIC fashion. This second mapping is more interesting and corresponds to the mapping used by most implementations of a linear file model. This distribution results in accesses by each CP to each IOP. In a system with 4 CPs, the first CP would access the first 1/4 of the data in each subfile, and so forth. Thus, using the second mapping, a partitioned pattern at the file level leads to a partitioned pattern at each IOP. As with the broadcast pattern, applications may access data in this pattern using a small record size if the the data is to be stored non-contiguously in memory.

In an *interleaved* pattern, each compute node requests a series of noncontiguous, but regularly spaced, records from a file. For the results presented here, the interleaving was based on the record size. That is, if 16 compute nodes were reading a fork with a record size of 512 bytes, each node would read 512 bytes and then advance its index into the file by 8192 (16*512) bytes before reading the next chunk of data. The pattern models the accesses generated by an application that distributes the columns of a two-dimensional matrix across the processors in an application, in a CYCLIC fashion. To see how this file-level pattern maps onto an IOP-level pattern, assume that the linear file is distributed traditionally, with blocks distributed across the subfiles in a CYCLIC fashion. In the simplest case, the block size might be evenly divisible by the product of the record size and the number of CPs. In this case, every block in the file is accessed with the same interleaved pattern, and any rearrangement of the blocks (between or within disks) will result in the same subfile-access pattern. Thus, the blocks can be declustered across the subfiles, but the access pattern within each subfile will still be interleaved. There are, of course, more complex mappings of an interleaved file-level pattern to an IOP-level pattern, but we focus on the simplest case.

6.2 Experimental Platform

The Galley Parallel File System was designed to be easily ported to a variety of workstation clusters and massively parallel processors. The results presented here were obtained on the IBM SP-2 at NASA Ames' Numerical Aerodynamic Simulation facility. This system had 160 nodes, each running AIX 4.1.3, but only 140 were available for general use. Each node had a 66.7 MhZ POWER2 processor and at least 128 megabytes of memory. Each node was connected to both an Ethernet and IBM's high-performance switch. Although the switch allowed throughput of up to 34 MB/s using one of IBM's message-passing libraries (PVMe, MPL, or MPI), those libraries cannot operate in a multi-threaded environment. Furthermore, neither MPL nor MPI allow applications to be implemented as persistent servers and transient clients. As a result of these limitations, and to improve portability, Galley was implemented on top of TCP/IP.

6.2.1 TCP/IP Performance

To determine what effect, if any, our use of TCP/IP would have on the overall performance of our system, we benchmarked the SP-2's TCP/IP performance. According to IBM, and verified by our own testing, the maximum TCP/IP throughput between two nodes on the SP-2 is approximately 17 MB/s. Unfortunately, as the number of communicating nodes increases, they are unable to maintain this throughput at each node, as shown in Figure 6.2.

For each test shown in that figure, we used 16 *sinks*, and varied the number of *sources* from 4 to 64. For a given test, each source sent the same amount of data to each sink, in a series of messages, using a fixed record size. For each sink/source configuration, we measured the throughput for a variety of message sizes. As the throughput ranged over several orders of magnitude, we varied the total amount of data transferred as well, from 1.5 MB with 4 sources and a 64-byte record size, to over 800 MB with 64 sources and a 64-kilobyte record size.

In each of these tests, we used `select()` to identify sockets with pending I/O, but we did not attempt to use any flow-control beyond that provided by TCP/IP. As the figure shows, the achieved maximum throughput increases with the number of sources, until the number of sources exceeds 32. Even with many sources, we are only able to achieve about 220 MB/s, or less than 14 MB/s at each sink.

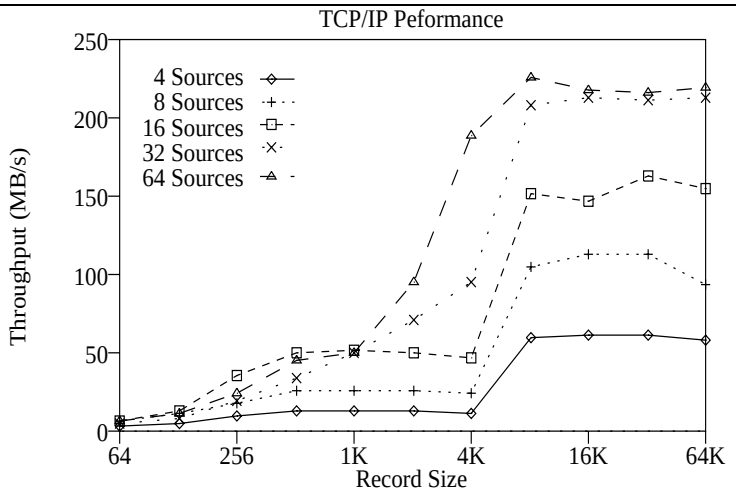

Figure 6.2: Measured TCP/IP throughout on the SP-2. For each test, there were 16 *sinks* (similar to CPs reading a file), and a variable number of *sources* (similar to IOPs servicing read requests).

6.2.2 Simulated Disk

Each IOP in Galley controls a single disk, logically partitioned into 32KB blocks. For this study, each IOP had a buffer cache of 24 megabytes, large enough to hold 750 blocks. While each node on the SP-2 has a local disk, that disk must be accessed through AIX's Journaling File System (JFS). Although Galley was originally implemented to use these disks, we found that the performance results obtained using those disks were unreliable, primarily due to the prefetching and caching performed by JFS. Specifically, we frequently measured apparent throughputs of over 10 MB/s from a single disk. Furthermore, as we discuss below, the choice of access pattern could greatly affect Galley's disk scheduling, which in turn affected performance by changing the amount of time the disk spent seeking and how effectively Galley was able to use the disks' on-board caches. Accessing disks through JFS caused these differences to be minimized or eliminated, concealing the impact an application's access pattern would have on Galley's performance. To ensure we were actually evaluating the performance of Galley, rather than that of AIX's prefetching and caching implementations, for this study we used a simulation of an HP 97560 SCSI hard disk rather than the physical disks on each node. The HP 97560 has an average seek time of 13.5 ms and a maximum sustained throughput of 2.2 MB/s [HP91].

Our implementation of the disk model was based on earlier implementations described in [RW94,

KTR94]. Among the factors simulated by our model are head-switch time, track-switch time, SCSI-bus overhead, controller overhead, rotational latency, and the disk cache. To validate our model, we used a trace-driven simulation, using data provided by Hewlett-Packard and used by Ruemmler and Wilkes in their study.[1] To evaluate the accuracy of our model, we used the same metric as the previous two implementations, This metric, which Ruemmler and Wilkes call the *demerit figure*, is obtained by plotting the cumulative time distribution curves of the real and simulated disk outputs, and calculating the root mean square of the horizontal distance between the two curves. Comparing the results of our trace-driven simulation with the measured results from the actual disk, we obtained a demerit figure of 5.0%, indicating that our model was extremely accurate.

The simulated disk is integrated into Galley by creating a new thread on each IOP to execute the simulation. When the thread receives a disk request, it calculates the time required to complete the request, and then suspends itself for that length of time. In most cases the disk thread does not actually load or store the requested data, but since metadata blocks must be preserved, the disk thread maintains a small pool of buffers, which is used to store this metadata. When the disk simulation thread copies data to or from a buffer, the amount of time required to complete the copy (which we calculate at system startup) is deducted from the amount of time the thread is suspended. It should be noted that the remainder of the Galley code is unaware that it is accessing a simulated disk. It should also be noted that this pool of buffers was small enough (ten 32 KB blocks on each IOP) that we are confident that it remained memory-resident, so this small cache was not likely to affect the node's virtual memory behavior.

6.3 Performance Results

For this performance analysis, we held the number of compute processors constant at 16, and varied the number of IOPs (each with one disk) from 4 to 64. Thus, the CP:IOP ratio varied from 1:4 to 4:1. Each test began with an empty buffer cache on each IOP, and each write test included the time required for all the data to actually be written to disk. Although the size of each fork was fixed, the amount of data accessed for each test was not. Since the system's performance on the fastest tests was several orders of magnitude faster than on the slowest tests, there was no fixed amount of data that would provide useful results across all tests. Thus, the amount of data accessed for each

[1] Kindly provided to us by John Wilkes and HP.

test varied from 4 megabytes (writing 64-byte records to 4 IOPs) to 2 gigabytes (reading 64-KB records from 64 IOPs). We performed each test 5 times. Since other applications were running on the machine at the same time, there was some variability in the results. In an attempt to identify the system's typical performance on each pattern, we disregarded the lowest and highest results, and present the average of the remaining 3.

6.3.1 PIOFS

Our first workload study revealing the prevalence of small requests in multiprocessor workloads was published in 1994. To determine whether or not parallel file system performance on these small requests had not improved since our findings came to light, we examined the performance of IBM's PIOFS, one of the newest commercial parallel file systems, on the patterns discussed above.

We measured the performance of PIOFS (using its Unix interface rather than its lower-level Vesta interface) on the SP-2 described above. There were 3 I/O nodes, each of which used a 4-disk RAID 0 for its underlying storage medium. We measured the performance using 16 CPs. The results of these tests are shown in Figures 6.3 and 6.4. These plots show that that even the newest parallel file systems still provide poor performance for the requests we observed to be common in practice.

Although these results were obtained on the same machine as the performance results for Galley (discussed below), the two filesystems used very different kinds of disks. Furthermore PIOFS had access to the user-level network interface, which allowed communication at up to 35 MB/s. Finally, we were unable to force the IOP's caches to be flushed between tests, so it is likely that all the read tests indicate PIOFS's performance when reading from cache. Given these differences, we cannot directly compare the performance of PIOFS with the performance of Galley. These results are included simply to show that with the standard Unix-like interface, even the newest parallel file systems do not deliver high performance for the small requests that dominate parallel scientific workloads.

6.3.2 Traditional Interface

We first examined the performance of Galley using the standard read/write interface. This interface required each CP to issue separate requests for each record from each fork. Each CP issued

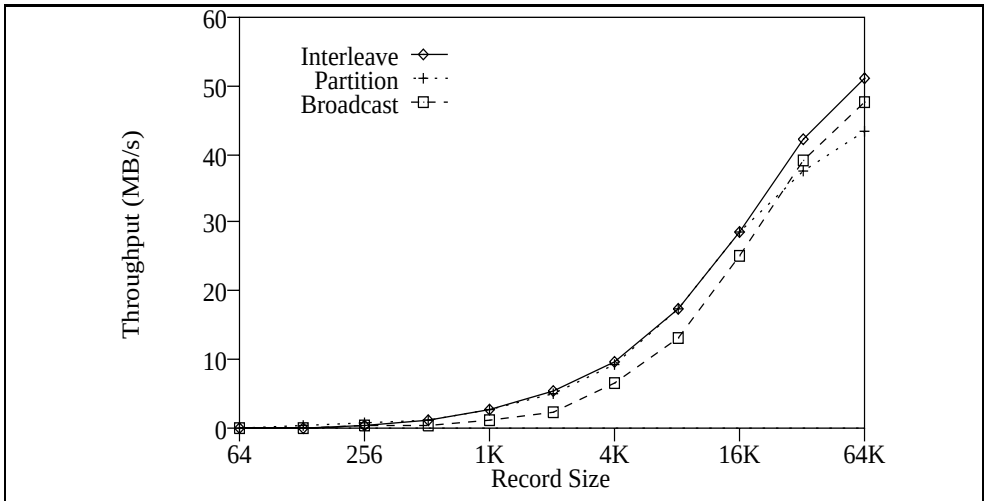

Figure 6.3: Read performance of PIOFS on the SP-2. There were 16 CPs and 3 PIOFS servers.

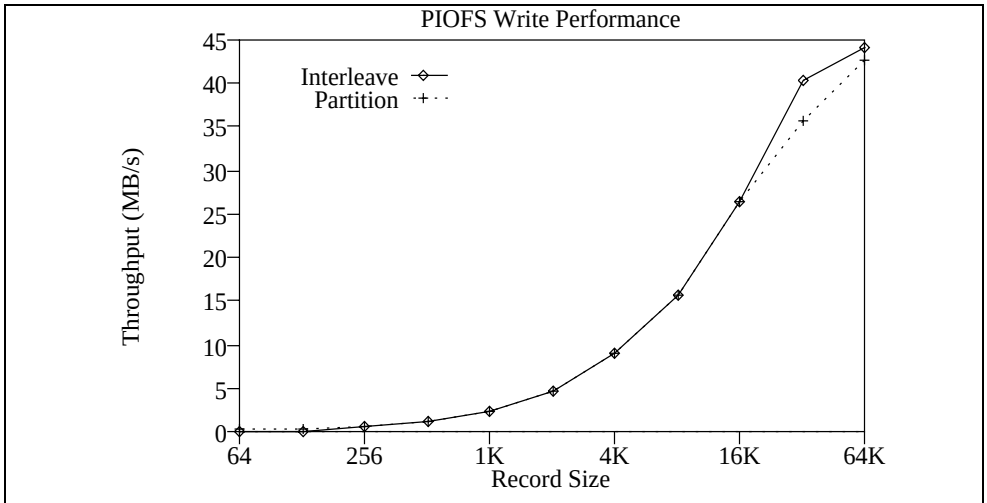

Figure 6.4: Write performance of PIOFS on the SP-2. There were 16 CPs and 3 PIOFS servers.

asynchronous requests to all the forks, for a single record from each fork. When a request from one fork completed, a request for the next record from that fork was issued. By issuing asynchronous requests to all IOPs simultaneously, the CPs were generally able to keep all the IOPs in the system busy. Since each CP accessed its portion of each subfile sequentially, the IOPs were frequently able to schedule disk accesses effectively, even with the small amount of information offered by the traditional interface. Furthermore, the CPs were generally able to issue requests in phase. That is, when an IOP completed a request for CP 1, it would handle requests for CPs 2 through n. By the time the IOP had completed the request from CP n, it had received the next request from CP 1. Thus, even without explicit synchronization among the CPs, the IOPs were frequently able to service requests from each node fairly and were able to make good use of the disk.

Figure 6.5 shows the total throughput achieved when reading a file with various record sizes for each access pattern. Figure 6.6 presents similar results for write performance when overwriting an existing file, and Figure 6.7 shows Galley's performance when writing to a new file. The performance curves have the same general shape as throughput curves in most systems; that is, as the record size increased, so did the performance. As in most systems, eventually a plateau was reached, and further increases in the record size did not result in further performance increases. The precise location of this plateau varied between patterns and CP:IOP ratios. Not surprisingly, when accessing data in small pieces, the total throughput was limited by a combination of software overhead and by the high latency of transferring data across a network, regardless of the access pattern.

The choice of access pattern had the greatest effect on performance when reading data with large blocks. When reading an interleaved pattern, the system's peak performance was limited by the sustainable throughput of the disks on each IOP (about 2.2 MB/s). Interestingly, there was a small dip in performance as the record size increased from 2 KB to 4 KB. With records of 2 KB or smaller, every CP reads data from every block. So, regardless of the order in which CPs' requests arrive at an IOP, that IOP reads all of the blocks in its fork, in order. With a record size of 4 KB, each CP reads data only from alternate blocks. As a result, it is possible for a request for block $n+1$ to arrive before a request for block n, possibly causing a miss in the disk cache and an extra head seek, slightly degrading disk performance. The overall performance when reading the partitioned pattern was limited by the time the disk spent seeking from one region of the file to another.

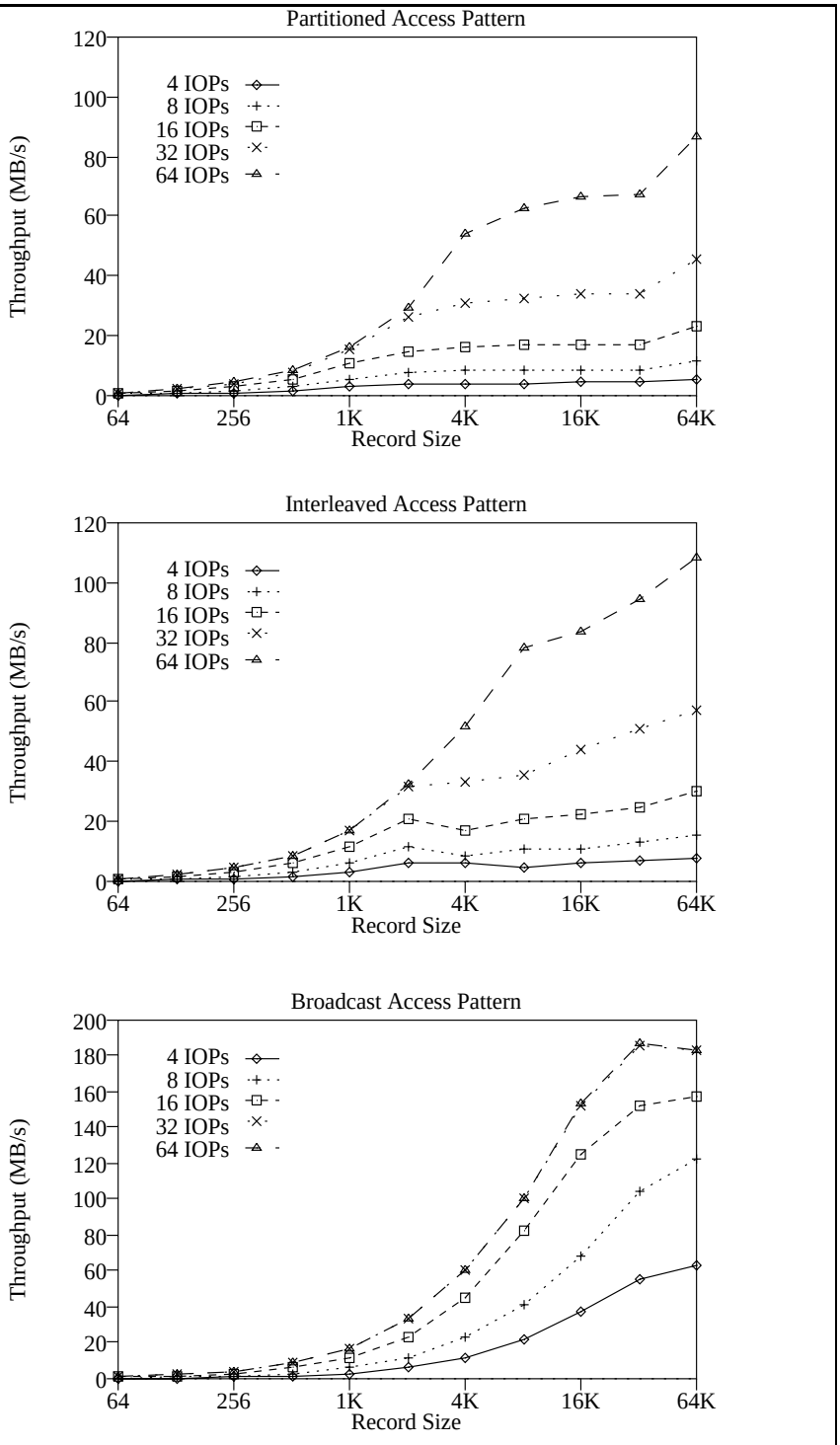

Figure 6.5: Throughput for **read** requests using the traditional Unix-like interface. There were 16 CPs in every case. Note the different scales on the y-axis.

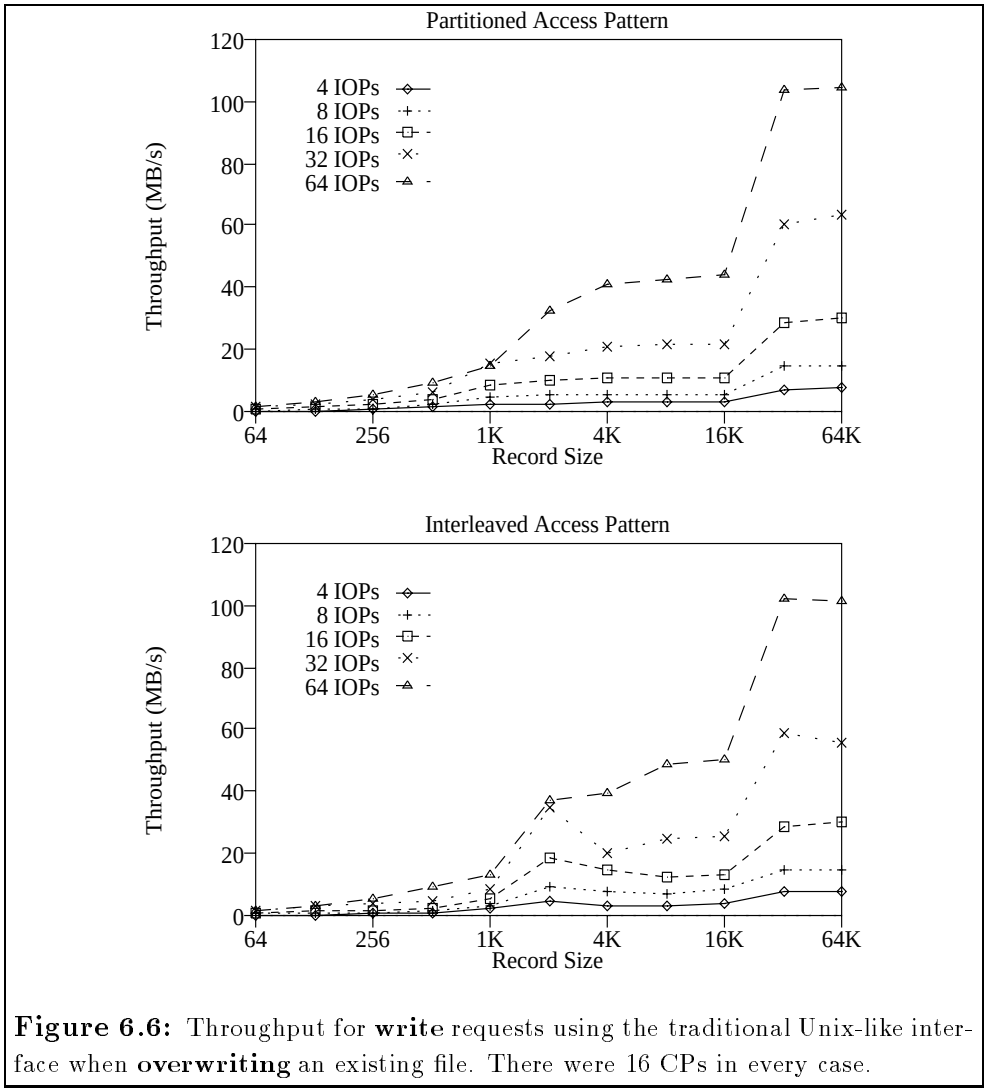

Figure 6.6: Throughput for **write** requests using the traditional Unix-like interface when **overwriting** an existing file. There were 16 CPs in every case.

When testing an earlier version of Galley we found that with large numbers of IOPs, the network congestion at the CPs was so great that the CPs were unable to receive data and issue new requests to the IOPs in a timely fashion [NK96]. This congestion was also responsible for the limited TCP/IP throughput with large numbers of nodes. As a result, the in-phase ordering of requests discussed above broke down, so the DiskManagers on the IOPs were unable to make intelligent disk scheduling decisions, causing excess disk-head seeks and thrashing of the on-disk cache. The combination of the network congestion and the poor disk scheduling led to dramatically reduced performance with large record sizes in the interleaved and partitioned patterns. To avoid this problem, we added a simple flow-control protocol to Galley's data-transfer mechanism. This flow control essentially

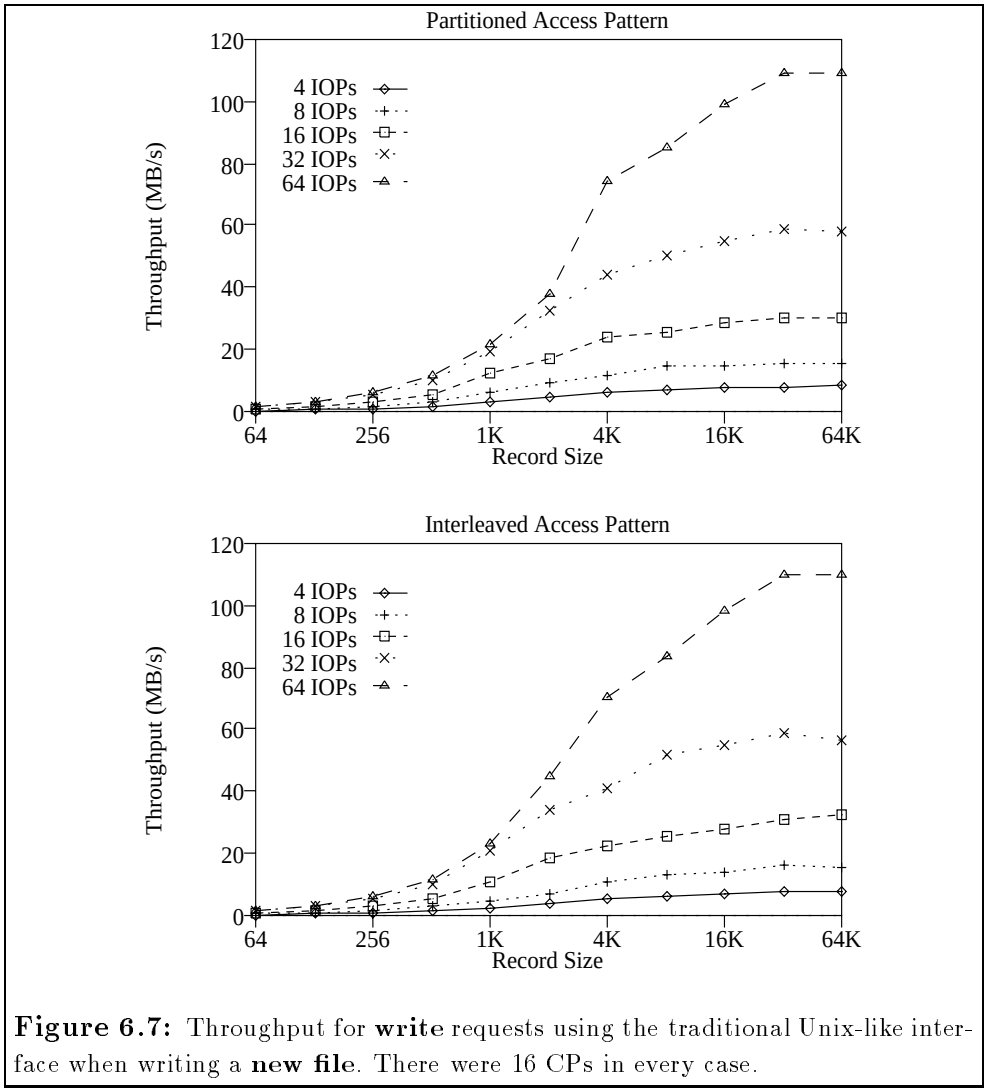

Figure 6.7: Throughput for **write** requests using the traditional Unix-like interface when writing a **new file**. There were 16 CPs in every case.

requires an IOP to obtain permission from a CP before sending each chunk of data. By limiting the number of outstanding permissions, the CP can reduce or avoid this network congestion. Simple experiments on the SP-2 showed that choosing a limit between 2 and 8 led to the highest, and most consistent, performance. While this limit is currently a compile-time option, it may be worth exploring the possibility of allowing the CP to set it dynamically as well.

Under the broadcast access pattern, data was read from the disk once, when the first compute processor requested it, and stored in the IOP's cache. When subsequent CPs requested the same data, it was retrieved from the cache rather than from the disk. Since each piece of data was used many times, the cost of accessing the disk was amortized over a number of requests, and the limiting

factors were software and network overhead. In this case, the total throughput of the system was limited by the SP-2's TCP/IP performance, as discussed above.

We now consider Figure 6.6. When overwriting an existing file, and using records of less than 32 KB, the file system had to read each block off the disk before the new data could be copied into it. Without this requirement, any data that was stored in that block would be lost — even data that was not being modified by the write request. As a result, the system's performance was significantly slower when writing small records than when reading them. As when reading data, the interleaved pattern had the higher throughput because the partitioned pattern forced the disk to spend time seeking between one region of the file and another. The performance difference between the two was smaller when writing since many of the disk accesses in the write case occurred at the end of the test, when the benchmark forced each IOP to write all dirty blocks to disk (with a `gfs_sync()` call). Since most of the disk accesses occurred at once, the DiskManager was able to schedule those accesses efficiently.

When the record size reached 32 KB, the write performance of both patterns increased dramatically. With the record size at least as large as the file system's block size, Galley did not have to read each data block off the disk before copying the new data in. Since the file system could simply write the new data to disk (rather than read-modify-write), the number of disk accesses in each pattern was cut in half.

We finally consider Figure 6.7. In these tests we measured the time to write data to a new file, rather than overwriting an existing file. We did not use Galley's `gfs_extend()` call (which preallocates disk space for a fork) for these tests; new blocks were assigned to the fork on the fly, as it grew. Not only was writing to a new file generally faster than overwriting an existing file, in many cases it was even faster than reading a file. For small requests, writing a new file was faster than overwriting an existing file because there was no need to read the original data off of disk. There is some additional overhead involved when writing a new file, as new blocks must be assigned to the file, but this cost was significantly less than the cost of the read-modify-write cycle. In those cases where writing a new file was faster than reading a file, the write tests benefited from the nearly perfect disk schedule during the `gfs_sync()` call, as discussed above.

6.3.3 Strided Interface

When reading data with a traditional interface, in many cases we were able to achieve nearly 100% of the disks' peak sustainable performance. This best-case performance seems respectable, but as with most systems, Galley's performance with small record sizes was certainly less than satisfactory. The goal of Galley's new interfaces is to provide high performance for the whole range of record sizes, with particular emphasis on providing high throughput for small records.

The tests in this section were again performed by issuing asynchronous requests to each fork. Rather than issuing a series of single-record requests to each IOP, we used the strided interface to issue only a single request to each IOP. That single request identified all the records that should be transferred to or from that IOP for the entire test. All other experimental conditions were identical to those in the previous section.

Figure 6.8 shows the total throughput achieved when reading a file with various record sizes for each access pattern using the new interface. Figure 6.9 shows corresponding results for overwriting an existing file and Figure 6.10 shows the results when writing to a new file.

Given the traditional interface, the disk scheduler had to handle each request in the order they arrived from the CPs. This requirement led to excess disk-head movement primarily in the partitioned pattern, but also in the interleaved pattern when the record size was larger than 2 KB (32 KB/16 CPs). Since each CP read from the same data blocks in the broadcast case, and in an interleaved pattern with small records, the disk schedule was optimal even with the traditional interface. Since many of the disk accesses in the traditional write cases occurred after a call to `gfs_sync()`, the disk scheduler was able to make intelligent decisions then as well. Therefore, the tests on which the new interface led to the greatest improvements in the disk schedule were the interleaved and partitioned read tests, and these were the two tests where the peak throughput to the CPs improved most dramatically. Note that the interleaved pattern again sees a significant performance increase when the record size reaches 32 KB. As with the traditional interface, with smaller records each block must be read before it is modified. A file system that offered a collective data-access interface would frequently be able to avoid the read portion of the read-modify-write cycle, as it would be aware that the entire block would be overwritten by data from all the processors.

Once again, network contention was a problem for large numbers of IOPs. The peak throughput

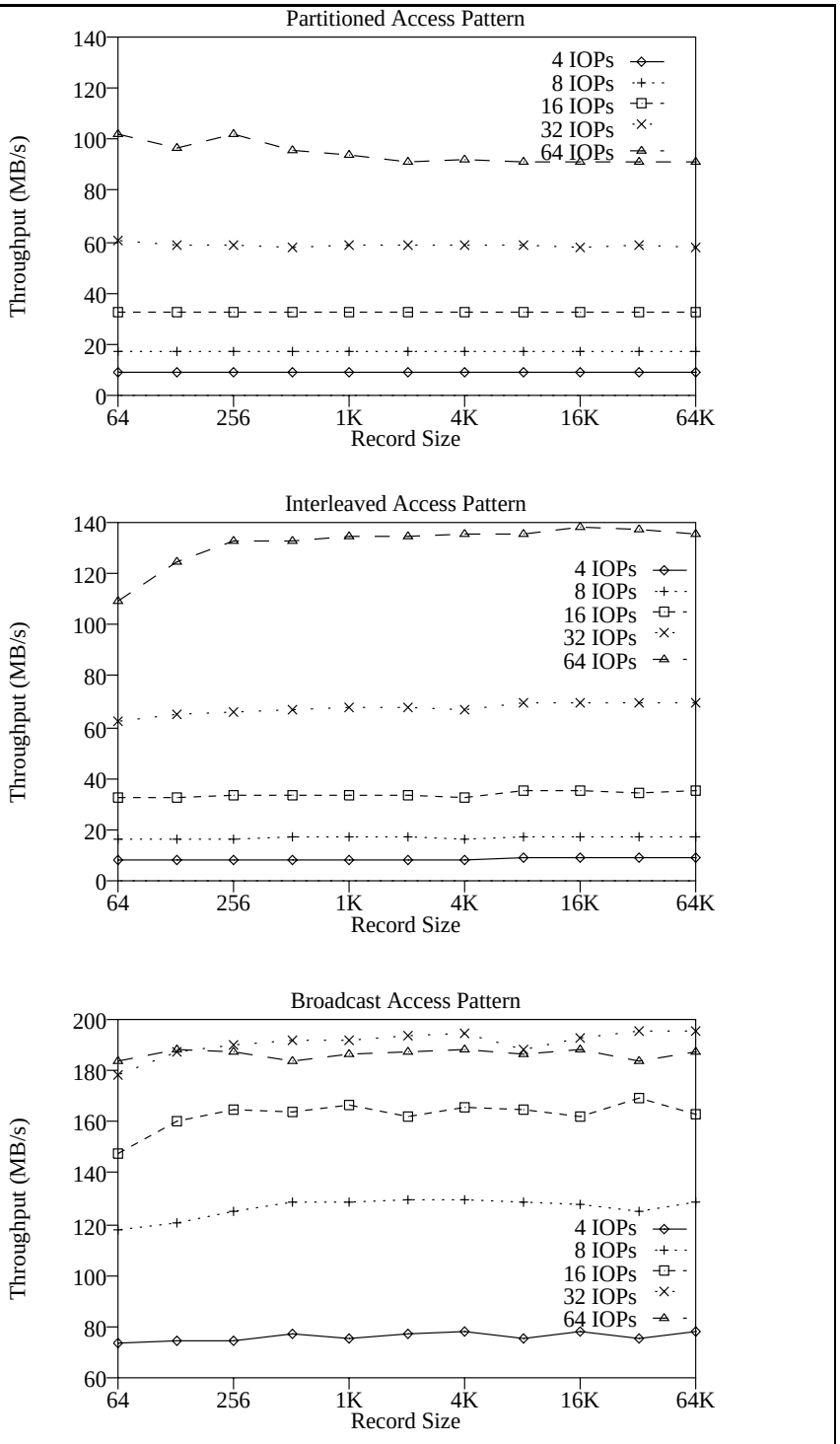

Figure 6.8: Throughput for **read** requests using the strided interface. There were 16 CPs in every case. Note the different scales on the y-axis.

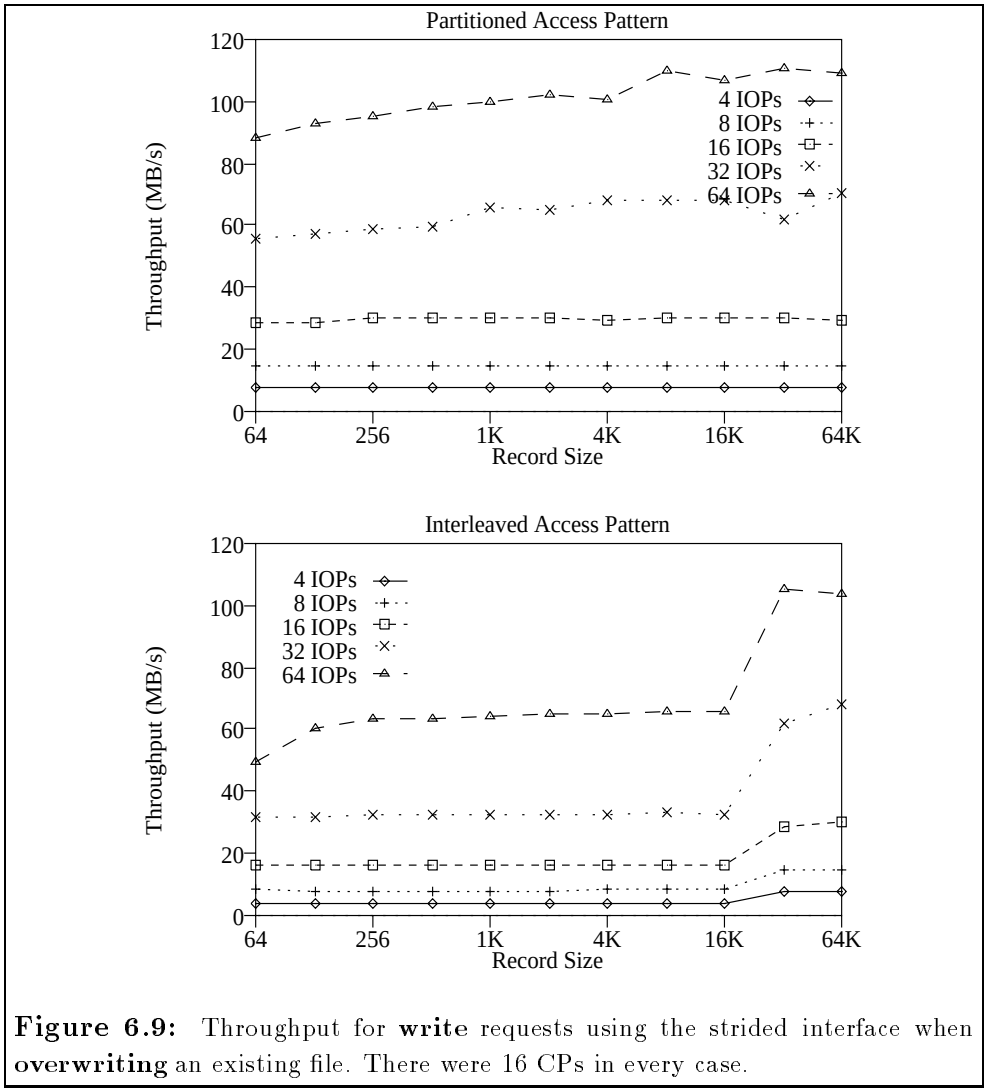

Figure 6.9: Throughput for **write** requests using the strided interface when **overwriting** an existing file. There were 16 CPs in every case.

on the broadcast pattern was limited to 13–14 MB/s to each CP. The best disk schedule can also be the worst network schedule, as in the partitioned pattern, where all IOPs first served CP 1, then CP 2, and so forth. This disk schedule, combined with the limits of TCP/IP on the SP-2, contributed to the interleaved-read pattern having higher performance than the partitioned-read pattern using the strided interface.

While the increase in peak performance is interesting, the most striking difference between the two sets of tests is that, in most cases, Galley was able to achieve peak performance with records as small as 64 bytes — two or three orders of magnitude smaller than the request sizes required to achieve peak throughput using the traditional interface. Other than increased opportunities for

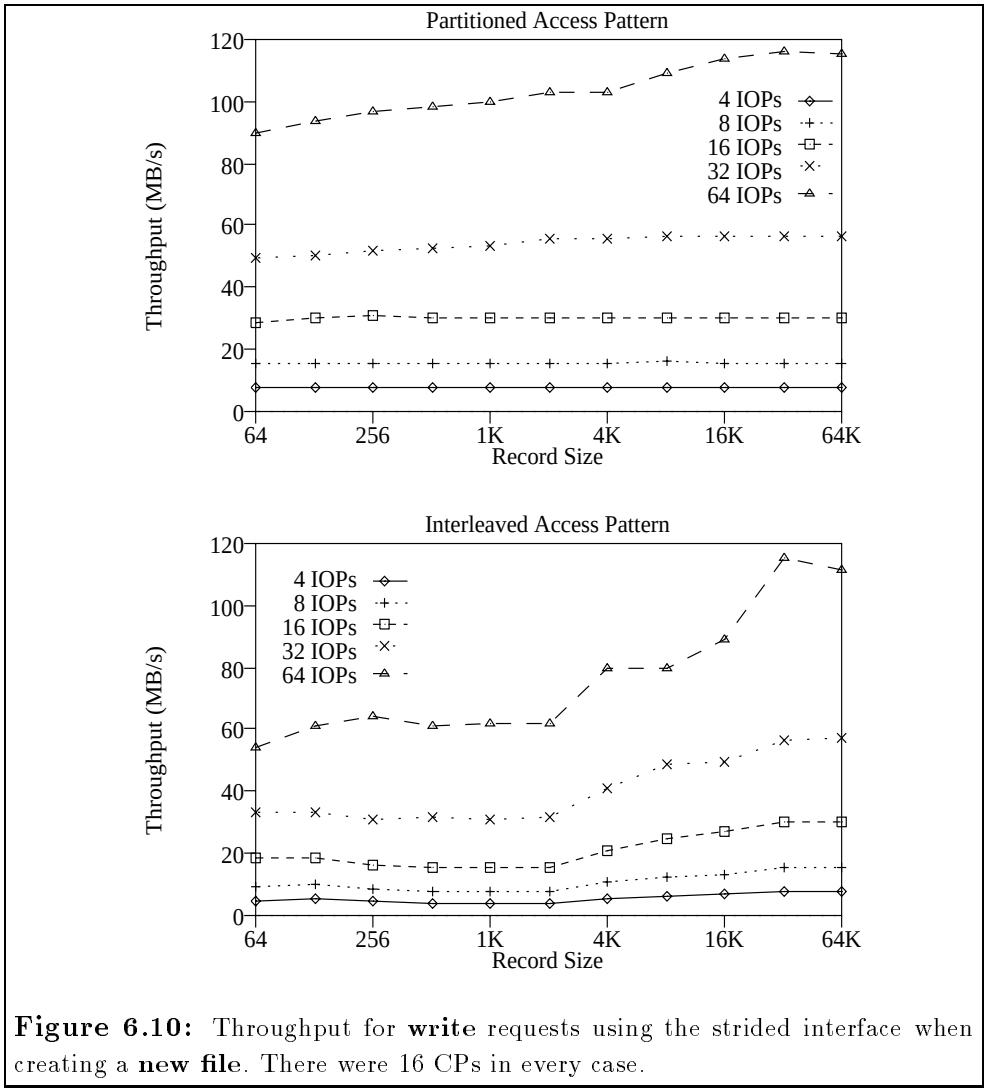

Figure 6.10: Throughput for **write** requests using the strided interface when creating a **new file**. There were 16 CPs in every case.

intelligent disk scheduling, the primary performance benefit of our new interface was a reduction in the number of messages, accomplished by packing small chunks of data into larger packets before transmitting them to the receiving node.

One interesting case where Galley was not able to achieve maximum throughput with a small record size was in writing a new file in an interleaved pattern. When a CP thread on an IOP receives the first request to write to a new fork, that CP thread *locks* the metadata for that fork. The CP thread then examines the list of requests for the fork, and asks the DiskManager to assign however many blocks are necessary for the new file. Only after all the blocks have been assigned does the CP thread *unlock* the fork's metadata, allowing the other CP threads to start processing

their requests. It appears that the delay caused by this long-term locking noticeably affects the system's throughput. This delay is less significant with the partitioned pattern because the number of requests is smaller; each CP has at most one request per block in the partitioned pattern, while they may have many requests per block in the interleaved case.

Although it is clear that the strided interface allowed the file system to deliver much better performance, the throughput plots shown in Figures 6.8–6.10 present only part of the picture. Figure 6.11 shows the speedup of the strided-read interface over a traditional read interface, and Figures 6.12 and 6.13 show similar results for the write interfaces, for both new files and overwriting preexisting files. When using an interleaved pattern with small records, the strided interface led to speedups of up to 98 times when reading, 30 times when overwriting an old file, and 23 times when writing a new file. There was a similar increase in performance for small records in a partitioned pattern: up to 92 times when reading, 56 times when rewriting, and 35 times when writing a new file. The broadcast-read pattern had the largest speedups for small records, ranging from 150 to over 350.

Although there was less room for improvement with large records, better disk scheduling when reading interleaved and partitioned patterns occasionally led to higher performance even for large records. When reading, the minimum speedups within the range of record sizes we examined were between 1.0 and 2.0, and occurred with the largest record sizes. When writing, the minimum speedups were mostly between 1.0 and 1.2, with one test (writing a new file in a partitioned pattern with 4 IOPs) as low as .93. Again, the minimum speedups in the write tests were smaller than the read tests because much of the writing with the traditional interface was performed during the `gfs_sync()` call, so the IOP was able to perform more efficient disk scheduling.

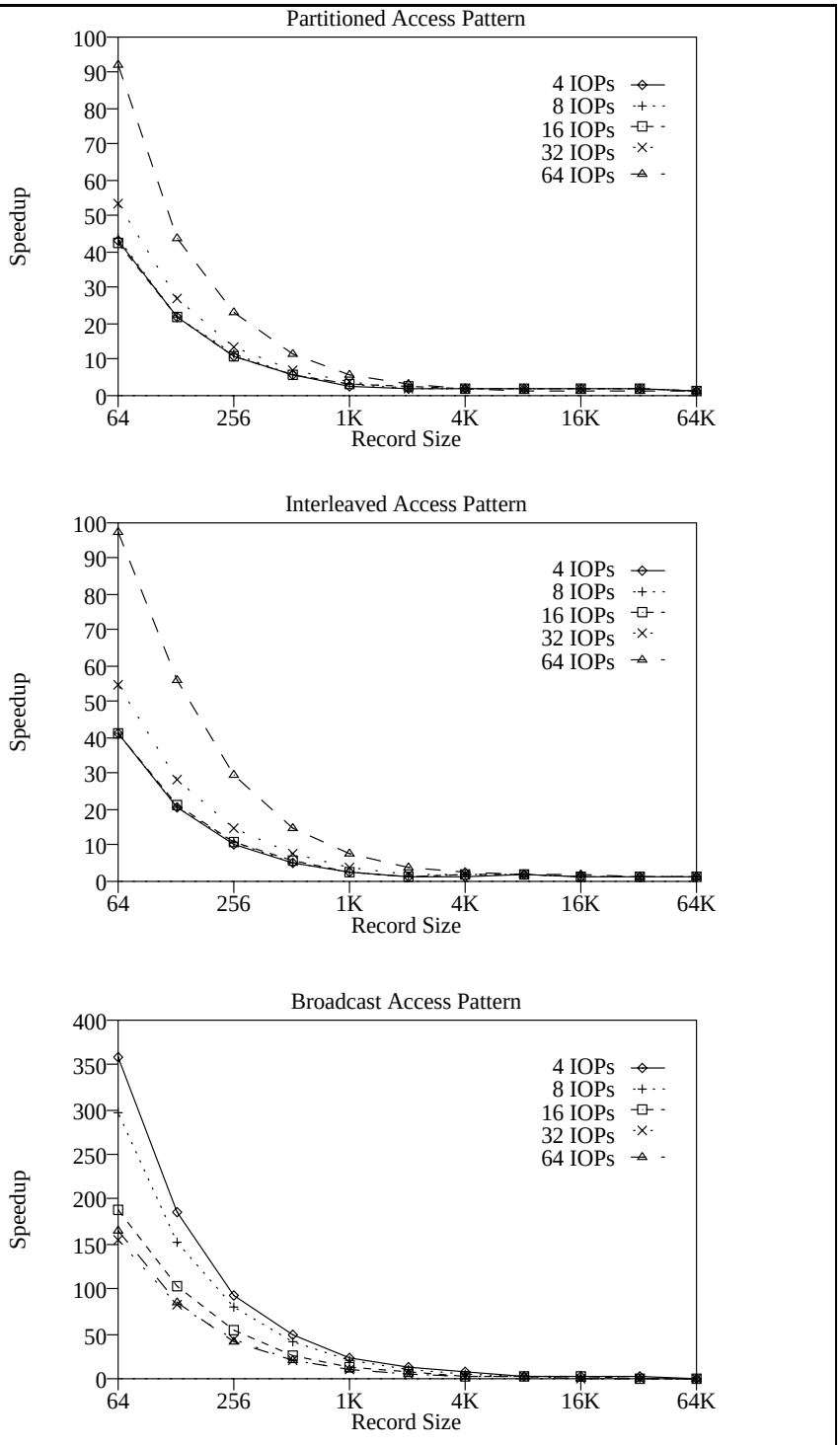

Figure 6.11: Increase in throughput for **read** requests using the strided interface. Note the different scales on the y-axis.

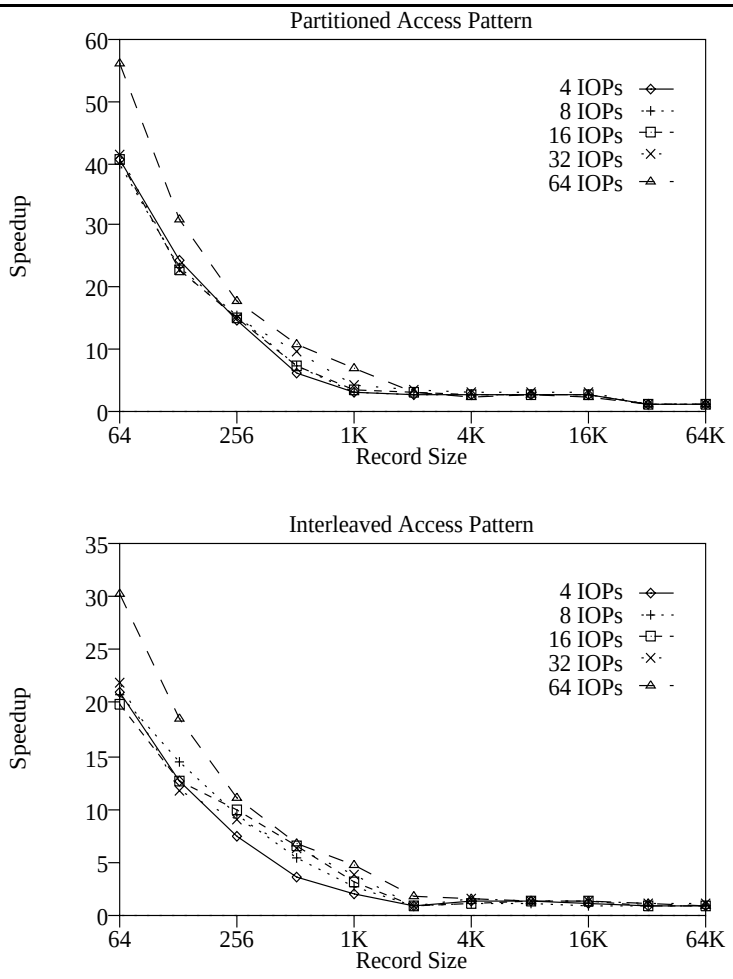

Figure 6.12: Increase in throughput for **write** requests using the strided interface when **overwriting** an existing file. Note the different scales on the y-axis.

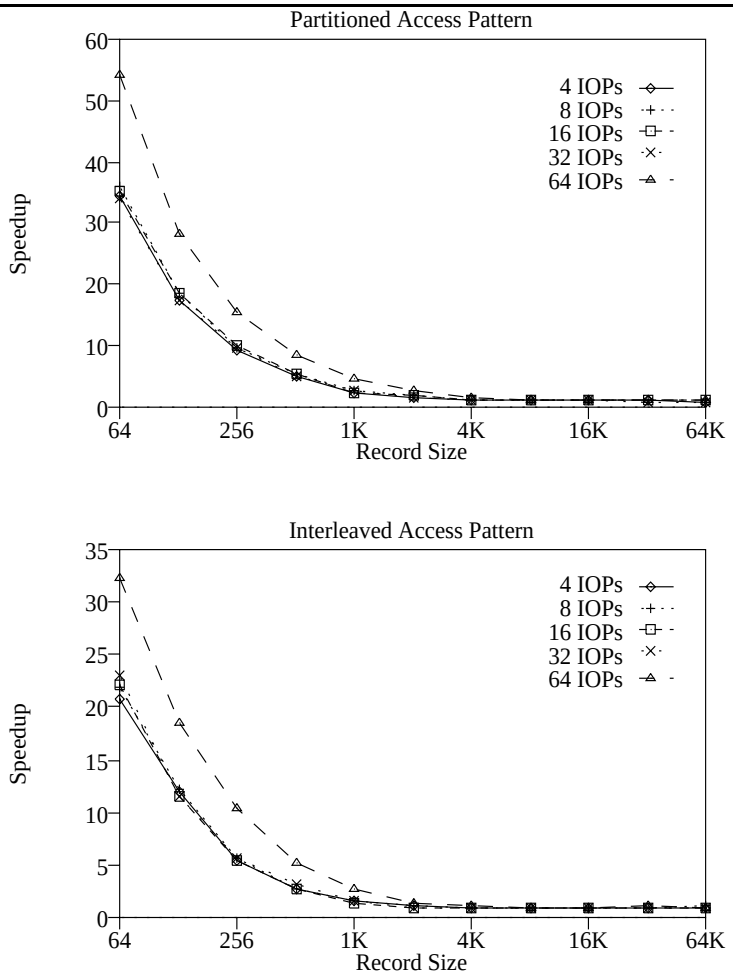

Figure 6.13: Increase in throughput for write requests using the strided interface when creating a **new file**. Note the different scales on the y-axis.

Chapter 7

Galley in Practice

We will now present several different examples, showing how Galley's features have been used in practice. We will discuss, in detail, an application written directly to Galley's API, a user-level library implemented on top of Galley, and an application implemented on top of the user-level library.

7.1 FITS

The Flexible Image Transport System (FITS) data format is a standard format for astronomical data [NAS94]. A FITS file begins with an ASCII header that describes the contents of the file and structure of the records in the file. The remainder of the file is a series of records, stored in binary form. Each record is composed of a key, with one or more fields, and one or more data elements. Each record within a single FITS file has an identical size and structure. Records may appear in any order within the file.

For this work, we created a system that was able to handle a specific type of FITS file in use at the National Radio Astronomy Observatory (NRAO), and generic queries on those files. A library that was capable of handling many kinds of queries and FITS files is a perfect example of the type of domain-specific library we expect to be implemented on Galley.

7.1.1 FITS at NRAO

One specific example of how FITS files are used in practice is described in [KFG94, KGF93]. This type of FITS file contains records with 6 keys, describing the frequency domain (U, V, W), the baseline, and the time the data was collected. The baseline is a single number that indicates

which antenna or combination of antennas generated that record. The data portion of each record contains a pair of data elements, one for each of two polarizations. Each data element contains floating-point triples for each of 31 frequencies. The triples represent a single complex number and a weighting factor. Thus, a single data element contains 372 bytes of data and each record contains 24 bytes of key information and 744 bytes of data.

These files are used in many different ways by different users at NRAO. The most common types of use involve scanning sub-volumes of the full, multi-dimensional, sparse data set, where the sub-volumes may be defined along one or more of the axes. For example, a user may want to examine all the records within a given time range, sorted along the U axis.

Previous work on these files has focused on increasing locality along several dimensions simultaneously. In [KFG94, KGF93], the authors examine studied the effectiveness of Piecewise Linear Order-Preserving Hashing (PLOP) files at reducing the amount of time required to perform common queries, by increasing certain kinds of locality within the files. While locality can also improve performance in parallel file systems, too much locality can reduce the number of disks being accessed at any time, actually leading to lower performance.

7.1.2 FITS on Galley

We now describe how we stored FITS files on Galley.

Since most of the queries common at NRAO include subranges of time as at least one of the constraints, we sorted the records by time before distributing them across the IOPs. The data was distributed in cyclic fashion, in blocks of 1024 records. That is, in a system with 4 IOPs, IOP 0 would hold records 0 to 1023, 4096 to 5119, and so on, while IOP 1 would hold records 1024 to 2047, 5120 to 6143, and so on.

For many queries, we were unable to determine *a priori* which data records would satisfy the query. As a result, we frequently examined many keys to identify the small number of data records that were relevant to the query. To improve performance, we chose to store the keys in one fork per IOP and the data in another. This setup allowed us to achieve higher performance when reading the keys, since we were not paying for the cost of retrieving uninteresting data from disk. Although we stored all the data in a single fork on each subfile, another reasonable choice would have been to store the data for each polarization in its own fork. Since many of the queries involved data

from only a single polarization, this setup would also have reduced the amount of uninteresting data that was read from disk.

To evaluate the efficacy of their PLOP-file implementation, the authors performed several queries, which were intended to be representative of those that were most commonly used in practice at NRAO [KGF93]. Their tests were performed on a single-processor, single-disk system. We performed the same set of queries, using the same data set, on our implementation. Our tests were performed on a cluster of IBM RS/6000s connected by an FDDI network. Since the original queries were performed on a single-node processor, we also used a single CP. We used four IOPs, each with a single disk. Each IOP used a raw disk partition to store its data, thus avoiding skewing the results by retrieving data stored in AIX's buffer cache.

The specific queries performed in both cases are briefly described below. More detail about each query, and why it is commonly used at NRAO, may be found in [KGF93].

1. Read the full data set.
2. Read the full data set, sorting records by time.
3. Read the full data set, sorting records by baseline.
4. Read a sub-volume of the data including 10% of the time range.
5. Read a sub-volume of the data including 10% of the time range, sorting the records by U.
6. Read the sub-volume for a single time and polarization.
7. Read a sub-volume including 10% of the time range and one polarization.
8. Read a sub-volume including 50% of the time range, a single baseline, and one polarization.
9. Read a sub-volume including 50% of the time range, antenna #1, and one polarization.
10. Read a sub-volume including 50% of the time range, antenna #14, and one polarization.
11. Read a sub-volume including 50% of the time range, antenna #27, and one polarization.
12. Read a sub-volume containing a single baseline and a single polarization, sorting records by time.

Using *a priori* knowledge about the structure of the queries and the on-disk data structures, many of these requests could have been most efficiently expressed using some form of strided request. Since our system was designed to handle generic queries, however, the queries were all performed using Galley's list interface. The buffer cache on each IOP was flushed prior to each query.

Table 7.1 shows the length of time required to complete each query for both the PLOP-file and Galley implementations. Since the PLOP-file results were obtained on a different system with only a single disk, we cannot directly compare the absolute time required to complete the queries. Instead, we compare the amount of time required to complete a query relative to the time required to read all the data. This crude normalization allows us to make some effort at comparison.

	Data	PLOP-file		Galley	
Query	Elements	Secs.	Normal.	Secs.	Normal.
1	126092	55.49	1.00	11.20	1.00
2	126092	70.50	1.27	11.72	1.05
3	126092	181.80	3.28	13.96	1.25
4	12636	4.10	0.07	1.00	0.09
5	12636	6.85	0.12	1.53	0.14
6	351	0.27	0.00	0.17	0.01
7	6318	2.33	0.04	0.62	0.05
8	186	1.45	0.03	0.55	0.05
9	4836	4.03	0.07	1.45	0.13
10	4836	14.50	0.26	2.10	0.19
11	4836	20.30	0.37	2.27	0.20
12	180	1.49	0.03	1.57	0.14

Table 7.1: Timing results for PLOP files on a uniprocessor system, and for Galley files on a 4-IOP, 1-CP system. Results are shown in raw form, as well as normalized to the time required to read the full data set with no filtering or sorting. The full data set contained 63,046 records, with 126,092 data elements.

While our implementation on top of Galley was far simpler than the PLOP-file implementation (only about 1/10 the number of lines of code), it performed significantly better in 4 out of 11 cases (disregarding the first case, which is used as a baseline), and had competitive performance in 5 of the remaining cases. Galley performed particularly well on queries 2 and 3. While the PLOP-file implementation had to sort the whole dataset in memory, Galley's interface allowed us to read just the keys from their forks, sort them, and then read the actual data into memory in sorted order. Galley also performed relatively well on queries 10 and 11. While the PLOP-file implementation

had to read in 3 to 5 times as many records as it needed, we were able to filter out the interesting records by looking only at the data in the key-fork. Galley's relative performance was worst on queries 9 and 12. In these two cases, Galley had to examine a large number of keys to identify a small number of interesting records, while the PLOP files were carefully structured to reduce the number of records they had to examine for these queries. This same structure also caused the PLOP-file implementation to be noticeably worse than Galley's on queries 10 and 11.

7.2 A Linear File Model

For decades, the Unix-like, linear file model has been the standard model for data storage on uniprocessor and vector supercomputer systems. That is, on most systems, files are collections of bytes, arranged in a one-dimensional structure. In view of this long-term trend, it is unsurprising that the designers of many parallel file systems adopted the same model.

To simplify the task of porting legacy applications and creating new applications, we have implemented LFM, a library that provides a linear file model on top of Galley. Furthermore, this linear file model provides applications with the interfaces necessary to achieve high performance on those common, yet difficult access patterns. Finally, our linear file model implementation provides a simple method of performing asynchronous data transfer, allowing applications to overlap computation and I/O.

The most common method of implementing a linear file model on a parallel file system is to break the file into a series of blocks, all the same size, and decluster them across the disks in the system, in round-robin order [Pie89, LIN+93, SGM86]. This method is relatively simple to implement, and can lead to good performance in some cases. Specifically, when individual nodes in an application access a linear file in large chunks, each access will involve data stored on multiple disks. By accessing those disks in parallel, the file system is able to deliver high aggregate bandwidth. The downside of this implementation of a linear file model is that an application that accesses data in small chunks is not able to achieve this high aggregate bandwidth, as each request involves data from only a single disk. Naturally, the designers of such systems were aware of this limitation, but since they expected multiprocessor applications to access data in large chunks, this limitation was not expected to be problematic in practice.

7.2.1 Implementation

As with most linear file model implementations, our implementation declusters its data by assigning blocks to disks in round-robin order. Unlike most systems, however, we allow (but do not require) applications to specify the declustering unit. That is, applications may choose the size of the blocks to be distributed. When a file is first created, the default declustering unit is 32 KB — the size of blocks in the Galley file system. An application may change the declustering unit at any time. Furthermore, an application may choose to make that change either temporary or persistent. A temporary change lasts only as long as the application has the file open. A persistent change lasts until the file is deleted, or until another application makes a persistent change in the declustering unit. Changing the declustering unit does not change the data already in the file, which can lead to unexpected results if one application makes a persistent change that is not anticipated or detected by the next application to access the file. We do not place any restrictions on the size of the declustering unit.

Most parallel file systems provide applications with little more than a standard Unix interface. Specifically, they only allow applications to accesses contiguous regions of the file in a single request. The most notable exceptions to this rule are Vesta [CF96] and MPI-IO [The96], which we discuss further in Chapter 8. Both of these systems allow applications to describe *logical views* of a file. That is, each process in a parallel application describes the subset of the file that it wishes to access. The file system then transparently maps requests against an abstract linear model to that process's subset of the file. Our linear file model provides interfaces similar to those provided by Galley. Specifically, we allow applications to explicitly make strided, nested-strided, and batched requests.

Unlike Galley's low-level interface, LFM provides applications with a file pointer for each open file. Although many linear-file-based parallel file systems provide shared file pointers, our library does not provide this facility for two reasons. First, we would need to find some way for all the nodes in an application to share the file pointer. Sharing this information in a reasonably efficient and portable way would be difficult unless we relied on some particular message-passing library. Since we did not want to limit the users' choice of language or message-passing library, this restriction was not acceptable. Second, and more importantly, application programmers do not seem to want

this feature. In the CFS workload we traced, those modes that used shared file pointers were almost never used. In the CMMD workload, modes with shared file pointers were used, but only because those modes were substantially faster than the independent modes. Anecdotal evidence collected from those users indicated that they would have preferred to use the independent modes [PEK+95].

7.2.2 Data Access Interface

The standard Unix interface provides only simple primitives for accessing the data in files. These primitives are limited to `read()`ing and `write()`ing consecutive regions of a file. In addition to a Unix-style interface, our linear file model provides three interfaces that allow applications to explicitly make regular, structured requests such as those observed in our workload characterizations. These interfaces allow the file system to combine many small requests into a single, larger request, which can lead to improved performance, as shown in the previous chapter.

The higher-level interfaces offered by our linear file model are summarized below. Although we show only the read requests, there are corresponding write requests for each call.

Simple-strided requests

```
int lfm_read_strided(int fid, void *buf, long size,
                     long f_stride, long m_stride, int quant)
```

Beginning at the location indicated by the file pointer, the library will read `quant` records, each of `size` bytes, from the open file associated with `fid`. As with Galley's simple-strided request, the offset of each record is `f_stride` bytes greater than that of the previous record. The records are stored in memory beginning at `buf`, and the offset into the buffer is changed by `m_stride` bytes after each record is transferred. Either the file stride (`f_stride`) or the memory stride (`m_stride`) may be negative. The call returns the number of bytes transferred. When the call is completed, the library updates the file pointer to point to the byte following the last byte in the final record read. Note that if the file stride is negative, the file pointer may actually move backwards in the file following this call. As with the standard Galley interface, if a negative stride in a read request leads to overlapping records in memory, the contents of that memory are undefined when the read request completes.

Nested-strided requests

```
int lfm_read_nested(int fid, void *buf, long size,
                    struct lfm_stride *vec, int levels)
```

The `vec` parameter is a pointer to an array of `(f_stride, m_stride, quantity)` triples listed from the innermost level of nesting to the outermost. The number of levels of nesting is indicated by `levels`. As with the simple-strided request, the location indicated by the file pointer is used as the initial offset for the request. Again, when the call is completed, the library updates the file pointer to point to the byte following the last byte in the final record read.

Nested-batched requests

```
int lfm_read_batched(int fid, void *buf, long size,
                     struct lfm_batch *vec, int quant)
```

Our Linear File Model library also provides applications with a nested-batched request. Note that the syntax for this request is nearly the same as that of Galley's nested-batched request. Unlike Galley, when using LFM, even the initial file offset may be given relatively. If the first file offset of a nested-batched request is relative, the library calculates the offset relative to the current file pointer. Since Galley does not maintain a file pointer, there is no such point that may be used as a base in a `gfs_read_batched()` call.

Non-blocking I/O

LFM provides applications with a simple means of utilizing non-blocking I/O. The non-blocking calls are of the form `lfm_nb_read_strided()`, and the parameters are exactly the same as those of the blocking calls. When using non-blocking I/O, the file offset is updated just before the call returns, not when the I/O is actually transferred.

Under LFM, there is no notion of a handle; applications may call `lfm_test()` and `lfm_wait()` on the file ID. Unlike Galley's low-level interface, there is no need to 'clear' the non-blocking request. Instead, once all the data for the non-blocking call has been transferred, the call is automatically cleared by the next blocking or non-blocking call on that file.

This approach has the limitation of restricting the application to one outstanding request per file. Any attempt to read or write from a file with an outstanding non-blocking request will cause

an error condition to be returned to the application. Since LFM provides the high-level interfaces necessary to request multiple chunks of data at once, we do not view this limitation as serious.

7.2.3 Performance

We tested the performance of the LFM library using the same system configurations and access patterns discussed in Chapter 6. While we applied those access patterns to individual forks in the tests described in that chapter, in these tests we applied the patterns to the whole file. In the tests discussed below, we use the default 32 KB striping unit. Each test was performed five times; we disregarded the lowest and highest results, and present the average of the remaining three.

When measuring the write performance in this section, we examined only the performance when rewriting an existing file. Since the library's behavior is exactly the same when creating a new file as when overwriting an existing file, any difference in performance between the two is caused by the underlying Galley file system. Since we already explored this issue with Galley's low-level interface in Chapter 6, we will not repeat it here.

Note that our LFM implementation is fairly straightforward, and little has been done to optimize its performance.

Traditional interface

We first examined the performance of the LFM Library using the standard Unix-style interface. Figure 7.1 shows the total throughput achieved when reading a file with various record sizes for each access pattern. Figure 7.2 presents similar results for write performance.

As with the Galley performance results discussed earlier, most of the performance curves have the same general shape as throughput curves in most systems; as the record size increased, so did the performance. Regardless of the access pattern, when accessing data in small pieces the total throughput was limited by the lower aggregate disk bandwidth, since the full parallelism of the file system was not being utilized, and by the high per-request latency, caused by software overhead and the inherently high latency of transferring data across a network.

Using the traditional-style interface with LFM led to lower performance than when directly accessing Galley's traditional interface. The primary cause of this difference is the number of disks being accessed. Using Galley's interface directly, each compute node always accessed every disk.

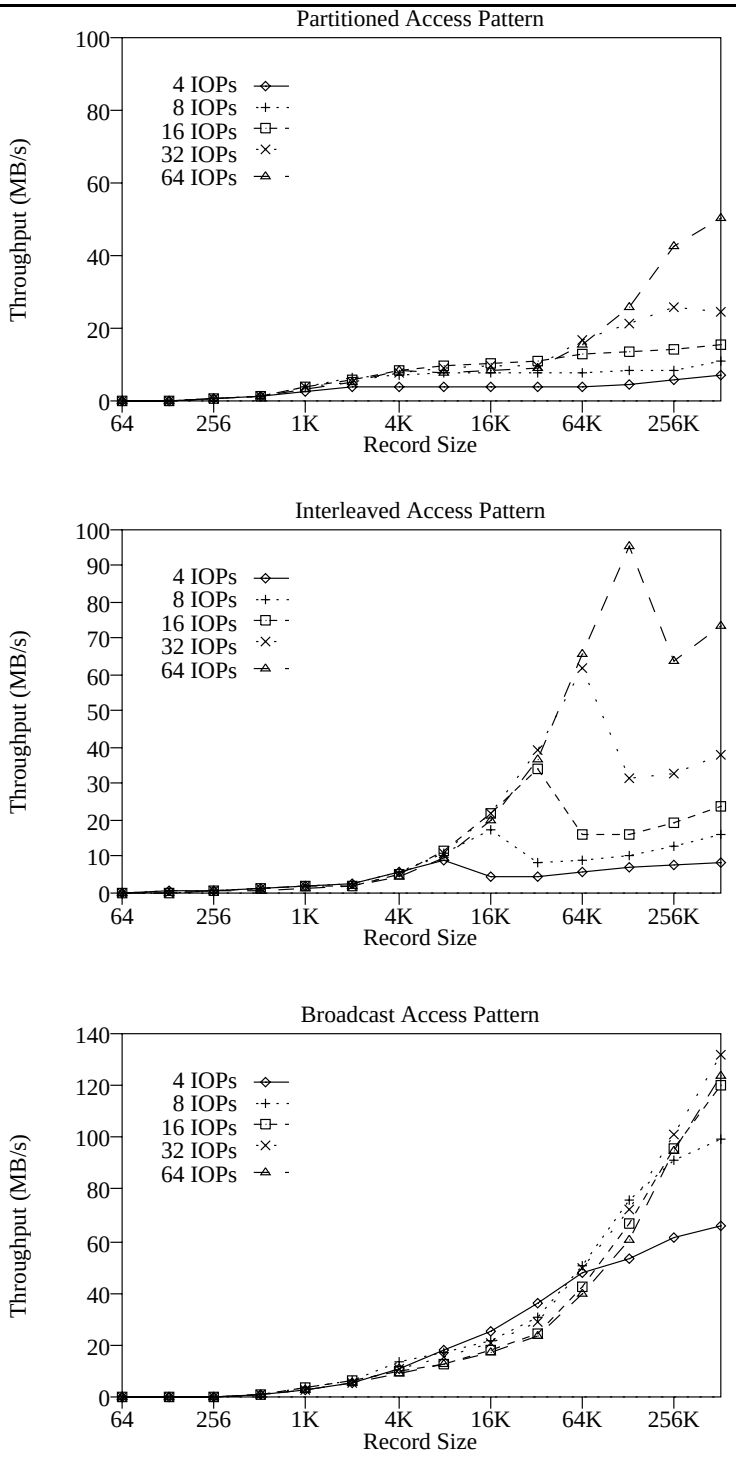

Figure 7.1: Throughput for **read** requests using LFM's traditional Unix-like interface with a 32 KB striping unit. There were 16 CPs in every case. Note the different scales on the y-axis.

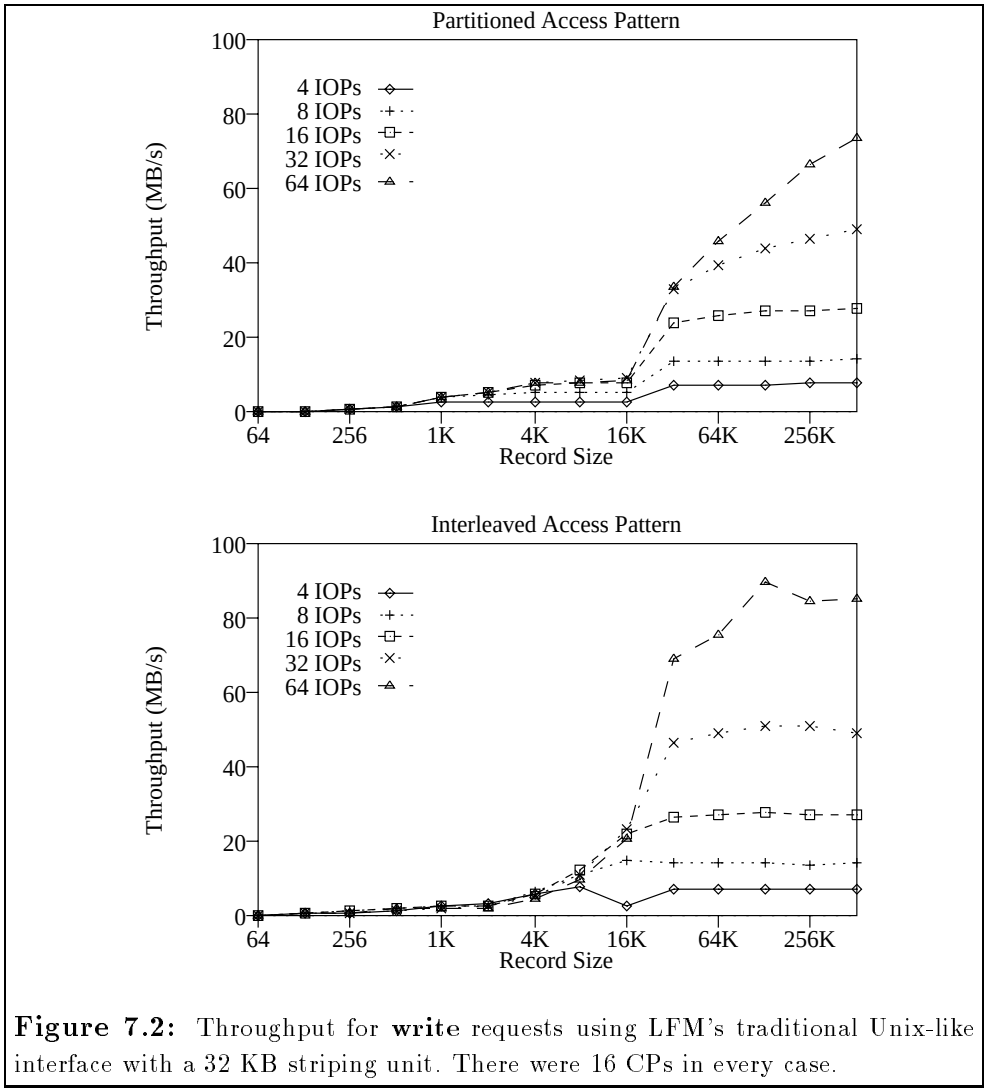

Figure 7.2: Throughput for **write** requests using LFM's traditional Unix-like interface with a 32 KB striping unit. There were 16 CPs in every case.

When making small requests for data using the LFM interface, it is possible that all of the CPs may be using the same one or two IOPs at once. This behavior is similar to that which we would expect to see in a traditional multiprocessor file system such as CFS or CMMD. Full I/O parallelism is only achieved when the record size is *#IOPs * striping_unit* bytes or greater.

The pattern with the lowest peak performance when reading with the traditional-style interface was the partitioned pattern. As with the low-level Galley interface, this poor performance was primarily caused by inefficient disk scheduling. Each compute node accessed data from a distinct region on each IOP, so each disk spent a great deal of time seeking from one region of the file to another.

Peak performance was better when reading an interleaved pattern, with the best performance occurring when the record size was exactly *(#IOPs /#CPs) * striping_unit*. With records that size or smaller, every CP that accessed data on an IOP accessed exactly the same blocks. With larger records, a single CP accessed every second block, or every fourth block, and so on. Since CPs were not accessing the same blocks, it was possible for requests for disk blocks to arrive out of order, degrading the disk schedule, and thus overall performance. As we discussed in Chapter 6, we saw similar behavior when using Galley's traditional interface for this pattern. The interleaved pattern achieved higher performance than the partitioned pattern because, even if requests arrived at the IOPs out of disk-order, the disk head was only forced to do small, local seeks, rather than seeking from one region of the file to another. Thus, the disks spent less time seeking.

Finally, as with Galley's low-level interface, we achieved the highest performance when reading the broadcast pattern. Every node read the same bytes in the file, in sequential order, so the IOPs were able to reuse the data stored in the buffer cache, and there were no out-of-order disk requests. With large numbers of IOPs, the overall performance was still lower than the raw Galley interface because each CP only accessed a subset of the IOPs on each request.

While the read tests generally outperformed the write tests, with large requests (i.e., 64 KB or larger), write performance was generally better than the read performance on the same access patterns. This difference in performance occurred because most of the disk accesses in the write case happened when at the end of a test, when we forced the IOPs to flush their buffer caches. Since most of the disk accesses happened at once, the DiskManagers were able to do a better job of scheduling disk accesses, avoiding many of the seeks that were required when reading. For the same reason, the interleaved-write pattern did not experience the same performance drop as the interleaved-read pattern.

Strided interface

We performed the tests in this section by issuing a single strided request from each CP for all of its desired records, rather than by issuing a separate request for each record. With the LFM library, the application running on each CP only had to issue one request, while with the raw Galley interface, each application had to issue a separate request to each IOP. While the code for the LFM benchmark was simpler than the Galley benchmark, this simplicity came with some

performance cost, particularly for small record sizes.

Figure 7.3 shows the total throughput achieved when reading a file with various record sizes for each access pattern using the new interface, and Figure 7.4 shows corresponding results for writing.

Using the strided interface, each CP generally accessed data stored on multiple CPs on each request, making full use of the parallelism available in the underlying system. Since the DiskManager got the full list of required blocks at once, the strided interface did not suffer the performance degradation due to poor disk scheduling that we saw with the traditional interface. Finally, with the strided interface, Galley was able to reduce the total number of packets sent across the network by packing multiple small records into larger packets.

In many cases, the peak performance of the strided requests came close to the peak performance achieved using Galley's raw interfaces on the same access pattern. However, this level of performance was not always achieved with a combination of small requests and large numbers of I/O nodes. These results indicate that the unoptimized implementation of the LFM spent a great deal of time translating the application's request against the linear file model to a collection of lower-level requests against Galley's multidimensional file model. This cost was further exacerbated by LFM's use of Galley's list-I/O interface. The list I/O interface incurs a fair amount of overhead, since it must verify that every (file offset, memory offset, record size) triple is valid. In contrast, Galley's three structured interfaces need only to verify that a small number of parameters are valid. So, it might be possible to reduce some of this overhead by using a strided Galley request to satisfy the most regular LFM requests. It is certainly likely that there is room for tuning the current LFM implementation in other ways as well, but it is also likely that there will still be measurable overhead for small requests.

Using Galley's low-level interface directly, we were able to achieve peak performance with records as small as 64 or 128 bytes. Using LFM, peak performance sometimes requires records of 256 bytes or greater. While this 256 bytes is slightly larger than the requests we found to be most common in practice on the iPSC/860, it is within the range of most of the requests on the CM-5. Furthermore, 256 bytes is far smaller than the request sizes needed to achieve peak performance in most other parallel file systems that provide linear file models [Nit92, LIN+93, Gro96]. Finally, although the peak speed is lower than that available through Galley's low-level interface, the increase in performance gained by using a strided interface is even greater for LFM than for the raw Galley

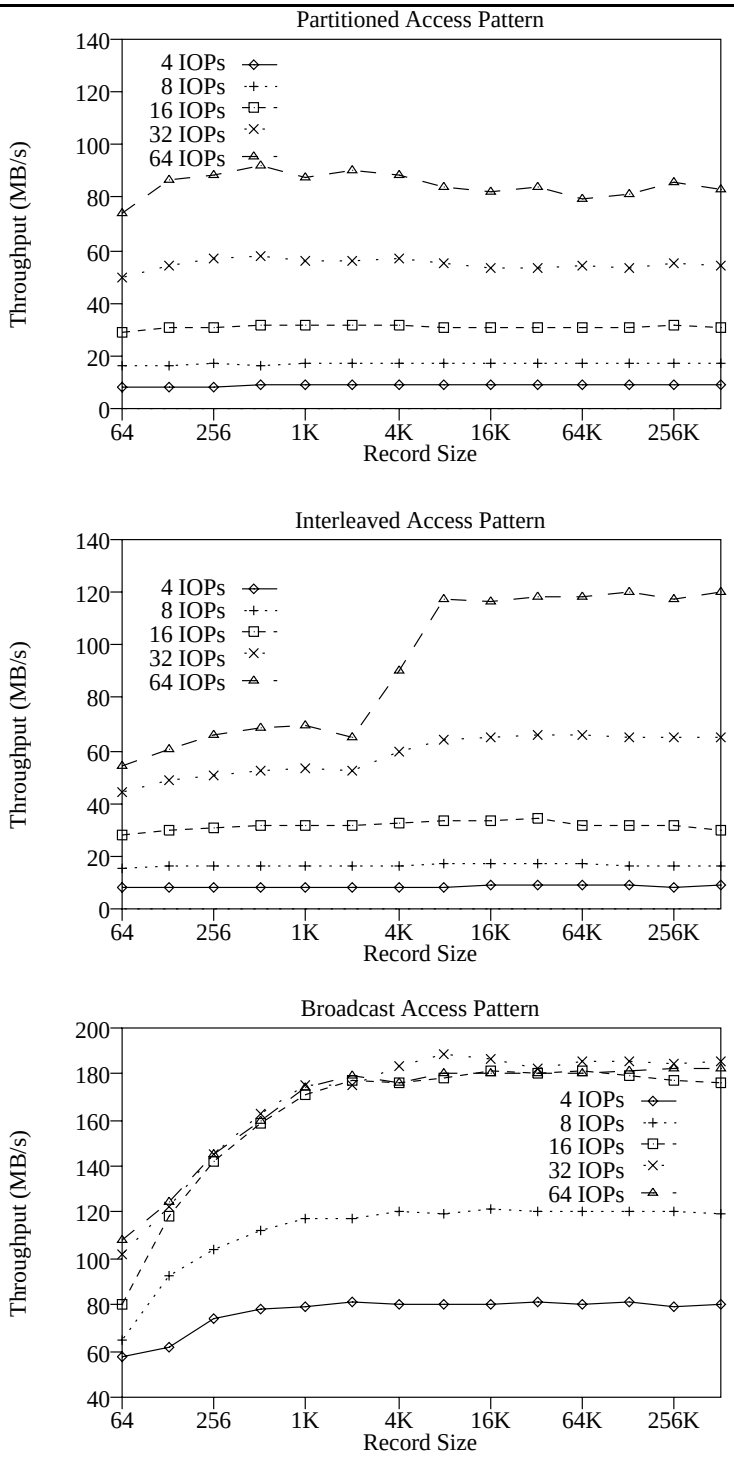

Figure 7.3: Throughput for **read** requests using the strided interface with a 32 KB striping unit. There were 16 CPs in every case.

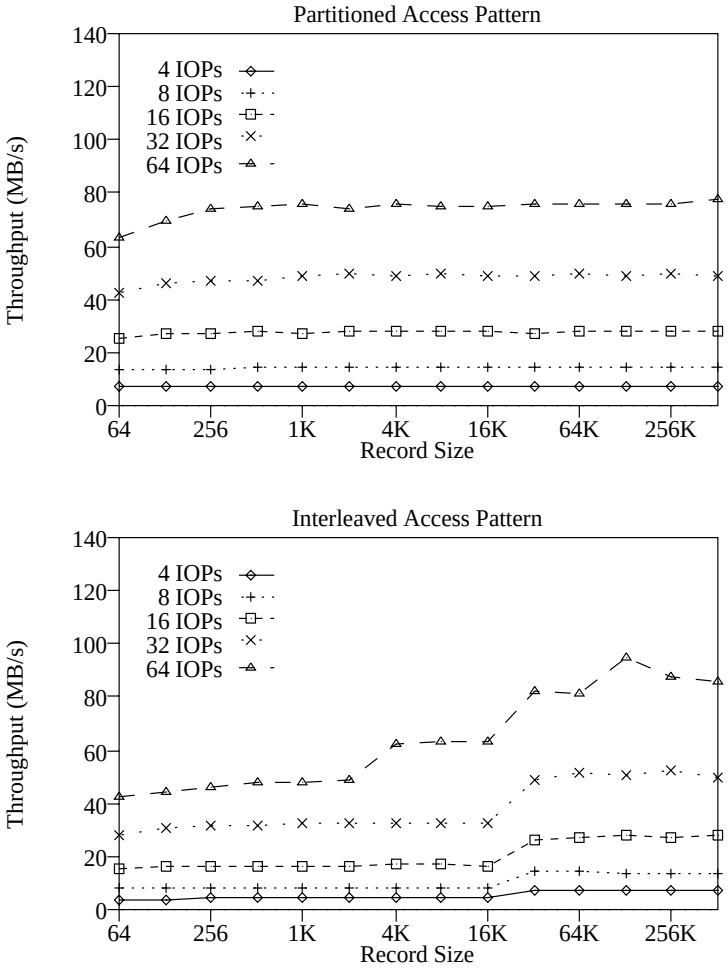

Figure 7.4: Throughput for **write** requests using the strided interface with a 32 KB striping unit. There were 16 CPs in every case.

interface, as can be seen in Figures 7.5 and 7.6, which show the speedup of read and write access patterns respectively.

When reading a partitioned access pattern, the strided interface resulted in speedups ranging from 42 to 142 for small requests. With an interleaved pattern, speedups ranged from 50 to just over 400. Finally, the broadcast pattern showed speedups ranging from 1100 to over 2000 for small requests. While there was a smaller increase in performance when writing, the increase was still greater than we saw with Galley's raw interface. The partitioned write pattern had speedups ranging from 37 to 97 for small requests, and the interleaved write pattern showed speedups from 25 to just over 250. The primary reason LFM showed more improvement with the strided interface

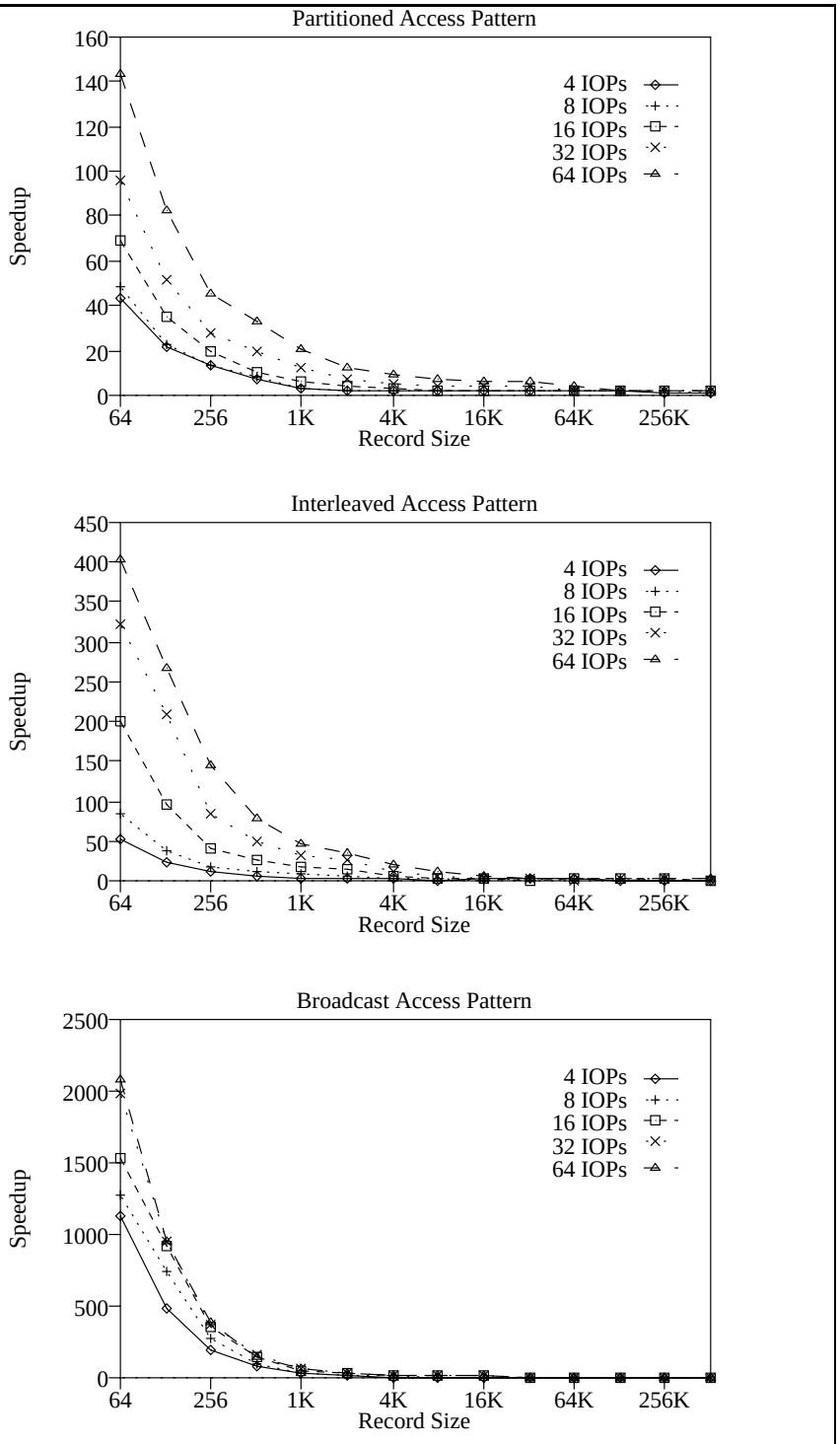

Figure 7.5: Increase in throughput for **read** requests using the strided interface. Note the different scales on the y-axis.

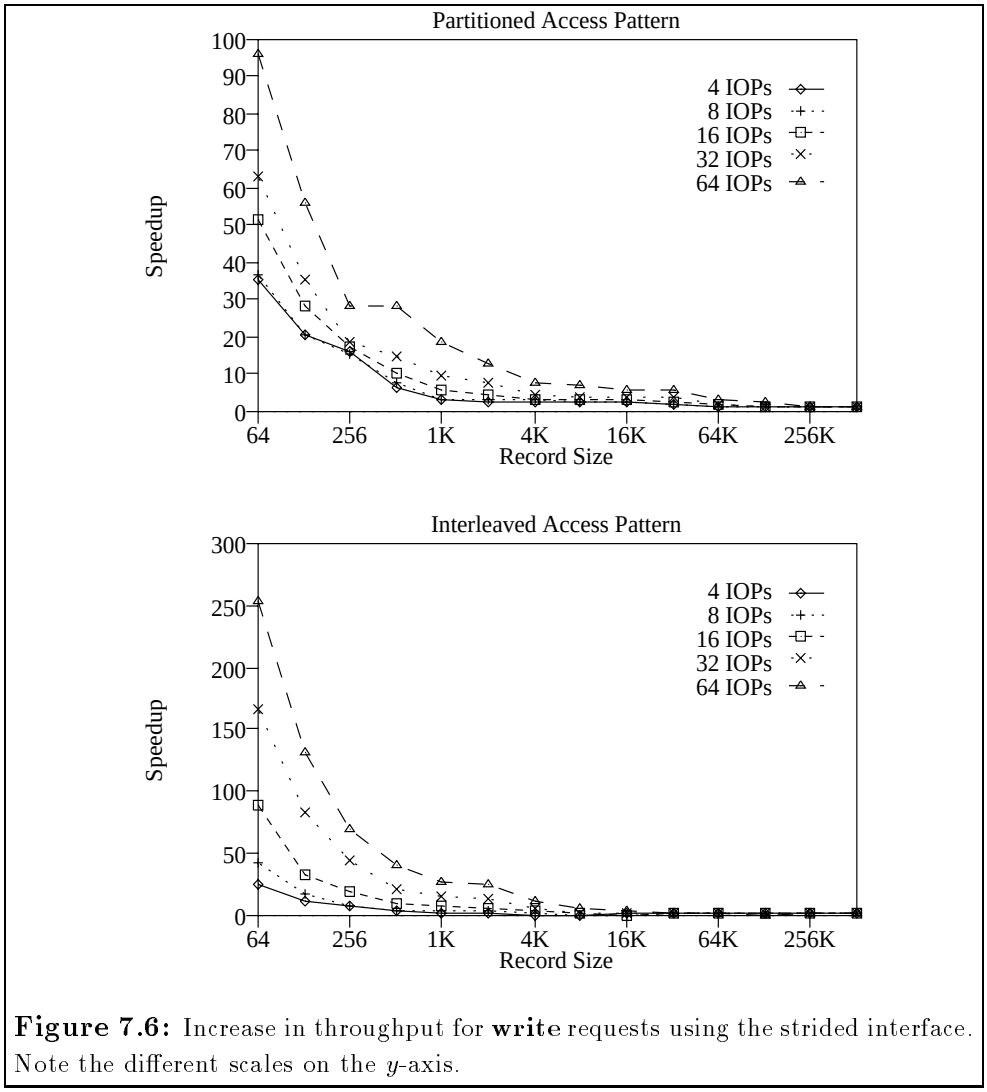

Figure 7.6: Increase in throughput for **write** requests using the strided interface. Note the different scales on the y-axis.

than Galley is that when testing Galley's raw performance, we allowed the CPs to use all the IOPs on each request, even using the traditional interface with small requests.

4 KB striping unit

To examine the effect of the size of the striping unit on the performance of the library, we repeated all the tests described above with a 4 KB striping unit. Figures 7.7 and 7.8 show the results of those tests using the traditional interface and Figures 7.9 and 7.10 show the results using the strided interface.

We first consider the performance of the traditional read interface. The partitioned pattern with the smaller declustering unit had better performance than the same pattern with the larger declustering unit, particularly with 64 IOPs. This performance increase was caused by the higher degree of parallelism being used on each request. For example, with a 32 KB declustering unit, a 256 KB request would only access data on 8 IOPs. With a 4 KB declustering unit, the same request would access data on 64 IOPs. This higher degree of parallelism can also be detrimental, however, as we see in the broadcast pattern. In that case, the 16-IOP configuration had the highest peak performance, with 32 IOPs performing slightly worse, and 64 IOPs performing significantly worse. As we have discussed before, in the broadcast pattern the network was typically the limiting factor. With the small declustering unit and large requests, each CP received small messages from every IOP, causing congestion at the CP's network interface. With only 16 IOPs, however, the number of incoming messages was smaller and the messages were larger, reducing latency and increasing bandwidth. As the other two patterns are disk-bound rather than network-bound, network congestion was less of a problem.

The traditional write interface produced even more interesting behavior. In the partitioned case, the performance was generally worse than with the larger declustering unit. With the larger unit, any request that was 32 KB or larger avoided the read portion of the read-modify-write cycle, since the entire disk block was overwritten. With the smaller declustering unit, a 32 KB write actually results in 4 KB writes at 8 different IOPs. Since the writes are distributed across multiple IOPs, we are not able to avoid this read until the record size is *#IOPs * disk_block_size* or larger. Indeed, we did see a dramatic increase in performance at those points (i.e., 128 KB with 4 IOPs, 256 KB with 8 IOPs, etc.). Similarly, the peak performance in the interleaved pattern occured when the combined writes from all the CPs (*#CPs * record_size*) were equal to *#IOPs * disk_block_size*. In that specific case, the disk schedule was perfect and all the IOPs were kept busy. We saw similar

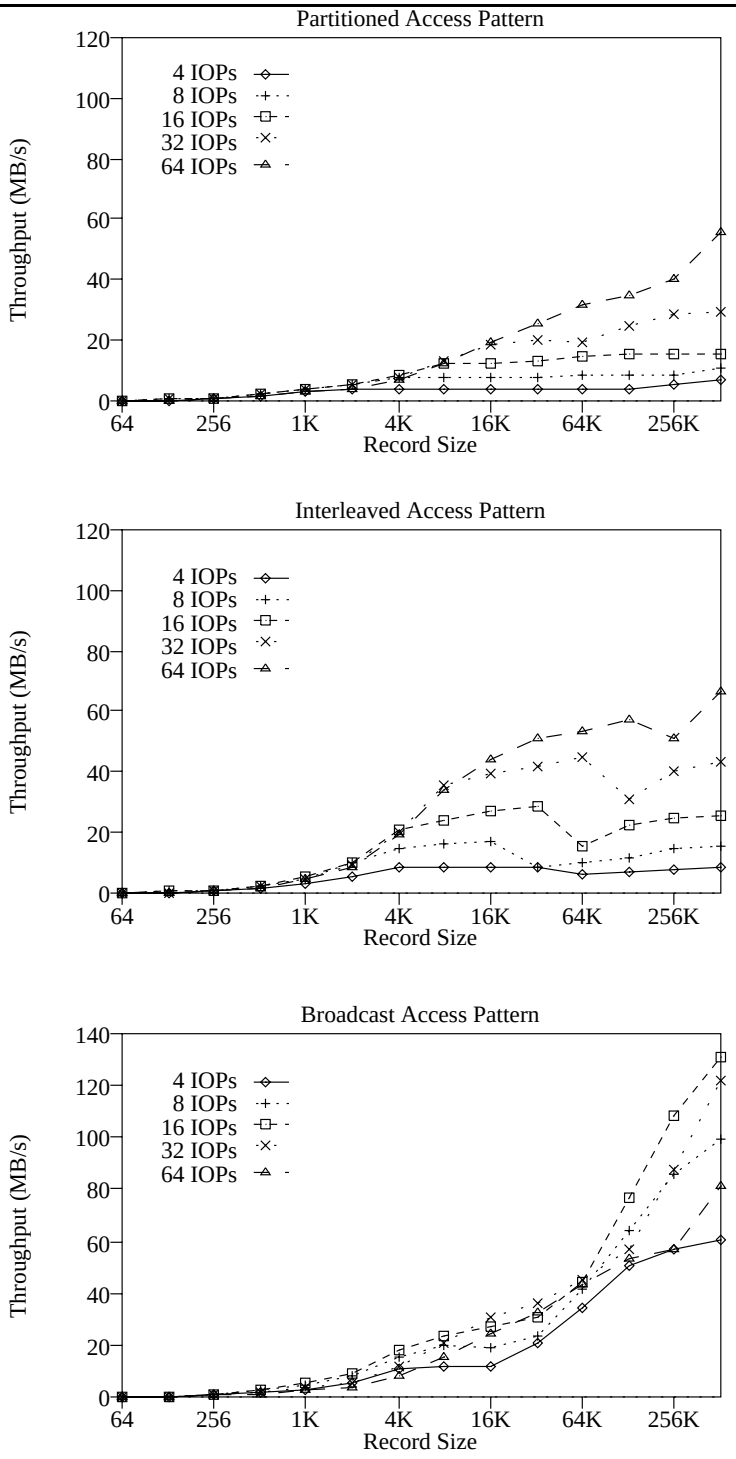

Figure 7.7: Throughput for read requests using the traditional Unix-like interface with a 4 KB striping unit. There were 16 CPs in every case. Note the different scales on the y-axis.

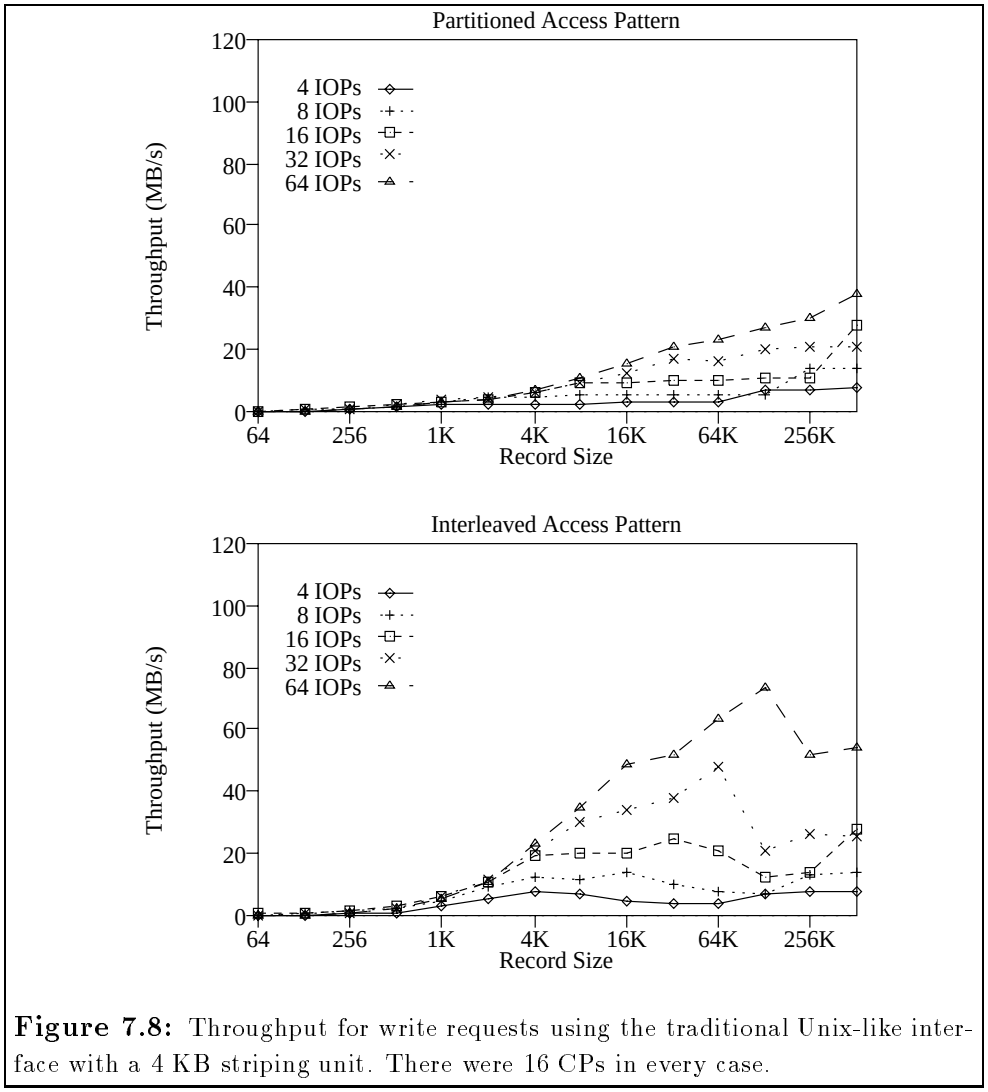

Figure 7.8: Throughput for write requests using the traditional Unix-like interface with a 4 KB striping unit. There were 16 CPs in every case.

behavior in the interleaved pattern in Figure 7.1.

The strided-read interface with the 4 KB declustering unit had performance nearly identical to the strided-read tests with the larger declustering unit, as did the partitioned-write test.

In the strided interleaved-write pattern with 16 or more IOPs, there was a significant bump in performance in the specific case when the *record_size = (#IOPs / #CPs)*declustering_unit*. In that one special case, each IOP was only servicing a single CP, so the pattern at the IOP was not interleaved; it was a contiguous write. With those conditions, the disk schedule was perfect and the read portion of the read-modify-write cycle could be avoided. As the record size increased beyond this peak, the number of CPs accessing each block was larger than one, so the the IOP

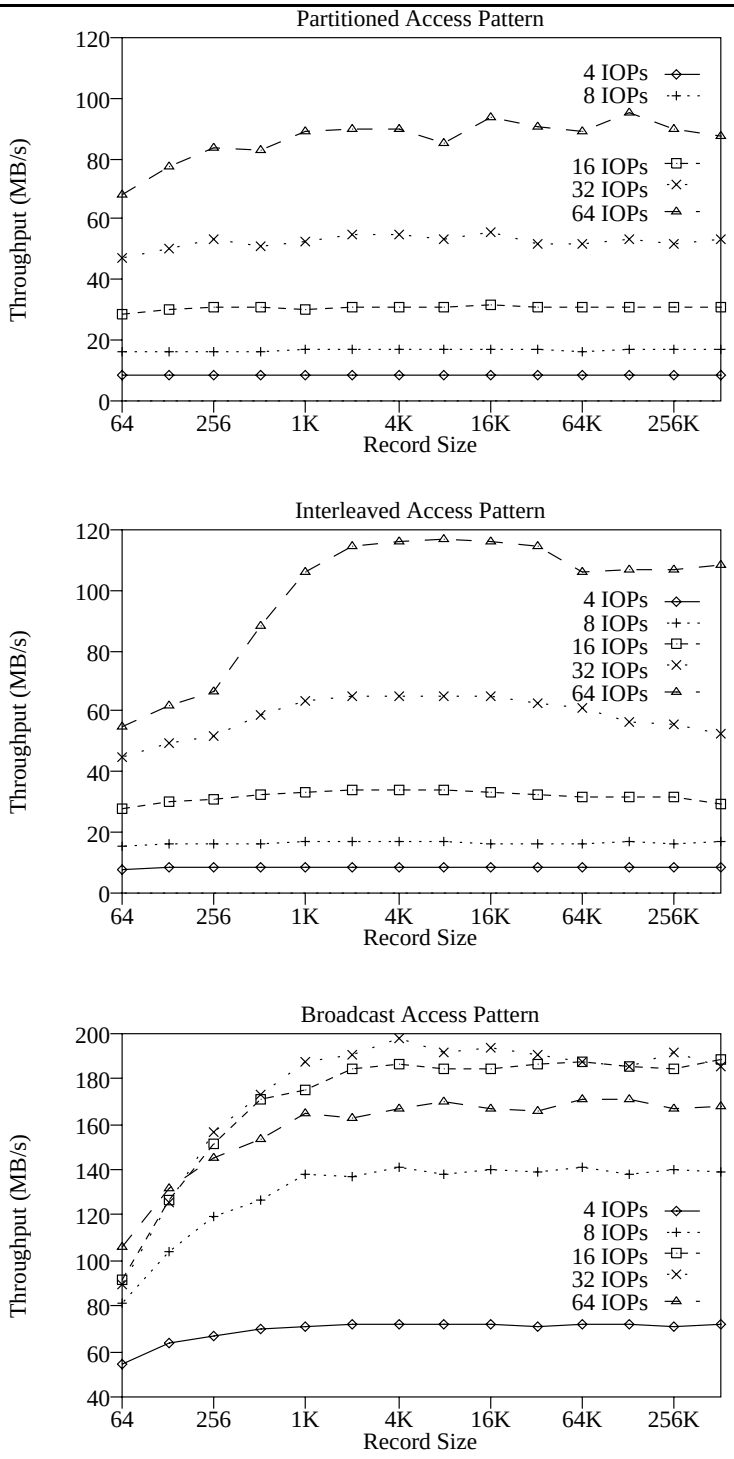

Figure 7.9: Throughput for read requests using the strided interface with a 4 KB striping unit. There were 16 CPs in every case. Note the different scales on the y-axis.

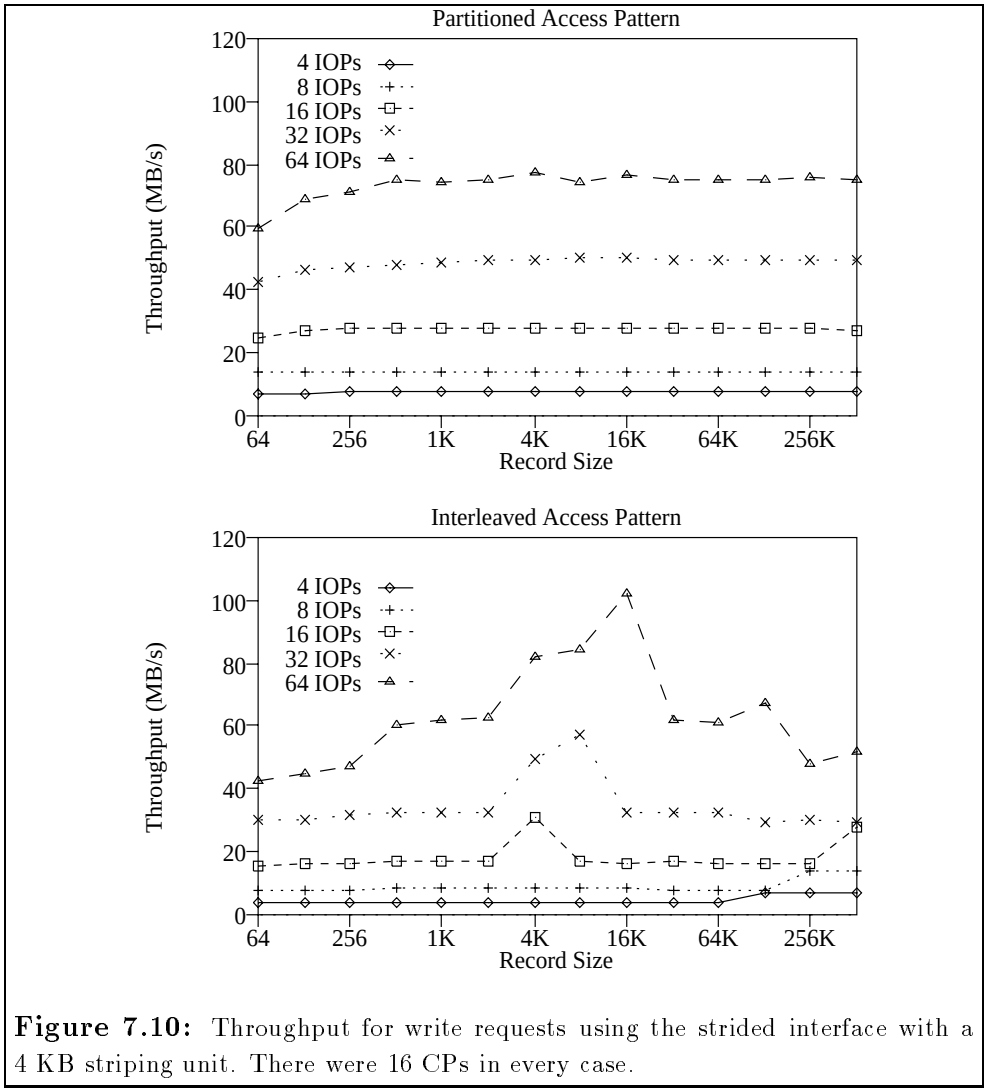

Figure 7.10: Throughput for write requests using the strided interface with a 4 KB striping unit. There were 16 CPs in every case.

had to perform the full read-modify-write cycle. Referring back to Figure 7.8, we can see a slight shoulder at each of the corresponding points in that plot. With 16 or fewer IOPs, the interleaved pattern exhibited an increase in performance with large requests, similar to that observed with the traditional interface.

Clearly, the choice of a declustering unit can have a significant impact on the performance of an application, suggesting that file systems should provide applications with this functionality. Unfortunately, selecting a "good" declustering unit is likely to be a difficult task, requiring a great deal of knowledge about the application's I/O patterns and the characteristics of the underlying file system.

7.3 BT I/O Benchmark

Several years ago, NASA Ames Research Center released a set of benchmarks, called the NAS Parallel Benchmarks (NPB), which has become a de facto standard for comparing the performance of high performance computers [BBDS92]. NPB was initially a set of paper benchmarks. That is, a set of problems were described in detail, but the implementations were left to the benchmarker. The goal of this approach was to find how quickly machines could solve real problems, rather than to find how quickly they could run a given piece of code. One of the main reasons this approach was adopted was the lack of a standard programming paradigm for parallel computers; each machine shipped with a proprietary message-passing interface or data-parallel language. With the rise of MPI and HPF, standards for both message-passing and data-parallel programming have become available. As a result, version 2 of NPB is an MPI-based collection of codes, rather than a paper benchmark [BHS+95]. There are plans to release a set of HPF-based benchmarks as well.

Following the success of NPB, NASA Ames attempted to devise a similar suite of benchmarks for I/O. Although this second suite has not caught on to the same extent as its predecessor, it is interesting nonetheless.

While most of the components of the suite were simple micro-benchmarks, there was one application benchmark as well: a modified version of one of the application benchmarks from NPB. Specifically, they used the BT benchmark with an added I/O phase. The BT benchmark is a pseudo-time stepping flow solver, and it involves finding the solution to a block tridiagonal system of equations. On each iteration, the solver must find solutions to three sets of uncoupled systems of equations. These systems are first solved in the x direction, then the y, and finally in the z direction.

The BTIO benchmark simulates the I/O required by such a flow solver that periodically writes its solution matrix to disk for postprocessing. The original I/O benchmark suite was also designed as a paper benchmark so, as with the computational benchmarks, implementors were free to write the BTIO benchmark in any way they saw fit. It is conceivable that an implementation that was designed for high I/O performance could have unacceptable computational performance. To avoid this problem, results were to be reported by comparing the total time of the I/O benchmark (including both I/O and computational time) with the *best* reported time of the standard BT

benchmark on the same machine. Although NAS has modified the NPB to be code benchmarks rather than paper benchmarks, the I/O suite has not yet been similarly changed — perhaps due to lack of interest.

Below, we discuss the implementation of the BTIO benchmark on top of Galley. Our implementation was based on the BT benchmark from NPB 2.1. In the spirit of the changes to NPB, we use the unmodified BT code as our basis for comparison, rather than attempting to identify the fastest SP-2 implementation of the original BT benchmark.

7.3.1 Data Distribution

In the NPB 2.1 version of the BT benchmark, the data is distributed according to a *multi-partition* scheme. In the multi-partition distribution, a three-dimensional matrix is partitioned into a number of disjoint three-dimensional submatrices, or *cells*, which are then distributed among the processors in the application. A simple, two-dimensional multi-partitioned matrix is shown in Figure 7.11. This distribution is designed to give good locality, which reduces inter-processor communication, and to lead to good load balancing, which increases processor utilization and reduces total execution time [vdW93].

Since the solution for each cell relies on knowledge about points on the borders of each surrounding cell, the submatrices on each processor are slightly larger than the size of the cells for which that processor is responsible. CPs exchange messages between each iteration to update this boundary information. A side effect of these larger matrices is that writing the solution matrix to disk is slightly more complicated; from its in-core matrices a node must extract just those points for which it is responsible, and store those points on disk.

7.3.2 Implementation

We implemented the BTIO benchmark on top of the linear file model in several different ways. In each case, we rely on information that is computed elsewhere in the code of the benchmark, such as the number of cells on each processor, and the coordinates of each cell, and so on. The BT benchmark is written in Fortran, and our output routines are all written in C. Recall that Fortran and C disagree on how to pass variables in a procedure call and how multidimensional matrices should be stored. So, rather than "translating" from Fortran to C every time we want to write out

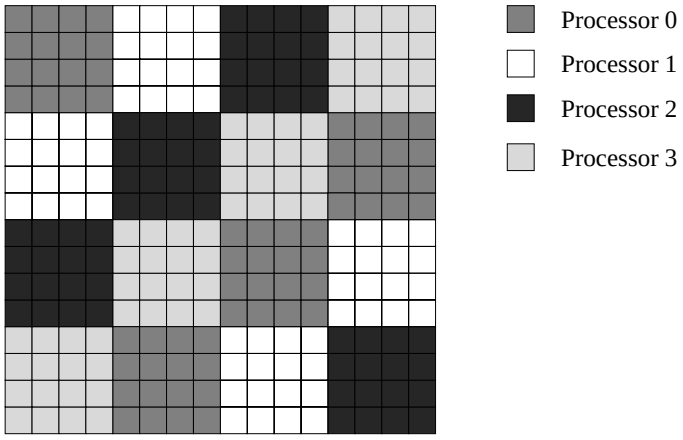

Figure 7.11: A 16x16 array distributed, in multipartition fashion, across 4 processors.

the solution matrix, we perform this translation once during the problem initialization phase, and store the results in the global variables shown in Figure 7.12. While the performance improvement caused by avoiding repeated translations is likely to be negligible, the readability of the output routines was greatly increased.

```
/* Location and size of a single cell within the global index space */
struct cell_shape {
      int x, y, z;                 /* Lower edge of the cell */
      int x_size, y_size, z_size;  /* Dimensions of the cell */
};

static struct cell_shape *cell_shape;  /* Shape of each local cell */

static int dim;          /* Problem size (A=64, B=102) */
static int pad;          /* Max. size of a cell with boundary info */
static int cells;        /* Number of cells local to this node */
static int nodes;        /* Number of nodes in application */
static int rank;         /* Rank of this node (between 0 and nodes-1) */
static int fd;           /* Target file for the solution matrix */
static double *fixed_u;  /* Safe copy of the solution matrix, so
                            non-blocking I/O can be done safely */

#define PT_SIZE 40       /* Each point in each cell is comprised of 5
                            8-byte doubles */
```

Figure 7.12: Global variables shared by all the different routines that output the solution matrix.

Other than the code shown below, our additions to the original BT code are simply for initial-

ization and shutdown of the file system, and a single call to the appropriate output routine.

The benchmark specification does not require that the data be stored on disk in any particular fashion; they only require that it be in some format accessible to other applications. All of our implementations store the 3-dimensional matrix on disk in one LFM file, with the z-axis most significant, then the y-axis, and finally, the x-axis. In C, this ordering would correspond to a 3-dimensional matrix with the subscripts in the following order: z, y, and x. In Fortran, this would correspond to a 3-dimensional matrix with the subscripts in the reverse order, or x, y, and z, which is the ordering used by the NAS implementation of the BT benchmark.

The first example, shown in Figure 7.13, shows how one could implement the benchmark using only the nested-strided interface. Each processor is able to write out each of its three-dimensional cells using a single nested-strided write. The memory striding allows them to skip over the padding, those boundary points that 'belong' to other processors. The file striding allows them to map the three-dimensional submatrices to a linear file. Since the cells belonging to a single processor are not always equidistant in the file (see for example the cells belonging to Node 2 in Figure 7.11), there is no common stride that we may take advantage of at the inter-cell level. Thus, we must issue a separate request for each of the cells that a processor writes to disk.

The second example, shown in Figure 7.14, shows one possible implementation of the I/O phase using a nested-batched request. This type of request allows us to submit multiple nested-strided requests at once. So, each `inner` vector directly corresponds to one of the inner-strided patterns in the first example, and each `outer` vector corresponds to one of the outer-strided patterns. Finally, the `top` vector allows us to collect all of the nested-strided requests together, and submit them all at once. It is worth noting that if each of the cells on a single processor were the same size, we would only need a single `inner` vector and a single `outer` vector, rather than one for each cell. We could use the `top` vector to repeat the same nested-strided pattern for each cell, just changing the initial memory and file offsets between each repetition of the pattern.

The final example, also shown in Figure 7.14, shows a simple way to use non-blocking I/O to increase the overall performance of the benchmark. In this case, we are able to simply copy the solution vector to a safe location in memory, and perform the I/O on that fixed copy while the benchmark continues to work on the original. If the solution vector were larger, it is possible that there might not be enough memory to maintain both an original and a fixed copy of the matrix.

```
void
nested(double *u)
{
    long f_off, m_off;
    char *base;
    struct lfm_stride stride[2];
    struct cell_shape *c;
    int i;

    base = (char *) u;
    for (i=0; i < cells; i++) {
        c = &cell_shape[i];
        f_off = (c->z*dim*dim + c->y*dim + c->x) * PT_SIZE;
        m_off = (i*pad*pad*pad + 3*pad*pad + 3*pad + 3) * PT_SIZE;

        stride[0].f_stride = dim*PT_SIZE;
        stride[0].m_stride = pad*PT_SIZE;
        stride[0].quant    = c->y_size;

        stride[1].f_stride = dim*dim*PT_SIZE;
        stride[1].m_stride = pad*pad*PT_SIZE;
        stride[1].quant    = c->z_size;

        lfm_lseek(fd, f_off, SEEK_SET);
        lfm_write_nested(fd, &base[m_off], c->x_size*PT_SIZE,
                   stride, 2);
    }
}
```

Figure 7.13: Implementation of the I/O portion of the BTIO Benchmark using LFM's nested-strided interface.

In that case, a more sophisticated approach to non-blocking I/O would be necessary.

7.3.3 Performance

Table 7.2 shows the length of time required to run the complete benchmark with problem size A, which has an output matrix with 64x64x64 elements, each of which contains 5 floating-point doubles. Table 7.3 shows similar results with problem size B, which has an output matrix with 102x102x102 elements. In each case, the benchmark ran for 200 iterations, and the solution matrix was written out every 5 iterations. For both benchmark sizes, we used varying levels of I/O support from the Linear File Model library. For all the tests, we held the number of I/O nodes constant at 16 and varied the number of compute nodes. The benchmark is structured in such a way as to require a square number of nodes. We ran each benchmark three times, and report the average length of time.

The performance of the benchmark on problem size A is essentially what we expected. The

```
void
batched(double *u)
{
    struct lfm_batch inner[20], outer[20], top[20];
    struct cell_shape *c;
    int i;

    for (i=0; i < cells; i++) {
        c = &cell_shape[i];

        top[i].f_off = (c->z*dim*dim + c->y*dim + c->x)*PT_SIZE;
        top[i].m_off = (i*pad*pad*pad + 3*pad*pad + 3*pad + 3)*PT_SIZE;
        top[i].f_absolute = 1;
        top[i].m_absolute = 1;
        top[i].subvec_len = 1;
        top[i].sub_vector = 1;
        top[i].quant    = 1;

        outer[i].f_off    = inner[i].f_off    = 0;
        outer[i].m_off    = inner[i].m_off    = 0;
        outer[i].f_absolute = inner[i].f_absolute = 0;
        outer[i].m_absolute = inner[i].m_absolute = 0;

        inner[i].quant    = c->y_size;
        inner[i].f_stride   = dim*PT_SIZE;
        inner[i].m_stride   = pad*PT_SIZE;
        inner[i].subvec_len = 1;
        inner[i].sub_vector = 0;
        inner[i].sub.size   = c->x_size*PT_SIZE;

        outer[i].quant    = c->z_size;
        outer[i].f_stride   = dim*dim*PT_SIZE;
        outer[i].m_stride   = pad*pad*PT_SIZE;
        outer[i].subvec_len = 1;
        outer[i].sub_vector = 1;
        outer[i].sub.subvec = &inner[i];

        top[i].sub.subvec = &outer[i];
    }

#ifdef USE_NON_BLOCKING_IO
    lfm_wait(fd);   /* Make sure previous write has completed */

    bcopy(u, fixed_u, pad*pad*pad*PT_SIZE);
    lfm_nb_write_batched(fd, fixed_u, top, cells);
#else
    lfm_write_batched(fd, u, top, cells);
#endif
}
```

Figure 7.14: Implementation of the I/O portion of the BTIO Benchmark using LFM's nested-batched interface. This example also shows how little needs to be changed to use non-blocking I/O.

Compute Nodes	I/O Type				
	None	Trad.	Strided	Batched	NB-Batched
9	320.9	415.2	325.3	324.9	323.6
16	188.7	271.8	192.5	189.4	188.9
25	124.3	212.8	129.8	126.6	124.4
36	91.9	182.2	102.8	94.1	92.9
49	69.8	163.8	79.8	72.1	71.6

Table 7.2: Seconds required to complete one run of the BTIO Benchmark on problem size A (64x64x64). The number of I/O nodes was held constant at 16.

Compute Nodes	I/O Type				
	None	Trad.	Strided	Batched	NB-Batched
16	762.1	962.0	768.2	781.8	768.2
25	502.3	708.0	514.8	534.8	506.6
36	351.1	573.8	368.0	365.0	356.8
49	268.6	492.2	287.2	280.4	271.9
64	209.3	453.3	232.0	217.5	210.5

Table 7.3: Seconds required to complete one run of the BTIO Benchmark on problem size B (102x102x102). The number of I/O nodes was held constant at 16.

execution time was lowest with no I/O performed. The traditional-interface implementation was by far the slowest of the I/O benchmarks – taking up to 135% longer than the benchmark without any I/O. With the higher-level requests, the benchmark with I/O required no more than 14% longer than the benchmark without I/O. The nested-strided implementation, where each cell was written with a separate request, was the slowest of the higher-level implementations. The nested-batched implementation was faster, and the non-blocking implementation was the fastest. This ordering held regardless of the number of processors.

With problem size B, the traditional-interface implementation required up to 117% longer than the benchmark with no I/O. The slowest higher-level implementation took 11% longer than the benchmark without I/O, and most took significantly less time. Surprisingly, with fewer than 36 processors the blocking nested-batched implementation was slower than the implementation that used the nested-strided interface. One possible explanation for this unexpected behavior could be the overhead imposed by one of Galley's attempts at optimization. Given a list of I/O requests, Galley sorts them and then attempts to coalesce any adjacent chunks into larger chunks. With

the strided implementation, the list of I/O requests would be created in sorted order, and Galley would not bother to sort them again. With the batched implementation, the requests within a cell would be generated in order (as in the strided implementation), but the cells themselves may not be generated in the same order they appear in the file. So, in this case, Galley would try to sort the whole list of I/O requests before sending them to the IOPs. With problem size A and with large numbers of processors on problem size B, the number of chunks to be sorted is smaller, so this optimization would incur less overhead.

7.4 Other Projects

The preceding sections discuss work that was performed as a direct part of the research for this dissertation. Galley has also been used by other people.

The Panda Array I/O library was designed at the University of Illinois, to support high performance I/O for multidimensional matrices [SCJ+95, SW94]. While most such projects are based on some variant of Fortran, this group chose to examine the issue of supporting distributed matrices under C++. Joel Thomas, a Dartmouth undergraduate, redesigned and modified the Panda Array Library to run on top of the Galley Parallel File System [Tho96].

The Vesta file system, from IBM Research, uses a two-dimensional file model. Vesta provides applications with a concise method of describing how these files should be partitioned among the compute nodes in an application. Matt Carter, another Dartmouth undergraduate, has implemented a library supporting the Vesta interface on top of Galley. In particular, Galley's subfiles and forks simplified the implementation of Vesta's two-dimensional file structure, and Galley's nested-strided interface supports Vesta's logical views. Vesta is described in greater detail in the next chapter.

SOLAR is a library of routines to support applications that use out-of-core, dense, matrix computations [TG96]. SOLAR relies on existing high-performance in-core subroutine libraries to do much of the computation, and it provides its own optimized matrix I/O library. SOLAR's author ported the package to Galley, and found that the subfile model provided him with a useful degree of control over the distribution of data. He also found that Galley's nested-strided interfaces allowed him to achieve significantly better performance than the two-phase I/O strategy he had

originally employed.

Finally, a group at Dartmouth is developing a compiler for ViC*, a variant of the data-parallel C* [CC94]. ViC* is designed to easily support out-of-core programming methods. Galley is one of the file systems that they are targeting. Unlike most parallel file systems, Galley's subfiles provide the functionality they need for many of their I/O-optimal algorithms, which require the ability to explicitly access specific disks.

Chapter 8

Related Work

In this chapter we provide some more details about other multiprocessor file systems, and compare them to Galley.

8.1 Unix-like Parallel File Systems

Most commercial multiprocessor file systems are based on the Unix linear file model. Some simply provide a Unix-like interface, and some provide the full semantics required by Unix standards.

8.1.1 CFS/PFS

Intel's Concurrent File System [Pie89, Nit92] is frequently cited as the canonical first-generation parallel file system. CFS was written for the iPSC family of parallel machines. Its successor, PFS, is similar and was written for the Paragon [EK93, RP95]. CFS and PFS provide a simple, Unix-like interface to the application. The blocks of a file are declustered across all the disks in round-robin order. CFS and PFS extend the conventional Unix interface to provide support for parallel applications by introducing several varieties of shared file pointer. The simplest form of shared file pointer is similar to the "atomic append" of BSD 4.3 [LMKQ89]. The remaining two types of shared file pointer are similar to the first, but enforce round-robin access to the file from all the nodes. The second type of pointer allows arbitrary sized records while the third, and fastest, shared mode requires that all records be of the same size. In addition to these four modes, PFS provides a *broadcast* mode, which allows all processes to access the same data. As we mentioned earlier, these modes were rarely used in practice on the iPSC/860 at NASA Ames.

8.1.2 sfs

Many of the applications on the CM-5 are written in a data-parallel language such as CMF, C*, or *Lisp. To efficiently support data-parallel I/O operations, Thinking Machines developed a file system called *sfs*, which was derived from SunOS [LIN+93]. Files in *sfs* are distributed across *logical devices*, which are groups of physical disks, clustered into a level 3 RAID. While *sfs* is capable of providing high bandwidth for large transfers that span multiple blocks, a high start-up latency leads to poor performance for small requests. *sfs* also has structural problems that lead to poor performance when multiple files within a single cylinder group are used [LIN+93]. *sfs* was designed as a low-level file system. Users were expected to access it through the I/O support built in to one of their data-parallel languages or through the CMMD library, described below.

8.1.3 CMMD

The CMMD library allows CM-5 applications to be written in a control parallel style [BGST93]. CMMD maintains the traditional stream-of-bytes abstraction of a file and supports all the standard Unix-like operations, which may be executed by each compute node individually. In addition to the standard operations, CMMD includes support for parallel applications in the form of access modes like those in Intel's CFS. The simplest mode is *local*, which is similar to traditional Unix semantics; each node maintains its own file pointer and may operate on the file without any communication with other nodes. CMMD also offers an *independent* mode in which each node maintains its own file pointer, but all other state is shared by all the nodes. CMMD also offers two *global* modes in which there is a single shared file pointer.

8.1.4 PIOFS

PIOFS, a parallel file system for IBM's SP-2, allows users and applications to interact with it exactly as they would interact with any AIX file system [CF96]. Indeed, the parallel file system is mounted on each of the nodes of the SP-2 using AIX's standard Virtual File System interface. Although PIOFS may appear as a standard sequential file system, it is implemented on top of the Vesta parallel file system (discussed below). Using Unix's `ioctl()` facility, administrators and advanced users may interact with the underlying parallel file system.

8.1.5 SUNMOS and PUMA

SUNMOS and its successor, PUMA, are operating systems that were developed by Sandia National Laboratory and the University of New Mexico for the Intel Paragon [WMR+94]. The design goal behind both SUNMOS and PUMA was to make as much of the power in the hardware available to the user as possible. For this reason, the designers adopted simple interfaces that could be implemented efficiently. Rather than attempting to match the full semantics of Unix's low-level I/O calls, their interface was designed to be compatible with the *stdio* library calls (e.g., `fread()`, `fwrite()`, etc). These calls are translated into requests for the I/O nodes, which use the high-performance message-passing system to transfer data to and from the compute nodes.

Neither of these operating systems actually provide their own low-level file system; both are built on top of the underlying PFS and UFS file systems.

8.1.6 OSF/1 AD

While SUNMOS and PUMA offer a "lean and mean" operating system to users of the Intel Paragon, OSF/1 AD provides the full power of the Unix operating system to each compute node [ZRB+93]. This power does come at some cost; on a Paragon, SUNMOS requires only 256KB of memory on each node, but OSF/1 AD occupies nearly 10MB [MMRW94]. File system access in OSF/1 is built on top of Mach Memory Objects [Roy93]. Since memory objects are restricted to multiples of the page size, small, non-contiguous requests, such as those common in parallel scientific workloads, may lead to poor performance. There is no support for a multiprocessing environment at the user interface. Furthermore, the designers explicitly did not intend for OSF/1 to satisfy the I/O performance needs of scientific, supercomputer applications [Roy93].

8.1.7 PPFS

Like the systems mentioned above, PPFS provides the end user with a linear file that is accessed with primitives similar to the traditional `read()/write()` interface [HER+95]. In PPFS, however, the basic transfer unit is an application-defined *record*, rather than a byte. PPFS maps requests against the logical, linear stream of records to an underlying two-dimensional model, indexed with a `(disk, record)` pair. Several different mapping functions, corresponding to common data distributions, are built into PPFS. An application is able to provide its own mapping function as well.

8.1.8 SIO Interface

The Scalable I/O Initiative is a collection of researchers from industry, academia, and the national labs. While the SIO Initiative has many goals, the most relevant to our work is the design of a new low-level interface for parallel I/O [BBD+94]. The long-term goal of the Operating Systems Working Group is to design a set of interfaces that may be added to the standard X/Open 4.2 interfaces. That is, the interface presented by the group is intended to be an extension to the Unix file system, rather than a replacement of it. The core of the Scalable I/O Iniative's interface allows applications to submit lists of simple-strided requests [CPD+96]. Earlier proposals borrowed our nested-batched requests for the core of their interface, but this level of functionality is now described as an "extension" to the core interface.

8.1.9 Scotch

Scotch is a parallel file system from Carnegie Mellon University [GSC+95]. It appears that they eventually intend Scotch to support the SIO low-level interface. Scotch differs from other Unix-like parallel file systems in two significant ways. The first difference is Scotch's reliance on application-influenced prefetching for high performance. Scotch currently provides only a simple data-access interface, but it allows applications to provide *hints* about expected future data accesses. Scotch then uses these hints when deciding which data should be prefetched from disk. Since Galley provides applications with the primitives necessary to allow them to express their complete I/O requirements at once, there is less need for prefetching based on guesses about future data accesses. Scotch's other unusual feature is its support for per-file redundancy. While most parallel file systems stripe a file's data across all the disks in the system, Scotch allows applications to indicate that the system should store a *parity block* with each such stripe. As in a traditional RAID, the addition of a parity block allows the reconstruction of a file in the event of the failure of one disk.

8.2 Non-Unix Parallel File Systems

Although most commercially available parallel file systems are based on the Unix model, there has been a great deal of interesting research done using alternative interfaces and file models.

8.2.1 Bridge

Bridge was one of the earliest parallel file systems [DSE88, Dib90], and is unusual in not separating I/O nodes from compute nodes. Bridge provides three interfaces. The simplest, designed to be used by a single compute node, is similar to a traditional Unix interface and transparently distributes the data across the system's disks in blocks. Although the authors mention that application-controlled block distribution could be useful, there is no indication that this was ever provided except as an extremely simple and slow prototype. The second interface allows multiple clients to access the same file in a structured manner similar to one of the modes provided by CFS. The final interface allows applications to explicitly access the local file systems on the different nodes. The designers of Bridge appear to have intended that this interface be used primarily for manipulating data on disk (e.g., copying or sorting files) rather than for the transfer of data to and from compute nodes.

8.2.2 nCUBE

A file-system interface proposed for the nCUBE is based on a two-step mapping of a file into the compute-node memories [DJ93]. The first step is to provide a mapping from subfiles stored on multiple disks to an abstract dataset (a traditional one-dimensional I/O stream). The second step is mapping the abstract dataset into the compute-node memories. The first mapping function is provided by the system software, while the second mapping function is provided by the user. The first function is composed with the inverse of the second to generate a function that directly maps data from compute-node memory to disk. Their mapping functions are essentially a permutation of the index bits of the data.

While the nCUBE interface is far more elegant and aesthetically pleasing than Galley's interfaces, it does have several important limitations. The most serious of these limitations is a direct outgrowth of its elegance: since the mapping functions are based on permutations of the index bits, all sizes must be powers of 2. This restriction includes the number of I/O nodes, the number of compute nodes, the disk-block size, the unit-of-transfer size, and, for some data distributions, the matrix dimensions. This interface was implemented, but never released.

8.2.3 Vesta

The Vesta file system breaks away from the traditional one-dimensional file structure [CBF93, CF96, FCHP95]. Files in Vesta are two-dimensional, and are composed of multiple *cells*, each of which is a sequence of *basic striping units*. BSUs are essentially records, or fixed-sized sequences of bytes. Like Galley's subfiles, each cell resides on a single disk. While Galley only allows a file to have a single subfile per disk, in Vesta a single disk may contain many cells. As discussed in the previous chapter, equivalent functionality can be achieved on Galley by mapping cells to forks rather than subfiles. Vesta's interface includes *logical views*, or logical partitioning, of the data, which indicates how the data should be distributed among the processors. Not only does this logical partitioning provide a useful means of specifying data distribution, it allows significant performance gains since it can guarantee that each portion of the file will be accessed by only a single processor. This guarantee reduces the need for communication and synchronization between the nodes.

While Vesta provides a flexible and powerful method of specifying the distribution of a regular data structure across compute and I/O nodes, it too has limitations. Vesta seems ill-suited to problems that use irregular data, where irregular is defined as anything that cannot be laid out in a rectangle or that cannot be partitioned into rectangular sub-blocks of a single size. One of Vesta's great strengths is its two-dimensional file abstraction, which allows programmers to specify layout information that will hopefully lead to performance improvements. As discussed in the previous chapter, a Vesta-like interface has been built on top of Galley. It is interesting to note that the implementor of the library found that reimplementing the Vesta interface on top of Galley was an easier task than writing applications for the Vesta interface itself.

Neither nCUBE nor Vesta appear to provide an easy way for two compute nodes to access overlapping regions of a file. Since many models of physical events require logically adjacent nodes to share boundary information, this could be an important restriction. This behavior can also be seen in the file-sharing results in Chapter 3, which show that most read-only files had at least some bytes that were accessed by multiple processors. On the other hand, the same results show that in many cases, the strict partitioning offered by nCUBE and Vesta may match the applications' needs for write-only files.

8.2.4 MPI-IO

MPI-IO [The96] is a draft standard for parallel I/O, which derives much of its philosophy and interface from the MPI message-passing standard [MPI94]. In MPI-IO, file I/O is modeled as message passing. That is, reading from a file is analogous to receiving a message and writing to a file is analogous to sending a message. Just as MPI provides structured messages based on simple and derived types, access to files in MPI-IO is based on *etypes* and *filetypes*. Like `structs` in C, MPI's derived types and MPI-IO's etypes are constructed from simple base types such as integers or floats. Filetypes in turn are structured collections of etypes. Unlike `structs` or derived types, filetypes may contain *holes* as well as data. Using the filetype as a template, these holes allow applications to specify which pieces of data in a file are to be accessed and which are to be skipped over. When multiple nodes in an application access a file, they typically all share a common etype while each node has its own filetype, which indicates which portions of the file that node will access. Through the proper combination of etypes and holes, filetypes may be used to generate many of the same regular access patterns as the interfaces we presented above.

MPI-IO presents three advantages. First, rather than being specified in bytes, I/O is specified in terms of the same data types programmers use in their applications, eliminating the need to painstakingly calculate offsets into the file. Second, MPI-IO may well benefit from its association with MPI, which is becoming the dominant message-passing interface. Finally, MPI-IO offers the promise of providing a common interface to parallel I/O across many different platforms. The primary disadvantage of MPI-IO is its unfamiliarity, particularly to those programmers who are accustomed to Unix-like I/O. It remains to be seen whether or not this interface will be embraced by scientific programmers. Finally MPI-IO has yet to be fully specified or implemented, and it is possible that design decisions that look good on paper will not work in practice. It appears that MPI-IO could also feasibly be implemented on top of a nested-batched interface.

8.2.5 ELFS

The ELFS system, from the University of Virginia, is an object-oriented file system that has tight ties to the Mentat programming language [GP91, GL91]. Files in ELFS are instances of object classes, which provide a high-level interface to an abstract data structure and encapsulate the access patterns and the actual structure of the file. Each object has a separate thread of control,

allowing them to perform asynchronous data transfers as well as to prefetch and cache in the background. The class is responsible for matching the caching and prefetching algorithms to the higher-level semantics. Since applications manipulate files only via the external interface to the objects, applications are robust under architectural changes.

8.2.6 HFS

The Hurricane File System (HFS) is a part of the Hurricane operating system and was designed at the University of Toronto to run on the Hector shared-memory multiprocessor [KS93, Kri94, SVW+93]. While HFS shares our goal of providing a flexible, high-performance file system, it adopts an entirely different approach. HFS is based on a complex, highly-structured, object-oriented model. Files in HFS are referred to as *storage objects* and are made up of three components, one for each of three levels of the file system. Each component in turn may be composed of multiple sub-objects, each designed to add a specific functionality to the file (e.g., transparent replication of data, a caching algorithm, etc.). This file structure is both liberating and limiting. A file may have a nearly arbitrary set of properties simply by selecting an appropriate combination of sub-objects. To achieve this flexibility, each sub-object must fit exacting specifications to fulfill the assumptions of other sub-objects. Most applications interact with the file system through a user-level application library such as the *Alloc Stream Facility* [KSU94]. Note that the ideas underlying HFS could be used to build class libraries on top of Galley.

8.2.7 Whiptail

Whiptail is a file system developed for the Intel Paragon, and was built on top of Intel's PFS. It was designed to support libraries for out-of-core applications [SW95b, SWC+95], and it implements many of the recommendations for parallel I/O discussed in [CK93]. Whiptail provides no byte-level operations; all file operations must be done in units that are multiples of the block size. Although their target users have a different set of requirements than those observed in our two workload studies, Whiptail's implementors share our goal of providing high performance to libraries by avoiding unnecessary functionality at the file system level. Indeed, Whiptail's interface could be easily implemented on top of our file system, although that would negate some of the performance advantages offered by its low-level nature.

8.3 Higher-level Interfaces

There are numerous interfaces that are designed to allow programmers to describe their I/O needs at a higher semantic level. These interfaces are sometimes tightly integrated into a particular language such as HPF [Lov93, BGMZ92, HPF93] or CMF [Thi94]. There has also been great deal of research in designing libraries to support parallel I/O. The focus of much of this research has been the support of distributed matrices, particularly for data-parallel variants of Fortran [CBH+94, TBC+94, TG96]. Other libraries support multidimensional matrix I/O under C++ [SW94, SCJ+95, CWS+96]. There has also been quite a bit of work done in supporting irregular data structures: [CSBS94, PSC+95] discuss the issues relating to interprocessor distribution of these structures, and Jovian explores the issues relating to their persistent storage [BBS+94]. ViC* provides nearly-transparent support for out-of-core applications written in C*, a data-parallel dialect of C [CH96].

Chapter 9

Conclusion

During the course of this research, we have explored two related areas: how scientific applications use current parallel file systems, and how parallel file systems can be designed to better meet the needs of those applications.

Across the two machines and two programming models covered in our workload characterizations, we found important similarities and differences in the way applications use the different parallel file systems. Compared to uniprocessor workloads, the parallel workloads used much larger files, and were dominated by writes. Although there were variations in magnitude, we found small request sizes to be common in both the parallel workloads, just as they are in uniprocessor workloads. Compared to vector-supercomputer workloads, we observed much smaller requests and a tendency toward non-consecutive, but sequential file access. Finally, parallelism led to new, interleaved access patterns with high interprocess spatial locality at the I/O node. While some of the details of our results may be specific to the two systems we studied, or to the workloads at the two sites, we believe that the general conclusions above are widely applicable to scientific workloads running on loosely-coupled MIMD multiprocessors.

Based on the results of these workload characterization studies, we designed Galley, a new parallel file system that attempts to rectify some of the shortcomings of existing systems. Galley is based on a new three-dimensional structuring of files, which provides tremendous flexibility and control to applications and libraries. We showed how Galley's new data-access requests reduced the aggregate latency of multiple small requests and allowed the file system to optimize the disk accesses required to satisfy the request.

The results of our experiments indicate that our new style of interface increased performance by

up to several orders of magnitude. More importantly, this new interface allows high performance on access patterns that are known to be common in scientific applications, and that are known perform poorly on most current multiprocessor file systems.

Finally, by implementing two applications and a library, we have shown how Galley's features can be useful in practice. Furthermore, we have also shown how those features can lead to higher performance in practice.

9.1 Future Work

There is still a great deal of work to be done in studying existing parallel file systems and developing new systems.

9.1.1 Workload characterization

When there were no workload studies of multiprocessor file systems, file system designers were forced to rely on studies of uniprocessor systems. Now that we have performed two such studies, there is a danger that other groups will have less interest in performing additional studies. Since parallel machines and file systems are evolving rapidly, it is possible that future parallel applications will behave very differently than the applications we observed in our studies. Without continuing work in workload characterization, file system designers will soon be relying on out-of-date information again.

We were fortunate enough to have access to the source code for the two commercial file systems we studied, as well as system administrators willing to put our modified file systems into production use. To simplify the task of performing such characterization projects in the future, we would like to see vendors of commercial parallel file systems insert *hooks* into their systems, allowing third-party instrumentation to be performed without modifying the original file system code. These same hooks would likely prove useful for the development of debugging and performance analysis tools as well.

Finally, the results of these characterizations could be used to develop statistical models for multiprocessor file system workloads. These models could be used to reason about parallel algorithms as well as about the effects of changes in a system on the behavior of that system. These models could also be used to create synthetic workload generators, which would be useful for early

performance evaluations of new parallel file systems.

9.1.2 Galley

There are several areas of parallel file system design that should be explored in greater detail. We hope that researchers will be able to use the Galley Parallel File System to investigate some of these areas.

- Caching

  Caching has traditionally been used to improve a file system's performance by exploiting spatial locality and temporal locality. From our workload characterizations, we know that there is a great deal of spatial locality in parallel file system workloads, largely due to inter-processor sharing of file blocks. Given the large sizes of files in these workloads, however, there is less opportunity for temporal locality, since most files cannot fit entirely within the cache. Should the absence of temporal locality have any impact on the design of file system caches? Should applications be given more control over the way their files are cached at the IOPs?

- Application control

  Our new data-access interfaces allow applications to communicate more information to the file system than traditional interfaces. We found that this information led to opportunities for tremendous performance improvements. Is there more information that applications can provide to the file system to increase performance further? How should applications provide this information?

  One of the more radical methods of addressing this issue is to allow applications to "download" code to the IOP, where it would run in the same address space as the I/O server [KN96]. This approach is similar in spirit to the techniques suggested by the SPIN [BSP+95] and Exokernel [EKO95] projects, which involve uniprocessor operating systems. The ability to run application code at the IOPs could be used to allow applications to perform data-dependent filtering or distribution, possibly reducing network traffic. This ability could also be used to implement application-defined caching and prefetching policies.

- Multiple applications

  Most descriptions of sequential and parallel file systems, including our own, have focused on providing high performance to individual applications. We have seen that in practice, however, it is common for file systems to serve multiple applications at once. A complex, but important, area for future research is the efficient support of these common, multiple application workloads. Among the questions that need to be answered are: How do we define fair service? How do we provide fair service to each application? How do we balance attempts to provide fair service to multiple applications with potentially conflicting per-application optimizations?

Finally, there a number of important, but less research-oriented, areas for further work on Galley.

- Naming

  The flat namespace provided by Galley is sufficient for a prototype or a research system. If Galley is every to become more generally useful, however, a more sophisticated naming system will be required. Ideally, Galley should support a full hierarchical naming system like that provided by Unix.

- Security

  Galley does not provide any security at all. Clearly, this limitation would not be acceptable in most production environments.

- Reliability

  Galley is currently designed to deliver the best performance the underlying hardware can provide. As we discussed in Chapter 6, we are frequently able to achieve that goal. To achieve this high performance, we have sacrificed some level of reliability. In particular, when writing, the DiskManager always schedules disk accesses in such a way as to avoid any excess seeks, without regard for what is being written. This scheduling can cause a fork's mapping blocks to be written to disk before the data blocks. If the system crashes before the data blocks can be written, the mapping blocks will be intact, but will point to "bad" data. Indeed,

it is likely that the fork will actually contain data from a fork that had been deleted, which clearly raises security concerns. It should be possible to improve reliability without greatly sacrificing performance by implementing a scheme such as soft metadata updates [GP94].

- Heterogeneity

  The current implementation of Galley does not support a heterogeneous environment. The assumption that the CPs and IOPs all use the same basic data representation pervades the implementation. While this limitation may be reasonable for a file system designed to run on a multiprocessor such as the SP-2, where all the nodes are the same, it is likely to be unacceptable in a file system designed for a cluster-of-workstations environment.

- Node colocation

  All communication between CPs and IOPs takes place over TCP/IP. If a CP and an IOP happen to reside on the same node, this heavy-weight communication stack could likely be replaced by Unix pipes or shared memory for higher performance.

- Integration

  Another serious limitation of Galley is its isolated nature. Applications can easily create and use files on Galley, but there is no standard way to migrate files between Galley and an external file system. To be truly useful in a production setting, Galley must be integrated with the existing environment.

  While Galley does not, and likely never will, support full Unix semantics, a reasonable compromise solution to this problem is to provide an NFS interface for Galley, which would allow standard workstations to "mount" a Galley file system as they would any remote file system. In this type of situation, Galley's files would look like immutable directories containing subfiles, and its subfiles would look like directories containing forks. Forks could be accessed as standard Unix files.

  The biggest drawback to this simple solution is that the resulting representation of a Galley file is not likely to correspond to the model envisioned by the creator of the file. For example, our Linear File Model library stripes the data of a linear file across many subfiles. If the file system were exported by NFS, that linearity would not be apparent to external applications.

A possible solution to this subproblem would be to allow user-defined export methods for files, possibly through the use of "downloaded" code, as discussed above.

9.2 Availability

The full source code for Galley is available at
`http://www.cs.dartmouth.edu/~nils/galley.html`.

Bibliography

[Are91] James W. Arendt. Parallel genome sequence comparison using a concurrent file system. Technical Report UIUCDCS-R-91-1674, University of Illinois at Urbana-Champaign, 1991.

[BBD+94] Brian Bershad, David Black, David DeWitt, Garth Gibson, Kai Li, Larry Peterson, and Marc Snir. Operating system support for high-performance parallel I/O systems, 1994. Scalable I/O Initiative Working Paper Number 4.

[BBDS92] D. H. Bailey, E. Barszcz, L. Dagum, and H. D. Simon. NAS parallel benchmark results. In *Proceedings of Supercomputing '92*, pages 386–393, 1992.

[BBH95] Sandra Johnson Baylor, Caroline B. Benveniste, and Yarson Hsu. Performance evaluation of a parallel I/O architecture. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 404–413, Barcelona, July 1995.

[BBS+94] Robert Bennett, Kelvin Bryant, Alan Sussman, Raja Das, and Joel Saltz. 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.

[BGMZ92] Peter Brezany, Michael Gernt, Piyush Mehotra, and Hans Zima. Concurrent file operations in a High Performance FORTRAN. In *Proceedings of Supercomputing '92*, pages 230–237, 1992.

[BGST93] Michael L. Best, Adam Greenberg, Craig Stanfill, and Lewis W. Tucker. CMMD I/O: A parallel Unix I/O. In *Proceedings of the Seventh International Parallel Processing Symposium*, pages 489–495, 1993.

[BHK+91] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 198–212, 1991.

[BHS+95] D. H. Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, December 1995.

[BSP+95] Brian Bershad, Stefan Savage, Przemysław Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety

and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 267–284, December 1995.

[BW96] Sandra Johnson Baylor and C. Eric Wu. Parallel I/O workload characteristics using Vesta. In Jain et al. [JWB96], chapter 7, pages 167–185.

[CACR95] Phyllis E. Crandall, Ruth A. Aydt, Andrew A. Chien, and Daniel A. Reed. Input/output characteristics of scalable parallel applications. In *Proceedings of Supercomputing '95*, December 1995.

[CBF93] Peter F. Corbett, Sandra Johnson Baylor, and Dror G. Feitelson. Overview of the Vesta parallel file system. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 1–16, 1993.

[CBH+94] Alok Choudhary, Rajesh Bordawekar, Michael Harry, Rakesh Krishnaiyer, Ravi Ponnusamy, Tarvinder Singh, and Rajeev Thakur. Passion: Parallel and scalable software for input-output. Technical Report SCCS-636, ECE Dept., NPAC and CASE Center, Syracuse University, September 1994.

[CC94] Thomas H. Cormen and Alex Colvin. ViC*: A preprocessor for virtual-memory C*. Technical Report PCS-TR94-243, Dept. of Computer Science, Dartmouth College, November 1994.

[CCFN92] Russell Carter, Bob Ciotti, Sam Fineberg, and Bill Nitzberg. NHT-1 I/O benchmarks. Technical Report RND-92-016, NAS Systems Division, NASA Ames, November 1992.

[CF96] Peter F. Corbett and Dror G. Feitelson. The Vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3):225–264, August 1996.

[CFP+95] Peter F. Corbett, Dror G. Feitelson, Jean-Pierre Prost, George S. Almasi, Sandra Johnson Baylor, Anthony S. Bolmarcich, Yarsun Hsu, Julian Satran, Marc Snir, Robert Colao, Brian Herr, Joseph Kavaky, Thomas R. Morgan, and Anthony Zlotek. Parallel file systems for the IBM SP computers. *IBM Systems Journal*, 34(2):222–248, January 1995.

[CH96] Thomas H. Cormen and Melissa Hirschl. Early experiences in evaluating the parallel disk model with the ViC* implementation. Technical Report PCS-TR96-293, Dept. of Computer Science, Dartmouth College, August 1996. To appear in *Parallel Computing*.

[CHKM93] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina. Architectural requirements of parallel scientific applications with explicit communication. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 2–13, 1993.

[CK93] Thomas H. Cormen and David Kotz. Integrating theory and practice in parallel file systems. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 64–74, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies. Revised as Dartmouth PCS-TR93-188 on 9/20/94.

[CPD+96] Peter F. Corbett, Jean-Pierre Prost, Chris Demetriou, Garth Gibson, Erik Reidel, Jim Zelenka, Yuqun Chen, Ed Felten, Kai Li, John Hartman, Larry Peterson, Brian Bershad, Alec Wolman, and Ruth Aydt. Proposal for a common parallel file system programming interface. WWW http://www.cs.arizona.edu/sio/api1.0.ps, September 1996. Version 1.0.

[Cro89] Thomas W. Crockett. File concepts for parallel I/O. In *Proceedings of Supercomputing '89*, pages 574–579, 1989.

[CSBS94] Frederick T. Chong, Shamik D. Sharma, Eric Brewer, and Joel Saltz. Multiprocessor runtime support for fine-grained, irregular DAGs. Technical Report CS-TR-3266, Department of Computer Science, University of Maryland, March 1994.

[CWS+96] Y. Chen, M. Winslett, K. E. Seamons, S. Kuo, Y. Cho, and M. Subramaniam. Scalable message passing in Panda. In *Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 109–121, Philadelphia, May 1996.

[dC94] Juan Miguel del Rosario and Alok Choudhary. High performance I/O for parallel computers: Problems and prospects. *IEEE Computer*, 27(3):59–68, March 1994.

[Dib90] Peter C. Dibble. *A Parallel Interleaved File System.* PhD thesis, University of Rochester, March 1990.

[DJ93] Erik P. DeBenedictis and Stephen C. Johnson. Extending Unix for scalable computing. *IEEE Computer*, 26(11):43–53, November 1993.

[DSE88] Peter Dibble, Michael Scott, and Carla Ellis. Bridge: A high-performance file system for parallel processors. In *Proceedings of the Eighth International Conference on Distributed Computer Systems*, pages 154–161, June 1988.

[EK93] Rüdiger Esser and Renate Knecht. Intel Paragon XP/S — architecture and software environment. Technical Report KFA-ZAM-IB-9305, Central Institute for Applied Mathematics, Research Center Jülich, Germany, April 26 1993.

[EKO95] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 251–266, December 1995.

[FCHP95] Dror G. Feitelson, Peter F. Corbett, Yarson Hsu, and Jean-Pierre Prost. Parallel I/O systems and interfaces for parallel computers. In *Multiprocessor Systems — Design and Integration.* World Scientific, 1995.

[FE89] Richard Allen Floyd and Carla Schlatter Ellis. Directory reference patterns in hierarchical file systems. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):238–247, June 1989.

[Flo86] Rick Floyd. Short-term file reference patterns in a UNIX environment. Technical Report 177, Dept. of Computer Science, Univ. of Rochester, March 1986.

[FPD93] James C. French, Terrence W. Pratt, and Mriganka Das. Performance measurement of the Concurrent File System of the Intel iPSC/2 hypercube. *Journal of Parallel and Distributed Computing*, 17(1–2):115–121, January and February 1993.

[GGL93] N. Galbreath, W. Gropp, and D. Levine. Applications-driven parallel I/O. In *Proceedings of Supercomputing '93*, pages 462–471, 1993.

[GL91] Andrew S. Grimshaw and Edmond C. Loyot, Jr. ELFS: object-oriented extensible file systems. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, page 177, 1991.

[GP91] Andrew S. Grimshaw and Jeff Prem. High performance parallel file objects. In *Sixth Annual Distributed-Memory Computer Conference*, pages 720–723, 1991.

[GP94] Gregory R. Ganger and Yale N. Patt. Metadata update performance in file systems. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 49–60, 1994.

[Gro96] NAS Scientific Consulting Group. Hitchhiker's guide to using the NAS SP2, 1996.

[GSC+95] Garth A. Gibson, Daniel Stodolsky, Pay W. Chang, William V. Courtright II, Chris G. Demetriou, Eka Ginting, Mark Holland, Qingming Ma, LeAnn Neal, R. Hugo Patterson, Jiawen Su, Rachad Youssef, and Jim Zelenka. The Scotch parallel storage system. In *Proceedings of 40th IEEE Computer Society International Conference (COMPCON 95)*, pages 403–410, San Francisco, Spring 1995.

[HER+95] Jay Huber, Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal. PPFS: A high performance portable parallel file system. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 385–394, Barcelona, July 1995.

[HP91] Hewlett Packard. *HP97556/58/60 5.25-inch SCSI Disk Drives Technical Reference Manual*, second edition, June 1991. HP Part number 5960-0115.

[HPF93] High Performance Fortran Forum. *High Performance Fortran Language Specification*, 1.0 edition, May 3 1993.

[IBM94] IBM. *AIX Version 3.2 General Programming Concepts*, twelfth edition, October 1994.

[JWB96] Ravi Jain, John Werth, and James C. Browne, editors. *Input/Output in Parallel and Distributed Computer Systems*. Kluwer Academic Publishers, 1996.

[KE93a] David Kotz and Carla Schlatter Ellis. Caching and writeback policies in parallel file systems. *Journal of Parallel and Distributed Computing*, 17(1–2):140–145, January and February 1993.

[KE93b] David Kotz and Carla Schlatter Ellis. Practical prefetching techniques for multiprocessor file systems. *Journal of Distributed and Parallel Databases*, 1(1):33–51, January 1993.

[KFG94] John F. Karpovich, James C. French, and Andrew S. Grimshaw. High performance access to radio astronomy data: A case study. In *Proceedings of the 7th International Working Conference on Scientific and Statistical Database Management*, pages 240–249, September 1994. Also available as UVA TR CS-94-25.

[KGF93] John F. Karpovich, Andrew S. Grimshaw, and James C. French. Breaking the I/O bottleneck at the National Radio Astronomy Observatory (NRAO). Technical Report CS-94-37, University of Virginia, August 1993.

[Kim86] Michelle Y. Kim. Synchronized disk interleaving. *IEEE Transactions on Computers*, C-35(11):978–988, November 1986.

[KN96] David Kotz and Nils Nieuwejaar. Flexibility and performance of parallel file systems. *ACM Operating Systems Review*, 30(2):63–73, April 1996.

[KR94] Thomas T. Kwan and Daniel A. Reed. Performance of the CM-5 scalable file system. In *Proceedings of the 8th ACM International Conference on Supercomputing*, pages 156–165, July 1994.

[Kri94] Orran Krieger. *HFS: A flexible file system for shared-memory multiprocessors.* PhD thesis, University of Toronto, October 1994.

[KS93] Orran Krieger and Michael Stumm. HFS: a flexible file system for large-scale multiprocessors. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 6–14, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies.

[KSU94] Orran Krieger, Michael Stumm, and Ronald Unrau. The Alloc Stream Facility: A redesign of application-level stream I/O. *IEEE Computer*, 27(3):75–82, March 1994.

[KTR94] David Kotz, Song Bac Toh, and Sriram Radhakrishnan. A detailed simulation model of the HP 97560 disk drive. Technical Report PCS-TR94-220, Dept. of Computer Science, Dartmouth College, July 1994.

[LIN+93] Susan J. LoVerso, Marshall Isman, Andy Nanopoulos, William Nesheim, Ewan D. Milne, and Richard Wheeler. *sfs*: A parallel file system for the CM-5. In *Proceedings of the 1993 Summer USENIX Conference*, pages 291–305, 1993.

[LMKQ89] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley, 1989.

[Lov93] David B. Loveman. High Performance Fortran. *IEEE Parallel and Distributed Technology*, 1(1):25–42, February 1993.

[MHQ96] Jason A. Moore, Phil Hatcher, and Michael J. Quinn. Efficient data-parallel files via automatic mode detection. In *Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 1–14, Philadelphia, May 1996.

[MJLF84] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[MK91] Ethan L. Miller and Randy H. Katz. Input/output behavior of supercomputer applications. In *Proceedings of Supercomputing '91*, pages 567–576, November 1991.

[MK93] Ethan L. Miller and Randy H. Katz. An analysis of file migration in a UNIX supercomputing environment. In *Proceedings of the 1993 Winter USENIX Conference*, pages 421–434, January 1993.

[MMRW94] Arthur B. Maccabe, Kevin S. McCurley, Rolf Riesen, and Stephen R. Wheat. SUNMOS for the Intel Paragon: A brief user's guide. In *Proceedings of the Intel Supercomputer Users Group Conference*, pages 245–251, June 1994.

[MPI94] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, 1.0 edition, May 5 1994.

[NAS93] NASA Ames Research Center, Moffet Field, CA. *NAS User Guide*, 6.1 edition, March 1993.

[NAS94] NASA/Science Office of Standards and Technology, NASA Goddard Space Flight Center, Greenbelt, MD 020771. *A User's Guide for the Flexible Image Transport System (FITS)*, 3.1 edition, May 1994.

[Nit92] Bill Nitzberg. Performance of the iPSC/860 Concurrent File System. Technical Report RND-92-020, NAS Systems Division, NASA Ames, December 1992.

[NK96] Nils Nieuwejaar and David Kotz. Performance of the Galley parallel file system. In *Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 83–94, May 1996.

[OCH+85] John Ousterhout, Hervé Da Costa, David Harrison, John Kunze, Mike Kupfer, and James Thompson. A trace driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 15–24, December 1985.

[PEK+95] Apratim Purakayastha, Carla Schlatter Ellis, David Kotz, Nils Nieuwejaar, and Michael Best. Characterizing parallel file-access patterns on a large-scale multiprocessor. In *Proceedings of the Ninth International Parallel Processing Symposium*, pages 165–172, April 1995.

[PEK96] Apratim Purakayastha, Carla Schlatter Ellis, and David Kotz. ENWRICH: a compute-processor write caching scheme for parallel file systems. In *Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 55–68, May 1996.

[PFDJ89] Terrence W. Pratt, James C. French, Phillip M. Dickens, and Stanley A. Janet, Jr. A comparison of the architecture and performance of two parallel file systems. In *Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 161–166, 1989.

[PGK88] David Patterson, Garth Gibson, and Randy Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD Conference*, pages 109–116, June 1988.

[Pie89] Paul Pierce. A concurrent file system for a highly parallel mass storage system. In *Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 155–160, 1989.

[Pow77] Michael L. Powell. The DEMOS File System. In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, pages 33–42, November 1977.

[PP93] Barbara K. Pasquale and George C. Polyzos. A static analysis of I/O characteristics of scientific applications in a production workload. In *Proceedings of Supercomputing '93*, pages 388–397, 1993.

[PP94a] Barbara K. Pasquale and George C. Polyzos. A case study of a scientific application I/O behavior. In *Proceedings of the International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 101–106, 1994.

[PP94b] Barbara K. Pasquale and George C. Polyzos. Dynamic I/O characterization of I/O-intensive scientific applications. In *Proceedings of Supercomputing '94*, pages 660–669, November 1994.

[PSC+95] Ravi Ponnusamy, Joel Saltz, Alok Choudhary, Yuan-Shin Hwang, and Geoffrey Fox. Runtime support and compilation methods for user-specified data distributions. *IEEE Transactions on Parallel and Distributed Systems*, 6(8):815–831, August 1995.

[Pur96] Apratim Purakayastha. *Characterizing and Optimizing Parallel File Systems.* PhD thesis, Dept. of Computer Science, Duke University, Durham, NC, June 1996. Also available as technical report CS-1996-10.

[RB90] A. L. Narasimha Reddy and Prithviraj Banerjee. A study of I/O behavior of Perfect benchmarks on a multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 312–321, 1990.

[RBK92] K. K. Ramakrishnan, P. Biswas, and Ramakrishna Karedla. Analysis of file I/O traces in commercial computing environments. In *Proceedings of ACM SIGMETRICS and PERFORMANCE '92*, pages 78–90, 1992.

[Roy93] Paul J. Roy. Unix file access and caching in a multicomputer environment. In *Proceedings of the Usenix Mach III Symposium*, pages 21–37, 1993.

[RP95] Brad Rullman and David Payne. An efficient file I/O interface for parallel applications. DRAFT presented at the Workshop on Scalable I/O, Frontiers '95, February 1995.

[RT78] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *The Bell System Technical Journal*, 6(2):1905–1930, July-August 1978.

[RW94] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, March 1994.

[SCJ+95] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95*, December 1995.

[SCO90] Margo Seltzer, Peter Chen, and John Ousterhout. Disk scheduling revisited. In *Proceedings of the 1990 Winter USENIX Conference*, pages 313–324, 1990.

[SGM86] Kenneth Salem and Hector Garcia-Molina. Disk striping. In *IEEE 1986 Conference on Data Engineering*, pages 336–342, 1986.

[SVW+93] Michael Stumm, Zvonko Vranesic, Ron White, Ronald Unrau, and Keith Farkas. Experiences with the Hector multiprocessor. In *Proceedings of the Parallel Systems Fair at the International Parallel Processing Symposium*, pages 10–17, 1993.

[SW94] K. E. Seamons and M. Winslett. An efficient abstract interface for multidimensional array I/O. In *Proceedings of Supercomputing '94*, pages 650–659, November 1994.

[SW95a] K. E. Seamons and M. Winslett. A data management approach for handling large compressed arrays in high performance computing. In *Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 119–128, February 1995.

[SW95b] Elizabeth A. M. Shriver and Leonard F. Wisniewski. An API for choreographing data accesses. Technical Report PCS–TR95–267, Dartmouth College Department of Computer Science, October 1995.

[SWC+95] Elizabeth A. M. Shriver, Leonard F. Wisniewski, Bruce G. Calder, David Greenberg, Ryan Moore, and David Womble. Parallel disk access using the Whiptail File System: Design and implementation. Unpublished Manuscript, 1995.

[TBC+94] Rajeev Thakur, Rajesh Bordawekar, Alok Choudhary, Ravi Ponnusamy, and Tarvinder Singh. PASSION runtime library for parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 119–128, October 1994.

[TG96] Sivan Toledo and Fred G. Gustavson. The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computations. In *Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 28–40, Philadelphia, May 1996.

[The96] The MPI-IO Committee. MPI-IO: a parallel file I/O interface for MPI, April 1996. Version 0.5.

[Thi94] Thinking Machines Corporation, Cambridge, Mass. *CM Fortran User's Guide*, 2.1 edition, January 1994.

[Tho78] K. Thompson. UNIX implementation. *The Bell System Technical Journal*, 6(2):1931–1946, July-August 1978.

[Tho96] Joel T. Thomas. The Panda array I/O library on the Galley parallel file system. Technical Report PCS-TR96-288, Dept. of Computer Science, Dartmouth College, June 1996. Senior Honors Thesis.

[vdW93] Rob van der Wijngaart. Efficient implementation of a 3-dimensional ADI Method on the iPSC/860. In *Supercomputing '93*, pages 102–111, Portland, OR, November 1993.

[WMR+94] Stephen R. Wheat, Arthur B. Maccabe, Rolf Riesen, David W. van Dresser, and T. Mack Stallcup. PUMA: An operating system for massively parallel systems. In *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences*, pages 56–65, 1994.

[ZRB+93] Roman Zajcew, Paul Roy, David Black, Chris Peak, Paulo Guedes, Bradford Kemp, John LoVerso, Michael Leibensperger, Michael Barnett, FaraMarz Rabii, and Durriya Netterwala. An OSF/1 UNIX for massively parallel multicomputers. In *Proceedings of the 1993 Winter USENIX Conference*, pages 449–468, January 1993.