A Parallel Interleaved File System

by

Peter C. Dibble

Submitted in Partial Fulfillment

of the

Requirements for the Degree

Doctor of Philosophy

Supervised by Michael L. Scott

Department of Computer Science

University of Rochester

Rochester, New York

March 1990

Curriculum Vitae

Peter C. Dibble was born in Waterbury Connecticut early in the morning on December 18th 1953. He went to sleep soon thereafter, setting a persistent pattern. Not many months later he disassembled the family typewriter (which never again typed right), and made a valiant attempt to dissect the vacuum cleaner.

He attended Salem School, McTernan, Taft, and University of Connecticut and earned a BS in Chemistry which went unused when he took a job at a computer service bureau. After a sequence of jobs in the computer field and much part-time education, he returned to school full-time at the University of Rochester in 1984. There he received a Masters degree in 1986 and continued toward a PhD. His interest in parallel algorithms lead him to do research on parallel operating system topics. His background in large computers and his impatience with typical I/O performance lead him to the area of this dissertation.

Acknowledgements

First, of course, I would like to thank my advisor, Michael Scott, for his advice and support. Numerous times he helped me back to a fruitfull path when I was stuck at a dead end. The other members of my committee, Bruce Arden, Robert Fowler, and Tom LeBlanc have always been there when I needed them.

On the first day of the graduate problem seminar, Jerry Feldman told us that a good class of graduate students would get to know one another very well. We did, and the time I spent with my classmates gathered around a keyboard, pizza, blackboard, or piano are among my happiest memories of the past five years.

Liudvikas Bukys kept the Butterfly running. That was not always an easy job. The department secretaries kept everything else operating, and that wasn't easy either.

Carla Ellis was my advisor when I started work on Bridge. She helped me initiate this research and handed me off to Michael with a good start.

Catherine faithfully supported, criticized, and tolerated this work for years.

Reaching back into the past, Ed North and Domina Spencer taught me to know and love science and mathematics. My family put up with the inconvenience of a son or brother who wanted to grow up to be a scientist, and supported me through everything.

Jeffry K. Price at Arrow Electronics sent me stacks of information about disk drive performance, and patiently sent it all over again several times as products evolved. Bill Hasset at IBM's Rochester office did similar service for IBM's I/O equipment.

This work was supported in part by the U.S. Army Engineering Topographic Laboratories research contract number DACA76-85-C-0001, NSF/CER research grant number DCR-8320136, and a grant from the IBM Corporation. Microware Systems Corporation also provided financial support.

Abstract

A computer system is most useful when it has well-balanced processor and I/O performance. Parallel architectures allow fast computers to be constructed from unsophisticated hardware. The usefulness of these machines is severely limited unless they are fitted with I/O subsystems that match their CPU performance.

Most parallel computers have insufficient I/O performance, or use exotic hardware to force enough I/O bandwidth through a uniprocessor file system. This approach is only useful for small numbers of processors. Even a modestly parallel computer cannot be served by an ordinary file system. Only a parallel file system can scale with the processor hardware to meet the I/O demands of a parallel computer.

This dissertation introduces the concept of a parallel interleaved file system. This class of file system incorporates three concepts: parallelism, interleaving, and *tools.* Parallelism appears as a characteristic of the file system program and in the disk hardware. The parallel file system software and hardware allows the file system to scale with the other components of a multiprocessor computer. Interleaving is the rule the file system uses to distribute data among the processors. Interleaved record distribution is the simplest and in many ways the best algorithm for allocating records to processors. Tools are application code that can enter the file system at a level that exposes the parallel structure of the files. In many cases tools decrease interprocessor communication by moving processing to the data instead of moving the data.

The thesis of this dissertation is that a parallel interleaved file system will provide scalable high-performance I/O for a wide range of parallel architectures while supporting a comprehensive set of conventional file system facilities. We have confirmed our performance claims experimentally and theoretically. Our experiments show practically linear speedup to the limits of our hardware for file copy, file sort, and matrix transpose on an array of bits stored in a file. Our analysis predicts the measured results and supports a claim that the file system will easily scale to more than 128 processors with disk drives.

Table of Contents

List of Figures

List of Tables

1 Introduction

This dissertation describes the design, implementation, and evaluation of a parallel interleaved file system (PIFS). The file system is designed to address the I/O problems of large, general-purpose, parallel computers—chiefly problems of performance and scalability.

A uniprocessor file system is inherently not scalable. Techniques that increase the performance of a uniprocessor file system may bring it into balance with a small multiprocessor computer, but a sufficiently parallel multiprocessor computer will require more I/O than any uniprocessor file system can provide.

A parallel interleaved file system runs as a parallel program in the same hardware environment as application programs. It can therefore scale as applications scale with the hardware parallelism. Files are interleaved across the processors participating in the file system; this structure gives a peripheral I/O data rate proportional to the parallelism of the file system.

A practical file system must meet some usability requirements. Most important is compatibility with traditional file systems. Our PIFS design addresses this requirement directly by supporting the traditional interface. The strictly traditional file system interface has no explicit parallelism and severely limited implicit parallelism so we enhance the file system interface with mechanisms that express parallelism. Software devices we call *tools* are our primary mechanism for parallel file operations. Depending on the security restrictions on tools they can be implemented as dynamically-linked high-level operations for the file system, or as specially-coded blocks of application code that can be co-located with the pieces of the file system. Tools can move processor activity to the data in the file system; for many tasks, this facility greatly reduces interprocessor communication.

My thesis is that a parallel interleaved file system will provide scalable high-performance I/O for a wide range of parallel architectures while supporting a comprehensive set of conventional file system facilities. I support this thesis with a detailed discussion of the practicality of a parallel interleaved file system, with a proof-of-concept implementation, and with measurements and analysis that show excellent scalability.

1.1 Motivation

A large multi-processor computer cannot be served adequately by a conventional uniprocessor file system. The speed of a large collection of processors can completely overwhelm the processor running the file system.

Consider pattern matching. At ten instructions per byte scanned, even a very modest microprocessor will require about 30k bytes per second of input. A 128-processor system will consume data at about 3.66 megabytes per second. A high-performance disk can certainly deliver that data rate and the file system on a conventional supercomputer can easily handle ten megabytes per second, but most parallel computers do not run their file system on a supercomputer. File systems for the high-performance microprocessors that are frequently used for nodes in parallel computers only sustain a data rate of a few hundred kilobytes per second [Mallett and Smith, 1989; Rinko-Gay and Varhol, 1989]. Even for the moderately processor-intensive pattern matching operation, 128 processors could outrun a file system by a factor of about 20.

It is easy to find problems that are dominated by I/O time even on a uniprocessor. Copying a file, for instance, involves on the order of ten machine instructions at the application level and 2000 instructions of system overhead per block copied. This CPU activity would take on the order of a millisecond. A fast file system might accomplish a read and a write in about 10 milliseconds. The I/O system would need a factor of 10 performance improvement and its own processor to let such a copy program run without waiting for I/O. A multiprocessor aggravates this I/O problem by increasing the computational power without improving I/O throughput.

Currently available parallel processor architectures couple tremendous CPU power with ordinary uniprocessor file system performance. A solution to the I/O problem based on parallelism will scale better than any solution that improves I/O performance without using parallelism because a file system that does not use parallelism will eventually hit some bottleneck that the parallel part of the system does not encounter. An I/O system that uses the same parallel technology as the rest of the system is likely to scale with the system.

1.1.1 Performance

The performance of a parallel system will improve as the number of parallel components increases until it reaches a degree of concurrency where some component of the system cannot benefit from further parallelism. That component constitutes a serial bottleneck, and it limits the scalability of the system.

An activity with multiple concurrent components is almost certain to have a bottleneck somewhere. A bottleneck is a symptom of poor performance balance, and it can be treated by improving the performance of the weak component. For instance, memory interleaving has been used to reduce the bottleneck between a cache and main memory by increasing the parallelism of the memory component of a computer system.

Not all performance problems are bottlenecks. A system's worst performance problem may be poor unsigned multiplication, but that is not a *bottleneck* unless some other part of the system waits while multiplication takes place. If an asynchronous coprocessor executes unsigned multiplications while the rest of the processor performs other useful work, then multiplications may be a bottleneck. If multiplications are executed in sequence with other instructions, they may be a performance problem, but not a bottleneck.

For thirty years computers have included facilities for I/O concurrent with other activities. I/O is, therefore, a much-studied bottleneck. A good file system will aim to have data ready so a program can read without waiting for the disk. Ideally, the file system would run exactly at capacity and let the processor run without pausing to wait for input. If the program waited for the disk, the program would be suffering from an I/O bottleneck.[1] The mismatch between disk speed and processor speed is usually accommodated with a multiprogramming operating system. Since other programs run while a program waits for I/O, and many programs can have I/O in progress concurrently, the system as a whole need not wait for I/O. Individual applications suffer an I/O bottleneck, but the system as a whole may be in balance.

A multiprogramming operating system is designed to fully utilize the computer's resources, not to minimize the run time of individual jobs. No amount of task switching will let a job with two hours of I/O time complete in less than two hours, and large multiprocessors are likely to be dedicated frequently to large jobs with lots of I/O.

As data moves from a disk drive to its destination, it passes through a number of potential bottlenecks. The disk drive (or other storage device) is at the beginning of the path, and it is the first potential source of an I/O bottleneck. The data moves from the drive along a cable to a controller. The cable and controller are possible bottlenecks. (The controller may actually be a sequence of controllers, but consider all the controllers one bottleneck.) From the controller the data moves through some connection into memory; the connection and the memory constitute two more possible bottlenecks. The file system software may operate on the data; in any case it takes note of the I/O operation. The file system itself is therefore a potential bottleneck. Finally, the data is delivered from the file system to the application. If that delivery involves moving the data, the medium through which it moves is another potential bottleneck.

[1]It is not usually an issue, but if the disk is not able to run constantly because the processor cannot use the data fast enough, it is suffering from a processor bottleneck.

Table 1.1: Potential I/O Bottlenecks

Bottleneck	**Approaches**
Processor	Algorithm improvements, processor technology, and parallelism
File System	Algorithm improvements, processor technology, and parallelism
Memory	Memory technology, caching, and parallelism
Controller	Hardware technology, and parallelism
Link	Communication technology, and parallelism
Device	Storage technology, caching, and parallelism

Device

The typical disk drive is about twice as fast now as it was ten years ago.[2] This change is not nearly enough to keep pace with processor performance, which has improved by about a factor of ten during the same interval.

Although the performance of typical disk drives has improved slowly, exotic devices have evolved to meet the requirements of the supercomputer community. A wide variety of very fast storage systems is available. Many of these use parallelism in the storage device:

- Parallel transfer disks [CDC, 1986; Gamerl, 1987] read or write several disk heads at a time. This improves the transfer rate linearly with the number of heads used. It does not improve seek or latency times, but it does not worsen them either. Parallel transfer disks require exotic electronics to synchronize the data from their heads. They are expensive.

- Storage arrays [Manuel and Barney, 1986; Masters, 1987] use multiple inexpensive disks. They usually use about eight disks and give about eight times the data rate of a single disk. A storage array's seek time is only slightly worse than a single disk.[3] If the array uses disks whose rotation can be synchronized, the rotational latency time is the same as a single disk. If the disks are not synchronized it may be almost twice the time for one disk. (Eight disks will average eight-ninths of a rotation

[2]Ten years ago the typical disk drive had about a 30 millisecond seek time and a two megabyte per second transfer rate. Now it has a 15 millisecond seek time and ordinary transfer rates may be as high as four megabytes per second.

[3]The seek time for a storage array is the worst seek time for any disk in the array. Since the disks are all the same make and model, the worst seek time in the array will be very close to the best time.

of latency.) A failure in a simple storage array is a failure of the entire device. As disks are added the probability of failure increases. The RAID project at Berkeley [Patterson *et al.*, 1988] is addressing this issue through redundancy.

Link

The physical connection between devices and controllers can become a bottleneck, but it is not a likely one. The connection is typically a group of wires. A SCSI connection is a ribbon cable carrying eight-bit wide data, and control and ground wires. An Ethernet connection is a single coaxial cable. An IBM channel connection (bus and tag) is a bundle of coaxial cables. A link can be widened by adding parallelism in the form of additional connections, or the performance of each connection can be improved (e.g., by moving from twisted pair to coax or fiber optics).

The bandwidth of the link will only be a bottleneck under unusual circumstances. If file servers are seen as devices, the network connection between the file server and the user can easily be a bottleneck. There might also be cases where the device is located very far from the controller. The tradeoffs that make it easy to upgrade a six foot cable do not necessarily apply to a three-thousand mile link; e.g., a microwave link to a device in a distant part of the country.[4] This is, however, an unusual circumstance. Links are generally easy to upgrade to any desired specifications.

Controller

A bottleneck at the controller is also often attacked with parallelism. The transfer rate for a single file can be increased by ***striping*** [Salem and Garcia-Molina, 1984] the file across multiple drives and putting the drives on separate controllers. A striped file system treates several disks as one logical disk. It spreads files across the disks such that consecutive sectors are stored on consecutive disk drives. The file system issues I/O commands for all the striped disks concurrently. The disks run in parallel, and feed separate controllers. With n-way striping, a file system can achieve data rates n times the rate of a single drive. Striping is inexpensive and flexible, but it suffers from the performance and reliability problems of storage arrays. It is quite common, being used by such vendors as Cray [cray, 1988] and Pyramid [pyramid].

Striping is limited by reliability concerns, software overhead, the number of controllers a system can accommodate, and the bandwidth between the controllers and memory.

[4]Extremely remote devices are not fantasy. A device called the Hyperchannel can carry an IBM channel attachment across a satellite link.

Memory

The effective speed of memory can be increased by increasing its width and interleaving access to the memory. This technique is often used in conjunction with caching, but cached memory without interleaving is of limited use for I/O if the I/O hardware uses DMA controllers—a reasonable assumption for a high-performance system. Programmed I/O (in which the main processor moves data between the device controller and main memory) does not suffer from the memory bottleneck, but only because of a more severe bottleneck in the component of the file system that moves data from the controller to memory.

File System

A combination of the mechanisms discussed above can deliver data to the file system as fast as the system memory can accept it. For machines in the super-computer class, this rate may exceed 100 megabytes per second. Since processors can usually access memory at least as fast as I/O controllers, the data rate up to memory will roughly match the processor speed.

A minimal file system will add little overhead to I/O it supervises, but its performance will probably be unacceptable for a high-performance computer. If the file system uses DMA hardware, a few hundred instructions to start an operation and another few hundred to return status to the application are sufficient. Such a minimal file system would be unlikely to become a I/O bottleneck, but the low-overhead strategy is unlikely to deliver good throughput. It leaves no room for I/O scheduling, data caching, or compression.[5]

A high performance file system will use processor power and memory to decrease disk activity. Caching, prefetching, write behind, and data compression accomplish this task without changing the I/O configuration. These techniques scale with the performance of the processor, but they use enough processor resources to make the file system software into a probable I/O bottleneck.

File system performance only scales with processor performance for a uniprocessor system. When processors are added to a multiprocessor system the requirement for I/O is increased and the load on shared resources is increased, but the file system runs no faster. If the processor uses shared resources the file system may even run a little slower than it would in isolation.

[5]Compression is not usually a feature of a general-purpose file system. It makes random access difficult. Compression is, however, common in database systems, and compression is frequently used when files are moved through telecommunication links.

1.1.2 Balance

Balance is the general solution for bottlenecks. The speed of the processor should match the I/O rate, and there should be an appropriate amount of memory. Amdahl stated a simple rule for balanced systems [Siewiorek *et al.*, 1982, page 46]. He calls for a megabyte of memory and a megabit per second of I/O per MIPS of processor speed. These numbers are not the last word. Hillis [Hillis, 1985], for instance, favors much less memory. Each application has its own requirements for balance, but whatever the chosen ratio, more processor power calls for more I/O bandwidth.

For uniprocessors, more processor power supplies more I/O bandwidth. High performance devices and striping can deliver practically unlimited data rates to the processor, and the file system on a uniprocessor naturally keeps pace with applications on the same hardware.

Parallel computers have no such automatic scaling. When a well-tuned file system reaches the performance limits of a single processor only improvements at the processor level will improve I/O performance. When a machine grows by adding processors or improving interprocessor communication, the file system remains fixed and balance suffers.

1.2 The Parallel Interleaved Approach

The file system bottleneck on parallel computers can be addressed elegantly with a file system designed as a parallel program. A computer with parallelism from the disk drive through the file system to each parallel application will suffer only the bottlenecks imposed by the parallel architecture, such as switch and memory contention.

Performance is not enough to justify a design for a file system. If performance were sufficient, the best design would be no file system at all. Ad hoc programmed access to parallel disks would have all the hardware advantages of parallel I/O with no file system overhead. But high performance is seldom so important that it overrides all other considerations. A practical file system must also be judged on its usability. Unfortunately, usability is a poorly-defined quantity. Our PIFS design supports two novel interfaces in addition to the standard file system interface. The unusual interfaces are not unequivocally usable; any claims we make for their usability must have a subjective element. The standard interface is, by construction, compatible with a standard file system. This compatibility is an objective argument for the file system's usability.

The standard interface is one of the services provided by the *parallel interleaved file server* (PIF server). The PIF Server also administers the parallel aspects of the file system. Together with the *tools* discussed below, it forms the top layer of the file system. The lower level of the PIFS consists of a collection of self-sufficient ***Local File Systems*** that store the pieces of parallel interleaved files.

A parallel file system must distribute data among its constituent processors and disk drives. We argue in section 2.3 that the best data distribution rule for general use interleaves the data so sequential access will cycle through the processors. This rule is simple, and it gives parallelism for sequential or random access at least as good as any other rule.

Operating system software hides irrelevant hardware details as much as possible. Our PIFS design follows this policy in the first of its novel interfaces, which we call *parallel-open.* The parallel-open interface groups a number of processes into a *job.* Each read or write from the job's lead process will transfer a record to or from each process in the job. This interface provides true parallel I/O up to the capacity of the PIFS. Beyond that the PIF server provides virtual parallelism.

Parallel-open is a convenient interface, but it entails IPC between the file system and the application that may not always be necessary. When enough file system processors are involved parallel-open files will be limited, not by the performance of the file system or the application, but by IPC bandwidth.

The possible interprocessor communication bottleneck is part of the motivation for the second novel PIFS interface which is based on the concept of *tools.* The tool interface lets I/O-intensive operations move into the file system. Tools interact with the PIF Server to learn the LFS placement of the files they will use; then they start processes that execute on the processors running the LFSs and communicate directly with the LFSs. Many tools will only send a small, fixed volume of data between processors.

Tools operate at the top level of the file system (rather than above it), but they use the PIF Server to manage parallel files. Tools never create, delete, or open the local component of a parallel file except by passing the operation through the server. This allows the server to protect the integrity of the PIFS.

1.2.1 Reliability

Any hardware component can fail, and a system with no redundancy and no superfluous components will fail if any component fails. The mean time to failure of a non-redundant parallel system decreases with its parallelism. Any file system that looses data has a serious problem, and a parallel file system with enough components will experience frequent component failures. The file system must be designed to tolerate failures.

Absolutely reliable disk drives would reduce the severity of a parallel file system's exposure to failure. The RAID devices discussed in section 1.3.4 are very nearly immune to failure. With such drives a PIFS that could detect processor failures could provide reliable storage almost effortlessly. Though RAID devices are likely to balance the powerful processor nodes of the future rather nicely, software-provided reliability may be more appropriate on existing machines.

I will consider three techniques for error recovery:

- Simple redundancy
- Multi-disk parity
- Application-level recovery

Application-level recovery is not strictly a file system issue. It is appropriate for robust data such as digitized audio and video.[6] Although a file system need have no direct role in this type of error recovery, it can contribute indirectly by permitting I/O to bypass file system error correction services.

Simple redundancy involves writing all data to at least two disks. This type of data protection is common for ordinary file systems. It works well and can give some small performance improvements for read operations, but it uses at least twice as much disk space as non-redundant storage. In many cases reliability is more important than the added storage cost. High-reliability computer systems routinely use redundant disks. They are now supported even for microcomputers [DPT, 1989].

Multiple-disk parity gives excellent reliability without much extra storage space. However, the straightforward implementation roughly doubles the time required for write operations. Multiple-disk parity is the technique used in RAID.

1.3 Related Work

Parallel techniques have been applied to I/O subsystems since the early days of computing. Channels and I/O processors are the rule on mainframe computers and have even been used on personal computers.[7] Database machines were tried and dropped, then tried again. Software techniques such as file striping have proven useful when high bandwidth is required for a single file. The earliest high-speed disk-like devices were "drum" storage. Drums used several heads

[6]A bad sector of audio or video data can be concealed by simply duplicating a nearby sector to replace the damaged data. This is commonly done for CD-audio, CD-interactive, and video-disk systems.

[7]Personal computers often use a SCSI controller. This controller implements some of the function that has traditionally been considered the role of the file system.

per track to give zero seek time and reduced rotational latency. Drums have given way to parallel transfer disks. It seems likely that parallel transfer disks will give way to disk arrays, a less expensive technology with potential for high I/O rates.

Other researchers have investigated parallel file systems. They have concentrated on comparatively limited parallelism (no more than sixteen processors), but their work is related to our PIFS research.

1.3.1 I/O Processors

Various forms of I/O processor are the earliest form of parallel I/O, and one that has been persistently successful since the 1950's. IBM realized in the early 1950s that low-level I/O was imposing a significant burden on the CPU. A "data synchronizer" for the IBM 705 allowed reading and writing to overlap computing. IBM also developed, but evidently did not sell, the IBM 703, a specialized tape sorting and collating machine [Bashe *et al.*, 1981].

Channels [IBM, 1988], I/O symbionts [cyberplus], and I/O processors by other names became the rule for large computers. These I/O subsystems ran in parallel with the CPU, and with considerable effort they could read several blocks of a file into memory concurrently. Their ordinary use was (and is), however, to improve multitasking performance [Freeman and Perry, 1977]. A large mainframe with 32 channels, 16 controllers per channel, and 16 heads of string[8] per controller can support 8192 seek operations concurrently and 32 concurrent data transfers at up to 4 megabytes per second per stream, for a total of 128 megabytes per second. This application of parallelism is intended to match slow disks to a high-speed processor complex.

1.3.2 Database Machines

The database field, especially the study of relational databases, has seen extensive work on parallel I/O. Database operations involve a significant amount of I/O, but also more processor activity than ordinary file access. Database operations often entail searching through files, assembling data from several records, decompressing and reformatting fields, and sorting the resulting collection of logical records. Parallel algorithms for relational database operations have been devised and implemented with good results [Bitton *et al.*, 1983; Valduriez and Gardarin, 1984].

[8]A head of string contains electronics for low-level disk operations. Drives are grouped in strings, with the head of string controlling its own disk mechanism and all the other disks in the string.

In 1983 Boral and DeWitt cast serious doubt on the whole concept of database machines [Boral and DeWitt, 1983]. They declared that fast database machines would be starved for data by slow disks. Options for faster data access such as parallel transfer disks were too expensive. The only architecture in which Boral and DeWitt saw any promise was one with a processor per disk.

Since 1983 parallel hardware has achieved some significant successes on database problems. Gamma [DeWitt *et al.*, 1986; DeWitt *et al.*, 1987] comes close to a processor per disk with 18 VAX 750's on an 80-megabit token ring with eight large disk drives. Gamma executes selection and update operations on the eight 750's with disks attached. Join, projection, and aggregate operations are executed in parallel on the ten processors without disks.

1.3.3 File Striping

Disk striping [Salem and Garcia-Molina, 1984] is a technique that applies interleaved memory ideas to disk storage. Consecutive sectors of a file are stored on consecutive disks in round-robin order. The only hardware required for disk striping is extra disk controllers and whatever hardware is required to hook them into the computer. Reliability suffers [Garcia-Molina and Salem, 1988], but striping usually spans a small number of disks so the effect on reliability is not disastrous.

Since striping only improves the I/O rate through the disk drives and controllers it is an alternative to disk arrays and parallel transfer disks, but it does nothing for the bottlenecks after the disk controller. When the data rate reaches the file system's limit further striping is ineffective.

1.3.4 Parallel I/O in Hardware

Parallel disks are a commercial reality. The CDC Hydra and the Fujitsu parallel-transfer disk [Gamerl, 1987] deliver data at about 10 megabytes per second by using five disk heads at once. Parallel transfer disks behave like ordinary disks in terms of seek and latency time; they just transfer data much faster (and cost much more). Storage arrays [Manuel and Barney, 1986; Masters, 1987] are comparatively inexpensive and give potentially unlimited data rates by combining large numbers of inexpensive drives with a single controller.

The RAID (Redundant Arrays of Inexpensive Disks) project at Berkeley [Patterson *et al.*, 1988; Katz *et al.*, 1988; Gibson *et al.*, 1989] proposes disk arrays with internal error correction and about a thousand processors. The RAID group has found no theoretical limits on the size of storage arrays. If they can work out the controller technology they should be able to build storage arrays with mean time to data loss measured in centuries and transfer rates measured in gigabytes per second.

RAID is complementary to our PIFS research. If progress in CPU and I/O technology were to continue as they have, the fastest ordinary disk drives will barely serve a single processor. Ordinary storage arrays and striping are not useful tools because they suffer intolerable failure rates when they are used with many drives. A RAID device can provide reliable storage and high-speed transfer. We depend on technology such as RAID to balance the performance of I/O devices with processors. If RAID devices are used in a PIFS, they can also simplify the attainment of reliability for the file system as a whole (see section 4.1).

1.3.5 Operations on Parallel Files

Parallel operations on databases have been a research topic for years, [Bitton *et al.*, 1983; Valduriez and Gardarin, 1984] but parallel algorithms for operations on sequential files are a comparatively recent development. In 1986 Sai Choi Kwan considered parallel external sorts [Kwan, 1986]. He created and tested parallel external sorts that achieved a factor of two speedup using four processors, but he failed to reach the performance of available commercial sorting packages—carefully tuned proprietary sort routines that make extensive use of parallelism inherent in the 370's channel architecture. Beck, Bitton, and Wilkinson [Beck *et al.*, 1988] built a carefully optimized implementation of an external sort on an AT&T research computer. They achieved excellent performance and speedup up to around four processors. The speedup was tailing off at eight processors, the upper limit of their tests.

1.3.6 Parallel File Systems

The Connection Machine [TMI, 1987] is a massively parallel SIMD machine. With up to 65,536 processors, it can outpace any conventional file system. Thinking Machines Inc. has implemented a SIMD file system. Their *Data Vault* uses 39 disk drives in what amounts to a storage array. The disks store 32-bit words with seven bits of ECC interleaved with one bit per disk. A data vault transfers data at 40 megabytes per second, and a fully-configured Connection Machine with eight data vaults can sustain 320 megabytes per second. The vault's file system can interleave a file across all eight vaults, so that data rate can applied to a single file. A disk in the vault cannot be considered in isolation unless the semantic unit of data is a bit, but in the context of the SIMD architecture of the Connection Machine, the data vault is an appropriate application of parallelism.

A parallel file system has been proposed for the Cal Tech hypercube [Flynn and Hadimioglu, 1988; Witkowske *et al.*, 1988]. The file system described in these papers is not interleaved, but rather divides the file into p groups of contiguous records and places each group under the control of a processor. The file access model is similar to the PIFS parallel open interface except that the Cal Tech file system supports three distinct open modes:

Single Each process that has a file open gets the same data at the same time. If the processes write, they must all write at the same time, but only the lead process will actually store data in the file.

Multi The file is accessible to one process at a time. When a process is done with the file it may specify which process receives control of it next.

Independent Processes that open a file get individual access to the file. There is no concept of a group or job. However, many processes can open a given file concurrently.

Intel, in cooperation with the University of Virginia, has designed and implemented a parallel file system that uses a loose form of interleaving [Pierce]. The Intel concurrent file system has not been tried with more than 8 I/O processors. Even early in the PIFS design we had at least 32 processors in mind. The difference between the Intel concurrent file system and our PIFS may be attributable to our different scaling requirements. The Intel concurrent file system does not allow applications to run on the I/O processors; this contrasts with the PIFS tool concept which encourages applications to move to the I/O processor. The Intel concurrent file system treats all the disks as one logical disk and allocates sectors from a free list that spans all the I/O processors. This leads to a more or less random distribution of data across processors. The PIFS placement rule is more restrictive and provably better for sequential access. Where our PIFS design uses a PIF Server and tools, the Intel concurrent file system uses a "name process" that maintains a directory of parallel files and a set of library functions that give applications direct access to the I/O processors.

Carla Ellis supervised the early development of the PIFS design and the first implementation. A technical report [Ellis and Dibble, 1987] covers this initial design work and some implementation experience. Carla Ellis continues to investigate parallel file systems at Duke University where she has studied prefetching in a PIFS [Ellis and Kotz, 1989]. The first widely-circulated publication of PIFS work was at the 1988 Distributed Computer Systems Conference, [Dibble *et al.*, 1988].

1.4 Overview

The remaining five chapters of this dissertation discuss the design and implementation of our PIFS, and the experiments used to test our design. Chapter 2 defines the role of a PIFS, discusses its design, and supports the claim that it satisfies reasonable usability requirements. Chapter 3 covers the implementation of our experimental PIFS and argues that experiments on it can be generalized to other implementations.

A file system operates in a complex administrative environment involving new disk drives, protected files, backups and other file system maintenance. Chapter 4 considers issues such as reliability, configurability, and security.

The underlying motivation for our PIFS design is high-performance I/O for parallel computers, and in chapter 5 we show that our design performs well and scales very well. Specifically, we discuss the design, implementation, and analysis of three file system tools.

Chapter 6 concludes the dissertation with a summary of contributions and suggestions for future work.

2 Structure

A parallel interleaved file system is designed to support a standard file system interface while simultaneously offering efficient access to the underlying parallelism. The internal structure of the file system software and the placement of data on processors supports these goals.

A PIFS offers three different types of parallelism through three different interfaces:

- Programs for which I/O is not a bottleneck, and programs with undemanding performance goals can use the ***standard interface***. The PIF Server may find and use implicit parallelism through the standard interface, but the interface includes no mechanism for expressing explicit parallelism. This is a consequence of compatibility with non-parallel file systems. On operating systems that permit alternate file systems, an ordinary program can switch between a PIFS and the default file system without modification.

- The *parallel-open* interface permits parallel programs to request truly parallel I/O to their constituent processes. The parallelism expressed at the parallel-open interface is an abstraction of the PIFS's underlying parallelism. The amount of true parallelism and the placement of PIFS data is not visible except as it is reflected in operation timings.

- *Tools* have full access to the file system's internal parallelism. They have potentially higher performance than any other I/O mechanism, but they must work through an interface designed more for performance than for convenience.

In this chapter we discuss the role of a file system and define a standard file system interface. Later we discuss the three PIFS interfaces and their relationship the the standard interface. We also discuss the internal structure of a PIFS, and explain the choice of interleaving as a record-placement rule. Finally we will characterize the PIFS design in terms of an effort to maximize transparency.

2.1 The Role of a File System

A PIFS is aimed at systems that need high I/O bandwidth to improve response time for long-running programs. These include scientific applications, business applications, and some real-time programs. We define a general-purpose file system as one that handles these challenging applications well, but does not inconvenience less demanding (for the file system) tasks such as program development and text editing.

In this section we will discuss the general-purpose file system interface in abstract terms. This general discussion is appropriate because the details of file system operations vary widely, and for most programs the details will be hidden under an I/O library. The elements common to most file systems used on general-purpose computers are:

- A file is a named array of data.
- The basic I/O operations are *read* and *write*. A standard file system also supplies *create* and *delete* operations that create and destroy files.
- *Read* and *write* are usually kept as simple as possible to improve their performance. Each *read* and *write* can include a file name and file position, but those data are seldom needed and moving them to *open, close* and *seek* operations optimizes the common case.

Though the *create, delete, read, write, seek* interface is the common minimum, it is frequently extended with keyed random access, database operations, and security. Keyed access and database operations are generally associated with *read* and *write*, while security is usually bound to file naming.

The standard file system interface makes few assumptions about the underlying hardware. The basic assumptions are:

- Stored data is permanent.
- By some measure, there is a large volume of storage.
- Random access is possible, but may be slower than sequential access.
- The storage is subject to failure, but failure is infrequent.

Random access is only required for certain types of files. Files on tape drives have slow random access with limitations on updates. If terminals, printers, networks, and so forth are treated as files, the file system will support access modes that permit no random access. With the exception of file system characteristics that depend on the above assumptions, the software attempts to conceal the details of the I/O hardware.

A file system interfaces with physical I/O devices. It will typically conceal every novel attribute of the physical device for at least three reasons:

1. If the file system provides any security for stored data, it must monitor all access to the storage devices. This is much easier if all access is through simple *read*, *write*, *create*, and *delete* operations.

2. By hiding the hardware, the file system makes programs independent of the actual I/O devices.

3. I/O hardware is often difficult to use. The control protocols are complex and must be followed carefully. The file system hides these details for the convenience of application programs, and because incorrect use of the hardware could have unfortunate results—up to and including reformatting a disk or incinerating a peripheral.

Given the clear evidence that the total storage requirement for computer systems is large and growing, one might expect evidence showing that files are large. Most studies show that files are small and short-lived [Floyd, 1986; Ousterhout *et al.*, 1985; Floyd, 1989; McKusick *et al.*, 1984; Satyanarayanan, 1981], but I assert that the studies reflect peculiarities of Unix. Unix files are used for inter-process communication and storage of small amounts of data that do not need a file system's permanence or capacity. The files maintained by the rwho daemon and the lock files used by the hack game could probably be better implemented outside the file system. A large number of Unix file operations are on such files [Floyd, 1989].

In the fall of 1984 I analyzed a month's SMF,[1] accounting, and database-tuning data from the University of Rochester's general-purpose IBM mainframe. The load included database work, administrative processing for the University and several local businesses, student program development, statistical analysis, text formatting, and extensive interactive work. The interactive use accounted for more than half the CPU time through the working day, and almost no resources at night. I measured total elapsed time in I/O as a function of file size, and discovered that the programs that were most I/O bound were accessing large sequential files. Time spent accessing small files was trivial. I argue without proof that if the load I measured was idiosyncratic, it was biased toward small files by the large number of students and other small-scale users found

[1]SMF is an IBM acronym for System Management Facility. It can be used to capture data about system activity in considerable detail. I used the file-by-file I/O counts and estimated I/O time in job-step accounting for my analysis.

at a University. I also assert that my measurements are more representative of general-purpose load than the measurements of Unix systems. My interest in high-performance computers and programs that are I/O-bound supports my choice of a mainframe load for measurement.

Comments in comp.arch on Usenet show that sequential I/O on very large files is important to the community that does scientific computing. Animated graphics is a particularly clear example. Assuming no compression, a 4096×4096 display of 24-bit color uses 48 megabytes per screen. Ten frames per second would not give smooth animation, but it would still require close to half a gigabyte per second. A fifteen second clip of such jerky animation would require a seven gigabyte file.

2.2 PIFS Internal Structure

A PIFS has two layers and an unlimited number of components. The top layer contains a parallel interleaved file server (PIF Server) and a set of tools. The bottom layer is an array of Local File Systems (LFS's). Figure 2.1 shows the interfaces between the components. Simple applications will interact only with the PIF Server, while more sophisticated applications may also invoke tools. For the highest performance, a sophisticated application may become a tool and gain access to the LFSs.

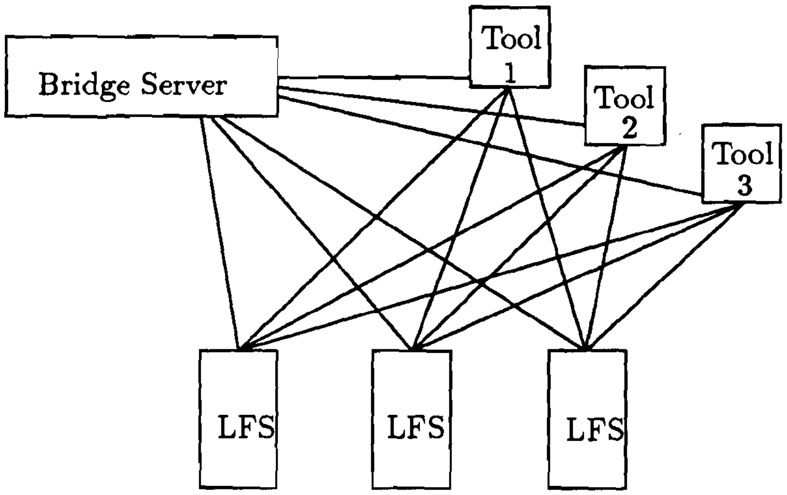
The standard interface and the parallel-open interface connect application programs with the PIF server, and only the PIF Server is visible to ordinary application-level programs. The PIF server communicates with application programs and with the other components of the file system: LFS's and tools. Tools communicate with LFS's and the PIF server. LFS's communicate with tools and the PIF server.

2.2.1 The PIF Server

The PIF Server defines and maintains the structure of parallel interleaved files, and repackages the parallelism provided by the LFS's into virtual parallelism that meets applications' requirements. All PIFS file operations originating from applications (not tools) are sent to the PIF Server. Tools can communicate directly with LFSs, but they must use the PIF Server to open, close, create, and delete PIFS files.

The PIF Server keeps handles that address each LFS. The server uses these handles to communicate with the LFSs, and it passes the handles to tools. The PIF Server also handles PIFS-level data for each open file. When a file is opened, the PIF Server locates the file in the PIFS directory. The PIFS directory entry

Figure 2.1: Components of a PIFS



includes identifiers for the LFSs participating in the file, and the LFS-level name of the file on each of those LFSs. The PIF Server uses that information to send an open request to each participant LFS, and retains the information for use in subsequent file operations.

For *create* and *delete* operations, the server will create or delete the PIFS file, issuing commands to the LFSs as necessary, and maintaining the PIFS directory to reflect the change. It also uses the PIFS directory entry with the an understanding of interleaving to direct reads, writes, and seeks to the appropriate LFSs. The algorithm for routing requests to LFSs is given below:

> The number of LFSs for the file is found in the file's directory entry.
>
> The LFS number for the particular record is calculated using the PIFS placement rule (see section 2.3).
>
> The file name for that component of the PIFS file is found in the directory entry.
>
> The LFS handle for the component is found in the PIF Server's list of LFS handles.
>
> The address in the operation is revised according to the PIFS placement rule.
>
> The operation is dispatched to the LFS.
>
> If a read or write spans several records, the PIF Server must dispatch several LFS operations to cover the I/O. It can dispatch them to run concurrently, but it must catch the returns from the LFSs and combine them into one response to the calling application.

2.2.2 The Local File System

Local file systems are complete, self-sufficient file systems. The PIFS relies on them to implement every feature of the file system that doesn't in some way involve parallelism.

At a minimum, a LFS must implement all the operations and concepts expected from a general-purpose file system: read, write, file creation, current position, end of file, and (optionally) file deletion.

If the PIFS is to support file security, it must be implemented at the LFS level. This is required because of tool-level access to LFSs (see section 4.4.2).

Support for logical records is not strictly required of the LFS, but it increases the utility of the PIFS greatly. Without support for logical records, the only semantically meaningful unit of data at the PIFS level is a byte. Since disks do not read or write single bytes, tools must use a larger grain, the physical record or sector. A tool can conveniently perform an operation on the first p 4 kilobyte (for instance) physical records of a file in parallel. If the data comes in units that do not pack neatly into 4k records, the tool must deal with single units of data that cross processors. With logical record support, a tool can easily process the first p meaningful units of data in parallel.

2.2.3 Tools and LFS

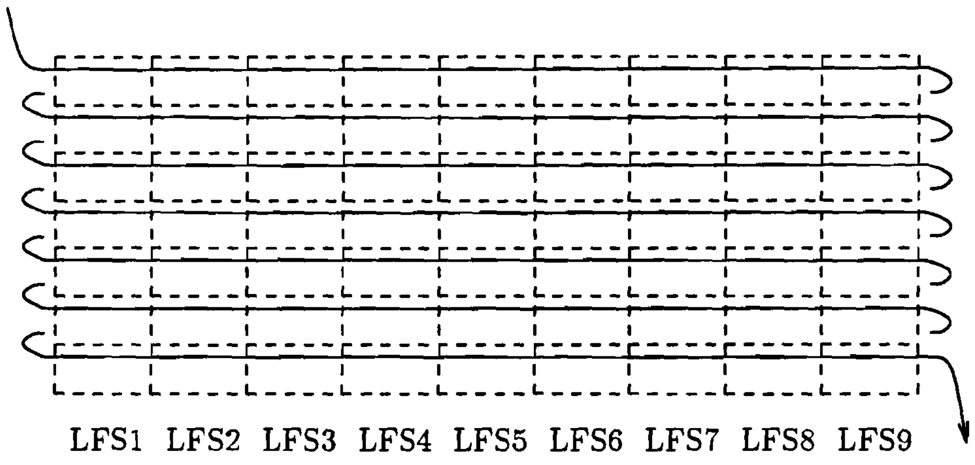
Tools communicate directly with LFSs. They may open, read, write, and seek. They may even create and delete files that aren't part of the PIFS. A tool uses the same PIF Server interface as ordinary applications; it must use this interface to open, close, create, and delete PIFS files, but it may also use any other PIF Server operation. For access to LFSs, tools use the same interface as the PIF Server. Access to an LFS by tools, or even ordinary programs, is not a security exposure because the LFS implements all the file system security.

2.3 Interleaving

A PIFS must distribute data among its LFSs. The placement rule that governs this distribution must distribute data evenly and predictably; it must be a reversible function, and it should be easy to calculate. We chose to use straightforward interleaving. The algorithm is simple and reversible, it is the best strategy for sequential access of all types, and it is as good as any other strategy for random access.

The equations for the location of a record in an interleaved file are the same as the equations that locate records in a two-dimensional array. For a file interleaved across processors numbered from 0 to $p - 1$, record n will be found

Figure 2.2: Sequence of Records in an Interleaved File



on processor $n \bmod p$. On that processor record n will be found in the local component of the file at record number $\lfloor n/p \rfloor$. The set of processors for a file is a subset of the processors participating in the file system. The description of a parallel interleaved file includes a list of the processors for the file.

The record placement rule has four important properties:

1. The rule is a reversible function. This lets a tool compute a record's position in a file from its local position.

2. Record placement can be calculated efficiently, usually with only two machine instructions.

3. For a p-processor file system, p consecutive records will always be on p different processors.

4. For random access, the rule is a good hash function. Any placement strategy that put the same number of records on every node with a small constant overhead would be satisfactory.

The rule that specifies the placement of data is not subject to change by any component of the system. It could be implemented as a library function or provided as an abstraction in the PIF server.

Other placement rules may require large maps or even I/O operations to locate a record. Hashing algorithms that work on the record number are in the same class as the PIFS interleaving rule (though these calculations usually cannot be reversed). For truly random placement or any other system that cannot be recalculated, the file system would require some form of index; for example, a in-memory map, an index stored on disk, or a linked list file structure.

Appends never cause movement in an interleaved file. Other structures can require reorganization of the entire file for each record appended. The placement rule used in [Witkowske *et al.*, 1988] partitions the file into p equal-size groups of consecutive records. Under that rule every pth append will require that the first record on each processor (except the first) become the last record on the previous processor.

The PIFS placement rule seems to conflict with allocation strategies such as that of the Gamma database [DeWitt *et al.*, 1986] which positions records according to key values such that each processor serves a known range of keys. However, a database at the tool level could position records in a file to get any required distribution. From a tool's point of view a file is a two-dimensional array of records. It can write a record into any position in the array. If the desired placement rule is $disk = H_1(k)$ and $record = H_2(k)$, the tool can calculate the record number in the PIFS file as $n = H_1(k)p + H_2(k)$, then use the standard PIFS placement rule to position the record at $(disk, record)$.

2.4 The PIFS Program Interfaces

2.4.1 Standard Interface

Like a conventional file system, a PIFS must implement the following facilities.

Read transfers one or more records from a specified file to program memory.

Write transfers one or more records from program memory to the specified file.

Delete removes a file name from the file system's name space and returns any resources allocated to the file.

Current Position is a handle that identifies the next record to be read or written. The current position is updated by the file system after every read or write. It can also be set through the program interface. This is usually done with a *seek* or *find* command.

End of File is asserted by the file system when the program attempts to read data that falls beyond the end of the file.

Parallelism in itself does not constitute a compelling reason for a file system to depart from the standard interface, but efficiency can, and usually does, motivate additions to the interface.

The minimum program interface places a heavy semantic load on *read* and *write*. Both commands must identify the file by its permanent name, check the program's access authority, set the current position in the file, and transfer data. The *write* command may also create or extend the file as required. Additional commands in the file system interface can simplify *read* and *write* and give the file system some useful hints. We therefore include them in our PIFS. The principle here is that heavily-used operations should be stripped for speed.

Create Adds a file name to the file system's name space. It may also allocate space, but this is not fixed by the design of the parallel file system.

Open gives the file system a chance to check file protection attributes. It also warns the file system to expect I/O on the named file.

Seek is included because non-sequential access is an exceptional case.

The *read*, *write*, and *seek* operations and current position are defined in terms of *records*, more specifically *logical records*. This is typical of general-purpose file systems. A logical record is a unit of data defined at the program level. The file system's application interface is defined in terms of these logical records:

- Offsets within a file are specified in terms of logical records.
- The size of a read or write request is expressed in terms of logical records.
- Interleaving is performed on the basis of logical records.

Logical record file structure is convenient for the program—it can address data in semantically-meaningful units. Logical records are also a powerful hint for the file system which can assume that I/O requests will involve logical records. For instance, a file system can expend effort to prevent logical records from spanning sector boundaries with a reasonable expectation that the effort will pay off when the record is read.

Unix-like file systems reduce the concept of the logical record to triviality. Such file systems support only one logical record size: a byte. This is a significant simplifying assumption for the file system implementor and for character-oriented applications, but it adds complexity and cost to the more sophisticated applications found in commercial and scientific environments.

A single processor program on a multi-processor system is not the main target for the PIFS design, but the file system can find and capitalize on parallelism in the standard interface. Parallelism can be found in ordinary read and write requests for more than one record. A read or write for p consecutive records can be dispatched to p LFSs concurrently giving an $O(p)$ speedup provided that IPC to the single originating processor does not form a bottleneck for the the operation. A PIFS's treatment of implicit parallelism is similar to the low-level parallelism of file striping [Salem and Garcia-Molina, 1986]. Disk striping reads sets of consecutive records in parallel at the drive and controller level, then serializes at the file system; a PIFS reads the same records in parallel, but carries the parallelism through the file system.

2.4.2 Parallel Open

The PIFS *parallel-open* interface lets the file system transfer data to a set of processes concurrently. This contrasts with the standard interface which can transfer data from all the LFSs in parallel, but only implements one destination per I/O operation. Concretely: given enough LFSs, an ordinary read of 5 records would execute the 5 reads in parallel, but the data would all go to the process that requested the read. A read on a parallel-open file could transfer those five records to five separate processes.

The parallel-open interface is so named because the interface is selected and defined at open time. The parallel-open request includes a list of I/O handles in addition to the usual file identification information. The nature of the I/O handles depends on the hardware and operating system software, but they identify paths through which the file system can communicate with processes. The handles might, for instance, be memory addresses or message queue IDs. The processes identified by the handles constitute a *job*, and the process that issued the parallel-open becomes the job's *controller*.

The parallel file system in parallel-open mode acts somewhat like a SIMD machine. When the controlling process asks to read a record, the file system delivers consecutive records to the handles designated when the file was opened. When the controlling process issues a write request, records are transferred from all the handles into the file system. The expected use of parallel open has one handle per process and one process per processor, but the location of the components of the job is not dictated by the parallel-open interface. All the handles could even reside on a single processor (though the memory on that processor might constrain the performance of the file system).

The file system implements no synchronization for the processes in a job. This is left to the job's controller. The rationale for this decision is that the job is capable of synchronizing itself and has more information about its synchronization requirements than the file system does. Consider an application for which data consumption is predictable at approximately 100 milliseconds per record. Depending on the contents of the records in the file and the nature of the handles, the controller for such an application could issue one read per 100 milliseconds without detailed synchronization. The only synchronization protocol would involve preemptive flow control from a process that was running far enough behind that it was exposed to a buffer overflow. Many tasks have approximately predictable record-processing time, and while they would run correctly with PIFS synchronization, they will run correctly and more quickly with internal synchronization.

The parallel-open interface hides the actual structure of the parallel file. The application does not even know the degree of parallelism provided by the file system. If an application requests more parallelism than the file system can offer, the file system sequentializes as required. Except for its influence on performance, an application using the parallel-open interface cannot determine the number of processors involved in the file system. Transfers through the parallel-open interface act on sets of consecutive records. For an n-way parallel open, if process x receives record r from one read, it will receive record $r + n$ from the next read.

The action of the PIF Server for a parallel-open file is closely related to its action for multi-record I/O. A single read or write request from the job's controller is translated to many requests by the PIF Server. The server dispatches enough LFS requests to satisfy the parallel read or write (as established in the parallel open). It waits for the LFS operations to complete and arranges for data to move between the processes in the job and the LFSs. When all the I/O for the request is done, the PIF Server responds to the initial request.

2.4.3 Tools

Parallel-open files use the parallelism in the file system but they cause unnecessary interprocessor communication. For example, a character-counting application would move the entire file between processors when it would actually suffice to exchange only two small messages with each processor controlling a disk if local counts were computed on those processors.

Interprocessor communication is often an important bottleneck for parallel systems. A mechanism for keeping I/O out of the interprocessor communication medium is therefore important.

The *tool* interface lets an application become a peer of the PIF Server. The application can move to the data such that a process of the application runs on each processor servicing the file. At this level, programs must know the internal structure of parallel files, but they can process the file with a minimum amount of interprocess communication. Tools are discussed in detail in chapter 5.

2.5 Transparency

In the interest of maximizing performance we kept the PIFS design as transparent as possible. Arguments for transparency such as [Parnas and Siewiorek, 1975; Snyder, 1986] and Lampson's summary paper on operating system design [Lampson, 1983] make it clear that the power of the underlying parallelism should be available outside the file system. Correctly implemented transparency gives the programmer opportunities for optimization that might be hidden in a less transparent system. However, Parnas emphasizes that a good design delivers usable power, not needless complexity, and Snyder points out that the interface should leave the programmer with an accurate picture of the relative costs of various operations. The arguments for transparency seem obvious almost to the point of triteness. They are not. Transparency runs counter to many other OS design goals: abstraction, security, protection, concealment of hardware details. The PIFS design strongly emphasizes transparency.

It would be unreasonable to allow completely transparent access to the *hardware* underlying a PIFS. The resulting interface would be restricted to particular hardware and would probably perform poorly. A file system with such transparency would only be completely defined if it included the specifications for the underlying hardware. This is generally bad style for operating system software. Protection of the file system's integrity under rigorous transparency would require an audit of each I/O request. The overhead of these audits on the low-level requests might cause them to perform more poorly than similar, seemingly less transparent, operations. If processes have access to I/O hardware, optimization of I/O for the system as a whole is severely hampered. Techniques such as track buffering and head scheduling are only useful when I/O devices are controlled by a single entity (such as the file system).

Transparency at the *device driver* level would make the interface largely hardware independent, but it would not solve the protection problem. The file system would have to check the disk address for each read or write operation to ensure that it was to a sector accessible to the calling program.

Restricted transparency at the *LFS level* is granted by the PIFS tool interface. The difficulties cited above are considerably weakened at this level. Tools are hardware dependent only to the extent that they must adjust to the number of processors in the file system. Since the LFS is a competent file system, its operations are secure within the LFS context. Tools have unlimited access to LFS operations, but PIFS files are not maintained by the LFSs. To protect the integrity of the parallel file system, tools are required to create and delete PIFS files through the PIF Server. A correct tool will not create incorrectly formed PIFS files, and an incorrect tool can only damage the PIFS files it is permitted to write. Files damaged by incorrect tools will not behave correctly when they are accessed though the more abstract interfaces, but they remain accessible to tools and are perfectly accessible to PIFS-level deletion.

The standard interface duplicates the LFS interface except that every operation that might reveal the underlying parallelism is removed. For instance, the PIFS standard interface does not support any command that would return an absolute disk address or disk IDs. These data could be deceptive or expose the underlying parallelism; for instance, a PIFS file has p different disk IDs depending on the current record.

Some applications may not want to use the non-parallel standard interface, but may still be unwilling to tackle the complexity of a tool. The parallel-open interface is a compromise designed for such programs. The virtual parallelism of the parallel-open interface gives these programs explicitly parallel I/O without exposing them to the LFS-level structure of the PIFS.

2.6 Summary

Each of the three PIFS interfaces contributes to the system design goals. The standard interface addresses the ease of use goal, but its performance is not much better than simple striping would provide. The parallel open interface extends the standard interface to include explicit virtual parallelism. It gives programs access to the full parallelism of the file system without giving them access to the underlying file structure. The tool interface supports high performance by permitting explicitly parallel optimized file system operations, and by promoting local communication with the LFSs (instead of forcing tools to communicate with the file system across processor boundaries).

No single placement rule is better than the interleaving record placement rule for sequential access or for random access. Ad hoc placement rules can outperform interleaving for some database operations, but a database tool can make provisions that superimpose the optimized rule on the standard placement rule. The placement rule always allows the maximum possible parallelism for standard and parallel-open access.

3 Implementation

Performance and scalability are among our fundamental goals for parallel interleaved file systems, and we claim that we have achieved these goals. This claim is supported by abstract analysis and by measurement. This chapter describes the implementation used for the measurements, and supports the claim that the measurements are meaningful. The prototype PIFS, Bridge, was implemented on a 120-processor BBN Butterfly [BBN, 1986] under the Chrysalis [BBN, 1987] operating system. It meets the stated performance goals for the standard interface and tool interface.

Bridge is an experimental file system. It clearly validates the basic PIFS design, but its absolute performance does not always compete with commercial file systems. We expended some effort on tuning, but we rely on measurements and extrapolation to project our measurements onto highly-tuned file systems. For sequential I/O, Bridge compares well even with file systems on more capable processors (see table 3.1). To the extent that our experiments are dominated by these operations, our measurements need no adjustment.

Aspects of our Bridge implementation were dictated by hardware constraints:

- Disk drives are simulated in main memory by code that imitates the performance of CDC Wren IV drives [CDC 88, 1988].
- Our experiments used at least 60 megabytes of RAM disk space; this consumed all the available memory on about 70 processors. The file system software and experiments were constrained to the remaining processors. This shortage of processors limited Bridge to 32-way parallelism. This is an unfortunate restriction since our analysis shows that over 100 processors would be useful.

Implementation of Bridge started in 1985 [Ellis and Dibble, 1987] under the supervision of Carla Ellis. The first version was functional, but remarkably slow—a write took over 150 milliseconds. Poor performance notwithstanding, the early version of Bridge demonstrated that the parallel interleaved file concept had some validity.

Table 3.1: Performance of Comparable File Systems

System	Read	Write
Xenix (33 mhz 80386, 18 ms drive)	55 k/sec	55 k/sec
Bridge (8 mhz 68000, 16 ms drive)	100 k/sec	30 k/sec
OS/2 (33 mhz 80386, 17 ms drive)	100 k/sec	55 k/sec
Mac II (with 19 ms drive)	170 k/sec	50 k/sec
Unix (Sun 3/80 with 16.5 ms drive)	1700 k/sec	730 k/sec

The non-Bridge figures in this table were taken from [Rinko-Gay and Varhol, 1989; Mallett and Smith, 1989].

Work on Bridge continued under the supervision of Michael Scott. Considerable effort, mostly improvements at the EFS level, yielded a factor of five improvement in performance. The performance of read and write improves slightly as processors are added. Open and close are independent of the number of processors. File creation becomes slightly slower as processors are added. File deletion improves linearly with the number of processors, but this is an artifact of the $O(n)$ file deletion algorithm used by the LFS.

3.1 Bridge on the Butterfly

We implemented [illegible]e on a 120-processor BBN Butterfly [BBN, 1986] running the Chrysalis o[illegible]ng system [BBN, 1987]. Each node in a Butterfly has a megabyte of m[illegible]ory, an 8 mHz 68000, and a microcoded processor that implements fast interprocessor communications.

3.1.1 Elementary File System

The LFS used in the Bridge implementation is the Elementary File System (EFS) [Thomas and Toner, 1984; Schantz, 1984] which BBN designed for the Cronus distributed operating system [Gurwitz *et al.*, 1986]. EFS features stable storage and stateless access, but it is slow and it is about as far from the standard file system interface as any "real" file system. It is an unusual choice as the LFS for a high-performance file system, but its simplicity made it easy to port to the Butterfly, and the sources were available.

The Elementary File System has a number of peculiarities:

- It is stateless. It has neither an open nor a close operation.
- The blocks in a file are stored in a doubly-linked circular list. The pointers are disk addresses supplemented with a disk ID.

- Random access is accomplished either through an index linked to the first block in a file, or by reading sequentially through the file from a known point (beginning, end, or current location) to the desired block.
- There is no support for logical records.
- The name space is flat. Moreover, file names are integers assigned by the file system.
- Unusual operations such as "copy file" and "make permanent" are provided.

The extra file management operations supported by EFS can be, and are, ignored by the Bridge Server.

The statelessness of EFS contributes to its uneven performance. Sequential read/write operations return the disk address of the probable next block as a hint for the next operation. With the hint, a sequential read will hit the next block on its first read. Without the hint, the read may traverse the links in the file from the beginning or the end until it reaches the desired block.

We did not want to require application programs to maintain hints for each open file, so we called on the Bridge Server to provide hints. By providing hints for the EFSs, the Bridge Server makes PIFS file operations not stateless even though they are based on stateless EFS file system operations. Since tool I/O does not pass through the Bridge Server, tools must supply hints to EFS. If the tool fails to preserve each file's state, in the form of a hint for the next operation, performance degrades markedly.

The Bridge server requires an *open* command to set up a context for subsequent file operations on a file. The file's context is associated with the file name and updated by the Bridge server after each read or write operation.

Blocks of data delivered to application programs are images of disk sectors. The disk driver level of EFS uses 1024-byte sectors. Each sector contains 24 bytes of EFS overhead (see figure 3.1), sixteen bytes of Bridge overhead and 984 bytes of user data. The pointers in the 24-byte EFS header lead to blocks that are interpreted as adjacent within the local context. In other words, the block pointed to by the *next* pointer is p blocks away in the Bridge file. Bridge claims 16 bytes of each block, and previous versions of Bridge have used that area for links analogous to the EFS links. The current version of Bridge only uses its reserved space in each block for internal consistency checks such as checking to make certain that each block read is the requested block in the correct file.

EFS has no concept of a logical record, only the notion of a 984-byte block. When we could, we designed our experiments around a 984-byte record length. When we required another record length, we implemented logical record support in the calling program.

Figure 3.1: Elementary File System Block Header

Bytes	Description
4	FileID number for this file
4	Block number in file for this block
4	Disk address of next block
4	Disk address of previous block
4	Disk address of last block
1	File type (short/long)
3	reserved

Table 3.2: The Full EFS Command Set

Create file	Initialize a new file and returns its name
Delete file	Free the storage and the name of a file
Sequential read	Read a record using links
Sequential write	Write a record using links
Random read	Read a record using the index block
Random write	Write a record using the index block
Make permanent	Mark a file "permanent"
Copy file	Create a copy of a file

Considered in context, the unusual EFS file structure is not poor design. EFS was constructed as part of a reliable distributed system. Each disk block includes enough information to allow a disk with a damaged directory to be reconstructed from a scan of the sectors on the disk.

EFS has been modified to fit into Bridge, but most of the modifications have involved adding instrumentation, improving performance, or adapting to the Butterfly and Chrysalis. The original version of EFS expected service requests in the form of function calls. The Bridge version accepts requests passed through the Chrysalis "dual queue" message system.

The revised version of EFS can still be used by a program that knows nothing about Bridge. In particular, the portion of a Bridge file controlled by one EFS server can be viewed locally as a complete file. The EFS server can ignore the fact that it holds every *p*th block of a more global abstraction. Even within the context of a PIFS the instances of EFS are self-sufficient and operate in ignorance of one another.

The EFS Primitive Operations

EFS as distributed for Cronus supports the operations listed in table 3.2. The command set of the version used with Bridge is somewhat different. Files are made permanent when they are created. We never use ordinary EFS random reads and writes because they have extremely unstable performance; instead we use sequential access and let EFS hunt through the linked file for the target block. The built-in copy operation performs much better than a high-level stream of reads and writes, but the operation is not supported by common file systems, and its use would have cast doubt on the validity of a copy tool. The copy operation still exists, but it is not used nor is it supported at the Bridge Server level.

Files are linked lists of blocks, but EFS does not use a free list structure to organize free space on a disk. EFS maintains free space as a bit map. This structure is justified by disk recovery considerations, but the combination of linked list files with bit map allocation is not good for performance. File deletion involves stepping down the linked list of a file marking each record free in the allocation map. The original code for file deletion took pains to keep the disk in a recoverable state. Removing those measures improved performance, but deletion is still $O(n)$ (where n is the size of the EFS file).

Sequential Read/Write Optimization

In EFS terms the I/O operations used by Bridge are all "sequential" even if they access the file randomly. If access is actually sequential, the Bridge Server will provide correct hints, and I/O operations will run in roughly constant time. When EFS receives an incorrect hint or a request for a non consecutive record, it must search for the requested record.

EFS file blocks are doubly linked. As originally implemented, EFS finds a requested record by checking at the hint then searching from the beginning of the file if the hint is incorrect. The Bridge implementation of EFS improves on the search algorithm by a large constant factor. Sequential read and write operations locate their target record by following links forward or backwards from the closest known point in the file: beginning, end, or hint location. If the block addressed by the hint contained the requested record for a read or either the requested record or the preceding record for a write, EFS will skip the search.

EFS Caches

EFS itself maintains one cache, an LRU write-through cache of disk blocks. This cache almost always contains the base sectors (the usual initial block in a circular doubly-linked list) of all the open files, and the blocks that contain the file system's global data such as the allocation bit map. The EFS cache improves EFS's performance by about a factor of ten over EFS performance without the cache.

The disk simulation includes a track buffer. The simulator only charges a full disk access delay on the first access to a track. Until the head moves to another track, accesses are only charged for SCSI transfer time. Without the EFS cache, the track buffer would be useless. EFS would move the head to the root of the file and to the global data structures for every read; since these are on two or three different tracks, every low-level read would cost a seek. In conjunction with careful file allocation, the track buffer reduces disk I/O time to an average of about 3 milliseconds per read. Figure 3.2 shows that the majority of read operations take no more than five milliseconds. This suggests that most reads involve no more than one low-level read. The longer read times result from overhead operations and EFS cache misses.

3.1.2 Bridge Server

In our implementation of Bridge the Bridge Server is a single centralized process, though this need not be the case. If requests to the server were frequent enough to cause a bottleneck, the same functionality could be provided by a distributed collection of processes. Our work so far has focused mainly upon the tool-based use of Bridge, in which case access to the central server occurs only when parallel interleaved files are created, opened, or deleted.

The Bridge Server has four functions:

1. It maintains the Bridge directory.
2. It implements the standard and parallel-open interfaces.
3. It makes EFS handles available to tools.
4. Like the rest of the Bridge system it is extensively instrumented for consistency checking and performance monitoring.

Server Operation

In support of the functions listed above, the Bridge Server monitors and optimizes the operations that pass through it. The Bridge Server is driven by messages from applications and LFSs. Its main control structure handles these messages. The following pseudocode contains a few representative fragments of this control structure:

```
pos is the current record
Receive Msg
switch(Msg.Op)
```

Table 3.3: Bridge Server Commands

Command	Arguments	Returns
Create File		File id
Delete File	File id	
Open	File id	LFS file ids
Sequential Read	File id	Data
Random Read	File id, Block number	Data
Sequential Write	File id, Data	
Random Write	File id, Block number, Data	
Parallel Open	File id, Worker list	
Get Info		LFS handles

```
case Read
    if the file is parallel open
        for i = 0 to parallel-open parallelism
            calculate EFS handle and record number for the record $pos + i$
            get a hint for the record from the hint cache
            send read message as calculated
        while a read is outstanding
            wait for a return from an EFS
            $n$ = the job member corresponding to the read
            route the message to the process[$n$]
            update the hint cache
        update $pos$
    else not open for parallel access
        calculate EFS handle and record number for the record $pos$
        hold the record number in the message
        insert the calculated record number in the message
        set the EFS hint in the message from the hint cache
        forward the read message to the calculated EFS
        wait for the response from the EFS
        update the hint cache
        replace the held record number in the message
        forward the message to the calling application
        update $pos$
case Write
    Write is similar to read
case ParallelOpen or Open
    allocate an FD
    update file name in FD
    read the Bridge directory entry for this file
```

```
        copy the EFS number list and file name list
              from the directory entry into the FD
        if this is a parallel open
            copy the parallel-open handle list and handle list size into the FD
        if the caller provided an information buffer
            Copy the EFS number list and file name list into
                  the information buffer
    case GetInfo *Called by tools*
        build a return message containing EFS Handles and EFS Processors
        send the message to the caller
    case Create
        for each EFS
            Send a "create file" message
        allocate an FD
        for each EFS
            wait for completion
            store the file name in the FD
        create a Bridge directory entry
        free the FD
        send a "completed" message to the caller
```

A file descriptor stores information related to an open file. Since file descriptors are identified by the associated file name (not by path number or other abstract handle), a file descriptor amounts to a somewhat augmented RAM copy of the Bridge directory entry. The file descriptor for a parallel-open file includes:

- A list of the EFS numbers for EFSs participating in this file;
- A list of the EFS names for the local components of the file;
- A hint for each EFS;
- The current position in the file.
- A list of handles for members of the parallel-open job. The first entry in the list is the job's controller.

Since all the lists are stored in static arrays, each list is accompanied by a field giving the list's length FDs for files opened with an ordinary *open* operation have no value in the list of handles for job members.

Our implementation of the hint cache contains a hint for each active file/EFS pair. An inquiry against the hint cache always returns a value. If there are no cache entries for the EFS/File combination the hint cache will return a special "unknown" code; otherwise the cache will return the one value it contains for that EFS file. Each hint specifies the location of the most recently accessed record of the file, on that EFS.

The Bridge Directory

The Bridge Server maintains the Bridge Directory, which is a standard Bridge file except that it is only accessible through the Bridge file-maintenance operations: create, open, and delete. The Bridge directory contains the Bridge names of files and a list of the LFS files that make up each Bridge file. An LFS file is identified by the processor ID of the processor that runs the LFS and the LFS's internal file name.

All the Bridge files with which we experimented spanned all the LFSs and used the same name for all their LFS components. We thus did not fully use the power of the Bridge directory. Much of the information in the Bridge directory is intended for use in an administered system. The directory supports different sets of processors for each Bridge file, and LFS's arranged in different orders. It also permits the LFSs to use different names for the parts of a file. The flexible naming is especially useful when the LFSs are used as autonomous file systems as well as components in a PIFS. In that case the Bridge Server does not have complete control over the LFSs and it may not be able to enforce uniform naming. Also the LFSs may not have identical disk subsystems. If one LFS uses a large drive and another LFS uses two small drives, the file names may reflect the drive selection. These file system administration issues will be covered in some detail in chapter 4.

3.1.3 Interprocessor Communication

The fastest communication medium on the Butterfly is shared memory. Bridge, and the Bridge tools, avoid that facility as the primary communication path in almost every case. If the system's performance depended on the shared memory, it could be argued that Bridge only demonstrated that a PIFS is practical on a shared memory multiprocessor. We did not want to put such limitations on our argument. Dual queue operations coupled with block memory transfers are used to implement message passing for Bridge and the tools. Dual queue operations and block copy are both supported in microcode on the Butterfly, and their performance is comparable to that of message passing systems on other tightly-coupled multiprocessors. Performance might change if our IPC mechanism were replaced with operations that passed through a slower network, but our choice of algorithms would remain the same.

Table 3.4: Low-level Chrysalis Functions

Operation	BFly Microseconds
Remote block copy	50 + 380 per K
Local block copy	50 + 600 per K
Make object	1000
Make remote object	6100
Map object	1350
Enq	80
Deq	80
Start process (simple)	10000

Dual Queue Function

A dual queue is a LIFO data structure with two states. In one state the dual queue contains a list of 32-bit enqueued data waiting for dequeues. In the other state the dual queue contains the process IDs of processes waiting for data to dequeue. A dual queue serves as both a synchronization mechanism and a tool for passing handles for blocks of memory, processes, and other dual queues.

Bridge moves blocks of data with the following sequence of operations:

```
Receiver                  Sender
dequeue(QID)
                          OID = Create object
                          x = map object(OID)
                          block move data to x
                          enqueue OID on QID
dequeue returns OID
y = map object OID
block move from y
```

This protocol shows the receiver's dequeue before any sender operations, but it could actually take place at any time. If the sender has already enqueued data the dequeue will simply return that data immediately. Typically the object would be created and mapped at the sender's end before the communication event. It might even be mapped in advance at the receiver's end.

The communication protocol would involve an enqueue and a block move at the sender's end and a dequeue, a map, and a block move at the receiver's end. Referring to table 3.4 and assuming that the sender had the communication object created and mapped in advance, we see that the send time for a one-kilobyte message is 730 microseconds. The receive time is 1860 microseconds. The total elapsed time is a little over 2.5 ms, which is not unusually fast. Around one millisecond is an excellent IPC time.

3.1.4 Simulation of Disk Drives

It would not have been practical to install a large number of physical disk drives on the Butterfly. We chose to use the memory on about 70 of the Butterfly's nodes as a storage medium, and implemented software disk simulators that intercept the device driver's accesses to the disk and convert the device control protocols into appropriate delays and data transfers.

The disk delays are simulated by two sleep times, a read sleep and a write sleep. The simulator sleeps the appropriate interval whenever the drive is called on to access data that is not in its track buffer. The track buffer protocol and sleep intervals were chosen to approximate the performance of a CDC Wren IV disk drive [CDC 88, 1988] (a standard SCSI hard disk drive with 18.5 ms average seek time and a 32k data buffer).

We impose the average seek plus latency time on every disk access. For a sophisticated file system this would be unreasonable, because many file systems attempt to group disk blocks according to expected access patterns. EFS takes no special care with its allocation patterns or seek optimization. After it had been running long enough to fragment the disk's free space it would probably fit our simulation quite well.

For sequential reading on a single large file, the EFS/disk simulation combination gives a ten millisecond read and a 31 millisecond write. In the area of EFS's strength it does well enough to compete with file systems on much faster processors and comparable disk drives (see table 3.1). The measurements in chapter 5 show that EFS performance loses about a factor of two under a less predictable load, but the performance EFS delivers to tools is reasonable for an 8 mhz 68000.

3.2 Performance of Primitive Operations

3.2.1 The Measurement Environment

The figures in table 3.5 are taken from a simple program that uses the standard interface to the Bridge server in order to read and write files sequentially. The performance of *open* and *write* operations is essentially independent of p (the number of file system nodes). Read operations pay an amortized price for startup tasks that would be borne by the *open* operation in a more traditional LFS. The startup overhead includes initial reads of file header and directory information. Average read time for typical files is more than a third less than disk latency because of full-track buffering in our version of EFS.[1]

[1]Write operations actually pay an amortized startup price as well, but its effect on average time is almost negligible, partly because writes take so much longer than reads, and partly

Table 3.5: Performance of Primitive Operations

Operation	Time in ms
Delete	$20 \cdot \mathit{filesize}/p$
Create	$145 + 17.5p$
Open	80
Read	$9.0 + 500p/\mathit{filesize}$
Write	31

The quoted performance figures are accurate average values when the file system is accessing a large file sequentially. Any individual read or write is likely to deviate widely from the average. We took histogram measurements of read and write times (see figure 3.2) and found that the file system would read or write much faster than the average time while it could satisfy the requests from one of its caches. When the requested data was not in a cache, the system would perform several disk accesses and change the equilibrium in its cache. That access and several subsequent accesses would be well below average performance. This non-uniform access time can be compensated for by application programs with large read/write buffers and asynchronous I/O into the buffers.

Parallelism has only a very weak influence on the performance of I/O operations on single records, but requests for more than one record can be parallelized by the Bridge Server. This implicit parallelism gives $O(p)$ speedup for large blocks of data until the aggregate transfer rate saturates the calling processor.

The Create operation must create an LFS file on each disk. Bridge gains some parallelism for this operation by starting all the LFS operations before waiting for them, but the initiation and termination are sequential, leading to an almost linear increase in overhead for additional processors. Performance could be improved somewhat by sending startup and completion messages through an embedded binary tree.

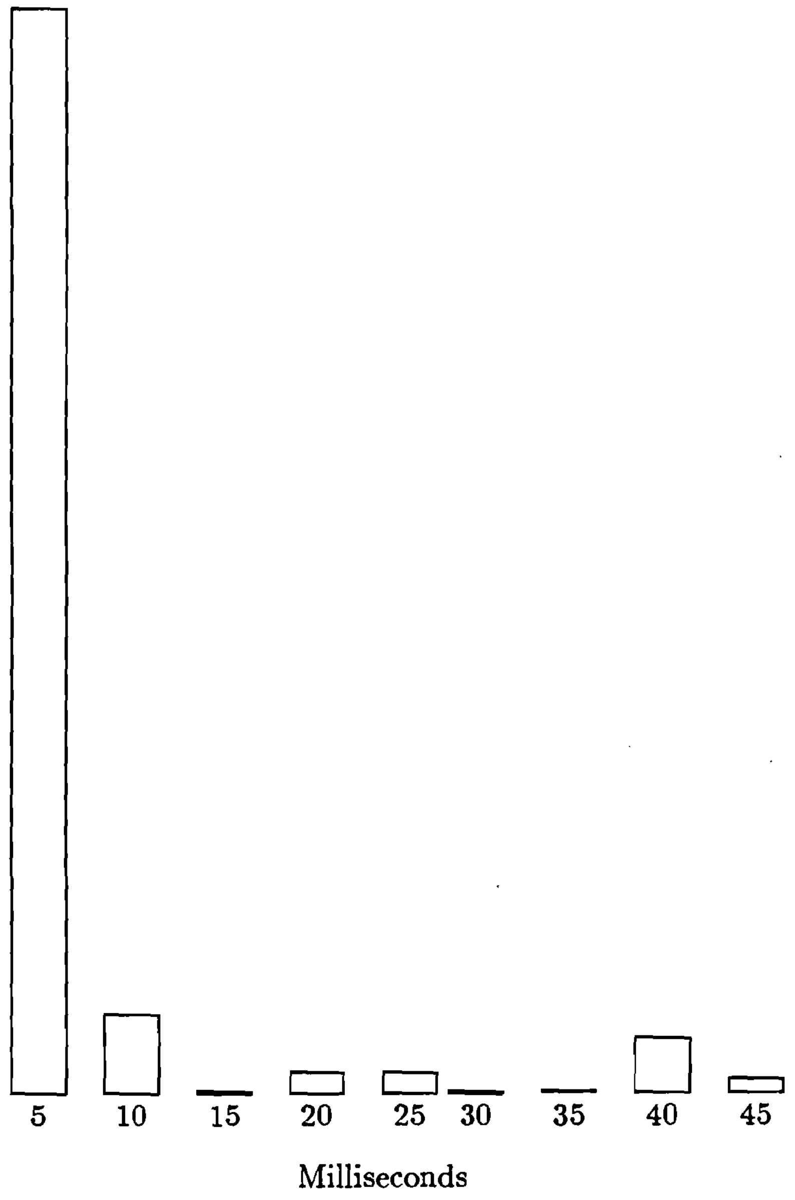
The Delete operation runs in parallel on all instances of the LFS, but because of the design of EFS it still takes time $O(n/p + p)$ where n is the size of the PIFS file being deleted. The additive p reflects startup time. The Bridge Server could be implemented so startup time was reduced to $\log p$, but the constant factor for this term is so small that the improvement would be inconsequential.

Pre-Fetch

We tried to address the problem of irregular read and write times by implementing prefetching in the Bridge Server. Each time it received a read or write

because of EFS peculiarities that make caching of directory information less effective for writes than it is for reads.

Figure 3.2: Distribution of Read Times



request the Bridge Server would transmit the request to the appropriate LFS, then send a prefetch request for the next record to the next LFS. Our theory was that the prefetches would ensure that every read and write was a cache hit. This would result in improved and more uniform performance. The results were slower access with no significant improvement in uniformity.

The expected operation sequence for the prefetch experiment was:

Server	**LFS 1**	**LFS 2**
Send read to LFS1		
Send prefetch to LFS2	reading	
Wait for read	reading	prefetching
Read returns		prefetching

Close inspection of the prefetching results showed that the actual sequence of events brought about by prefetching was:

Server	**LFS 1**	**LFS 2**
Send read to LFS 1	still prefetching	
Send prefetch to LFS 2	still prefetching	
	still prefetching	prefetching
wait for read	reading	prefetching
Read returns		prefetching

This sequence gave a nice performance improvement the first time around the LFS cycle, but the second time around the access time was a little more than a message time worse than the results without prefetching. A switch that turned prefetching on and off depending on recent history would have improved access time for the first p reads, but would not have made a noticeable difference for programs that did thousands of reads and writes.

3.3 Implementation Experiences

The design of EFS makes major performance sacrifices to provide robust and recoverable disk storage. These features contributed nothing to our Bridge experiments. The widely variable performance of the file system made analysis difficult, and the unusually expensive delete operation had an unfortunate effect on the absolute performance of our tools. The high-performance copy operation supported by EFS might have proven an advantage for a copy tool, but we chose not to use it because it is non-standard.

We used a subset of the spartan EFS command repertoire. It would have been far easier to work with a feature-rich file system, but the restricted EFS environment forced us to create portable tools. The algorithms we used should work efficiently on any system that performs well at sequential reading and writing. The difference between EFS's sequential I/O and its slower file operations is dramatic; according to IOBench figures [Rinko-Gay and Varhol, 1989], EFS's penalty for random access is much more pronounced than any comparable file system.[2] This may have skewed our design choices toward sequential I/O, but strong evidence [Floyd, 1986; Ousterhout *et al.*, 1985; Floyd, 1989; McKusick *et al.*, 1984; Satyanarayanan, 1981] suggests that applications almost invariably use sequential access in any case.

We considered the possibility that EFS's file deletion performance may have influenced our tool designs, but our consistent use of standard algorithms for tools alleviates this concern. EFS did not exist when mergesort became the standard file sorting algorithm. Since our tools do use ***delete***, we have to show that our good results do not depend on slow deletion. This is a reasonable concern. File deletion at the tool level is entirely parallel and file deletion time is more than twice the time it takes to read the entire file. Our most important measure for tools is scalability, and file deletion represents an $O(n/p)$ interval of linearly scalable activity. When we present the results of our tool experiments in chapter 5 we will use our analysis to predict the tool's performance on a canonical high-performance file system. This technique would detect tools that depend on EFS's performance peculiarities.

In summary, the principal redeeming feature of EFS was its availability. It has difficult performance characteristics that we fixed to some extent but ultimately had to tolerate. EFS also has some nice features that we had to ignore to support our claim that a PIFS doesn't call for special features in the LFS. The Bridge Server is a comparatively simple program. It does not use enough time to make optimizations useful and its functions are quite simple.

3.4 Summary

Bridge is a straightforward implementation of a PIFS. It uses a "real" file system as an LFS, and its performance on the most important operations is in many cases somewhat better than other file systems with similar processors and disk drives. Bridge is particularly good at sequential I/O. Largely because of its high-performance disk drives, EFS sequential access on an 8 mhz 68000 compares with the Xenix file system on a fast 80386. Bridge is not good at at random access, but the effect of this shortcoming is small.

We believe that the behavior of Bridge is typical of a PIFS on the class of hardware we use. Bridge deviates from this standard only for random access and file deletion. We seldom use random access, so it has little effect on our results. We do use file deletion, so we will consider the effect of deletion performance

[2]Here we classify EFS with file systems from Unix, MS-Dos, OS/2, OS-9, VMS, and MVS.

when we evaluate the performance of tools. Measurements taken on Bridge accurately characterize PIFS performance for most programs. Measurements combined with algorithm analysis can project beyond our implementation to PIFS in other hardware and software environments.

4 Meta-Issues

A usable file system requires more than a convenient program interface and high performance. A production file system should run without data loss for decades. When hardware fails, recovery should be quick and painless. Some people care about security; the file system should provide for them. System administrators like to add disk drives and adjust load balance to fit their concept of optimization. A good file system must certainly accommodate system administrators.

Our PIFS implementation is on a Butterfly, but it was carefully written as a generic parallel program. No aspect of the design or the implementation is specific to the Butterfly or a shared memory architecture. Our PIFS design should run well on any reasonable hardware.

The PIFS design accommodates all the above concerns. Some features, like flexible hardware configuration, fall gracefully out of the design. Other features, like security and resistance to failures, are straightforward enhancements. In this chapter we will discuss each requirement and show how a PIFS can meet them.

4.1 Resiliency

The large number of hardware components in a non-redundant parallel computer makes the machine very sensitive to unreliable hardware. The obvious solution is reliable hardware. If sufficiently reliable hardware is not available, redundant design gives excellent reliability. A PIFS adapts well to redundant hardware. In this section we will show that a PIFS can get a better error rate than a conventional file system with ten percent disk overhead, no performance penalty for reads, a small constant factor performance penalty for writes that have no explicit parallelism, and only a factor of two slowdown for writes of more than $p/2$ records in parallel.

The reliability of a system of independent components can be calculated from the failure probabilities of each component. For a PIFS with one disk drive on each processor, the failure of a single disk or processor will result in a partial failure of every file represented on that disk. For most purposes, partially destroyed files are useless. Therefore a failure that causes data loss anywhere in a PIFS should be considered a total failure. The components of a PIFS are roughly independent, so the probability of flawless operation for an entire PIFS is the product of those probabilities for all the components.

An excellent-quality disk drive can be expected to run without failure for an average of 60,000 hours [CDC 88, 1988]. The failure rate is not a simple function of time, but given a file system with many disk drives of different ages, a uniform failure rate giving a probability of failure of 1.28 percent per disk per month is a good estimate. That is a 98.7 percent chance of no failure. With 100 disks, the probability of flawless operation drops to 28 percent which gives a 72 percent probability of a failure in a month.

More reliable hardware would increase the reliability of the file system without altering the file system. A hardware solution is simple and efficient because it is invisible to the file system, and the extremely high reliability of RAID storage demonstrates the practicality of a hardware solution. The probability of failure for RAID storage can be made 0.012 percent per RAID per month or better [Gibson *et al.*, 1989]. An entire PIFS system composed of 100 RAIDs has a 1.2 percent chance of failure in a month—better than a single disk drive.

A second approach is simply duplicating (or "mirroring") hardware such that every record is written on at least two independent devices. This reduces the probability of failure to approximately zero even for parallel systems, and performs about as well as a system with no redundancy. The only flaw with this solution is its expense. It doubles the hardware cost of the file system and provides storage that is not appreciably more robust than the parity scheme that will be discussed in the next section. Parity on striped disk is reliable [Garcia-Molina and Salem, 1988] and uses less disk overhead than mirroring, but it is only reasonable when at least three or four disk drives are attached to each LFS.

4.1.1 Parity Records

When RAIDs, locally striped storage, or mirrored storage are not practical, redundancy can be implemented in the PIFS. Our solution is inspired by the RAID parity disk technique and the similar technique used for striped disks [Garcia-Molina and Salem, 1988]. We keep parity information across *bevies* of LFSs. Any single failed LFS in a bevy can be recovered by calculating the parity on the remaining LFSs in the bevy. Our parity system is distinguished from other similar schemes by the algorithm we use for parity record placement. Our algorithm is tuned to the PIFS performance characteristics.

The division of a PIFS into bevies is a characteristic of each PIFS file. Each PIFS directory entry for a file must indicate the bevy membership of each processor serving a file. This permits various-sized bevies even within a single file.

We interleave parity records with ordinary data records using an algorithm that minimizes the probability of having concurrent data and parity writes on any processor. The constraints on parity placement are:

1. A parity record must not be stored on a LFS that contains one of the associated data records, and the data records must all come from different LFSs.
2. For parallel writes of consecutive records, as many records as possible should use different LFSs for parity records.
3. For non-parallel sequential writes, the average "distance" between parity records and their data records should be maximized. Asynchronous writes can concurrently write a multiple sequential records, building up parallelism until they reach a processor that is busy writing parity for an earlier write. The further the parity records are separated from the data records, the more parallelism asynchronous sequential writes can achieve.

We will show that these constrains are satisfied by the following parity placement rule, illustrated in figure 4.1:

> Consider a bevy as a $(R \times B)$ array where B is the number of processors in the bevy and R is the number of records (data and parity) in each LFS-level file. R will be $B/(B-1)$ times as large as it would have been without parity.
>
> Parity for the bevy is stored in a companion bevy which must contain the same number of processors. The processors responsible for a PIFS file form an array of bevies $G_1, G_2, \ldots, G_n$ where n is even. Companion bevies are spaced as far apart as possible: bevies G_a and G_b are companions if and only if
>
> $$|a - b| = n/2. \tag{4.1}$$
>
> We can identify a record by its row r and column c within its bevy, with numbering starting at zero. Parity records are those for which $c = r \bmod B$. The jth parity record (again counting from zero) is located at $(j, j \bmod B)$.
>
> A data record located i records beyond the jth parity record in its bevy contributes to parity record number $k = B\lfloor j/B \rfloor + i - 1$ in the companion bevy. Conversely, the data records for the kth parity record are located $i = k \bmod B + 1$ records beyond parity records $j_0 = B\lfloor k/B \rfloor$ through $j_{b-1} = B\lfloor k/B \rfloor + B - 2$.

In terms of row and column coordinates, the data record at (r, c) contributes to the companion parity record at (x, y), where

$$\begin{aligned} x &= B\lfloor r/B \rfloor + i - 1 && (4.2) \\ y &= i - 1 && (4.3) \\ i &= (c + B(r \bmod B)) \bmod (B+1). && (4.4) \end{aligned}$$

The parity record at (x, y) is constructed as the XOR of the companion data records at locations of the form (r, c), where

$$\begin{aligned} r &= B\lfloor x/B \rfloor + \lfloor g/B \rfloor && (4.5) \\ c &= g \bmod B && (4.6) \\ g &= y + 1 + t(B+1) \quad 0 \le t \le B-1 && (4.7) \end{aligned}$$

The presence of parity records forces a change in the PIFS placement rule for data records. In a PIFS without parity, the bth record, S, of a file interleaved across p processors would be record $\lfloor b/p \rfloor$ on LFS $b \bmod p$. In effect, S's coordinates in the PIFS as a whole would be $(x, y) = (\lfloor b/p \rfloor, b \bmod p)$. If l is the first LFS in the bevy containing the LFS y, the coordinates of S within the bevy would be $(x, y - l)$ in the absence of parity. In a PIFS with parity, S is still placed in the bevy of LFS y, but its coordinates (r, c) must be chosen in a way that accommodates parity records interspersed with data. We can think of S as data record number $m = xB + y - l$ within its bevy. Since there are $B - 1$ data records per row, $r = \lfloor m/(B-1) \rfloor$. In row r, a parity record occupies column $r \bmod B$. Let $d = m \bmod (B-1)$. Then

$$c = d \text{ if } d < r \bmod B \text{ otherwise } c = d + 1. \qquad (4.8)$$

Data records in figure 4.1 are superscripted with their position in the PIFS file as a whole. When reading a file sequentially, a bevy of size B contributes B consecutive records (*not* $B - 1$) during each complete cycle through the LFSs.

The following three lemmas demonstrate that the parity placement rule obeys the three constraints on parity placement.

Lemma 1 *No parity record is stored on a LFS with a data record used to construct the parity record.*

Since there are at least two bevies, by equation 4.1 the data and the parity records will be in disjoint bevies. Since the parity and data records are written into different bevies they cannot fall on the same LFSs.□

Figure 4.1: Placement of Parity Records for $B = 4, p = 8$

Bevy 0					Bevy 1			
P_4	0_0	1_1	2_2	…	P_0	4_4	5_5	6_6
3_3	P_5	8_0	9_1	…	7_7	P_1	12_4	13_5
10_2	11_3	P_6	16_0	…	14_6	15_7	P_2	20_4
17_1	18_2	19_3	P_7	…	21_5	22_6	23_7	P_3
P_{12}	24_8	25_9	26_{10}	…	P_8	28_{12}	29_{13}	30_{14}
27_{11}	P_{13}	32_8	33_9	…	31_{15}	P_9	36_{12}	37_{13}
34_{10}	35_{11}	P_{14}	40_8	…	38_{14}	39_{15}	P_{10}	44_{12}
41_9	42_{10}	43_{11}	P_{15}	…	45_{13}	46_{14}	47_{15}	P_{11}

This figure shows the placement of data and parity records for two bevies of 4 processors each. The cells labelled P_i denote parity records, and the corresponding data records are labelled with i.

The pattern shown in the figure will repeat through the entire LFS.

Lemma 2 *The chosen placement rule will maximize the number of consecutive records that use different LFSs for their parity records.*

Since parity records are always stored in the companion bevy of a data record, it suffices to show that the placement rule prevents any B consecutive data records in a bevy from using the same parity record.

By equation 4.3 any set of p consecutive records will not share a parity LFS.

Since there are a total of p processors, p is the largest number of records (under any selection rule) that can use different LFSs for parity records. □

Lemma 3 *The chosen parity placement rule maximizes the average distance between parity records and their data records.*

The maximum possible distance between any two records is $p/2$.

A group of B data records cannot be on separate processors and all be at a distance of $p/2$ from their common parity record. Since p is restricted to even values, only one processor can be $p/2$ processors distant.

Our placement rule locates each parity record at a distance of $p/2$ from the center of a contiguous group of data records. The average distance to records in a group of n is

$$Avg_d = \frac{p}{2} - \frac{n}{4} \tag{4.9}$$

for n even and

$$Avg_d = \frac{p}{2} - \frac{n}{4} + \frac{1}{4n} \tag{4.10}$$

for n odd. To achieve a greater average distance, we would have to place more than $n/2$ data records at a distance of $p/2 - n/4$ or greater from the parity record, which is impossible without putting more than one of them on the same LFS, a violation of constraint 1. □

The number of processors between companion bevies is

$$s = \frac{p}{2} - B.$$

The placement rule for the companion bevy places the companion an equal distance before and after the first bevy. Any change in the rule would move the companions closer together.

Theorem 1 *The PIFS parity placement rule satisfies the three constraints on parity placement.*

Lemma 1 shows that the PIFS parity placement rule satisfies the first constraint, lemma 2 shows that the PIFS parity placement rule satisfies the second constraint, and lemma 3 shows that the PIFS parity placement rule satisfies the third constraint.□

4.1.2 Organizational Impact

It is consistent with the PIFS design philosophy to make the parity placement rule visible to the PIF Server and tools. Tools would be seriously hindered if they did not know the rule that controlled placement of records. Since parity is not local to each LFS, it must be maintained above the level of the LFS. We will show in section 4.1.3 that tools can be more efficient if they are allowed to maintain parity records, so we place responsibility for parity maintenance on the PIF Server and on each tool.

Parity updates could form a sequential bottleneck on a single-process PIF Server that managed parity operations. Each write operation spans two processors which would communicate through the PIF Server. This bottleneck can be alleviated by using a multi-processor PIF Server with a process located with each LFS.

The visibility of the parity operation is consistent with the file system's policy toward tools. If parity is defined as part of the PIFS-level structure of the file system, exposing parity structure to tools does not expose the file system to a level of harm that was not already possible; i.e., corruption of the PIFS-level file structure. Furthermore, tools cannot fully optimize an algorithm if they cannot direct an operation to a particular LFS.

Some tools, particularly those tools that produce little output, might not need highly-optimized parity maintenance. Since parity maintenance is fairly involved, it is a good application for a library package or parity server. The slowest and easiest path for tool output would be through the PIF Server's standard or parallel-open interface.

4.1.3 Performance Impact

We would like the PIFS parity scheme to have the minimum possible impact on PIFS performance. This section will sketch algorithms for I/O in the presence of parity records and use them to fix lower bounds for the parallelism and execution time of the basic PIFS operations.

Read A simple, single-record read will only see a small constant factor slowdown from parity. This slowdown comes from the slightly more involved record placement algorithm that would be implemented in the PIF Server, and the additional code that would skip around error recovery routines.

Reads for many records in parallel will lose no parallelism and will only see a small constant factor slowdown. One out of B of the records in PIFS storage will contain parity information, but data reads will ignore those records. The parity placement rule leaves data records distributed such that a read can access $p-1$ records in parallel.

Lemma 4 *Data record placement as modified by the parity mapping leaves at least $p-1$ consecutive records on distinct LFSs.*

The path from an LFS x in bevy G_a back to that LFS or an LFS after it in sequential order as mapped by equation 4.8 passes through every other bevy. This gives $p-B$ concurrent accesses. Added to this are any concurrent accesses in G_a.

Every access between x and the first access in another bevy is contention-free. The access directly before control moves to another bevy reaches the LFS before the LFS handling parity for that row. Contention may occur with the accesses after G_a is reentered. The bevy will be reentered at the LFS after the parity LFS. If x is on the LFS directly after the parity LFS, the return to the bevy will contend immediately, but the bevy will already be fully occupied with B accesses. If x is any other record, then the shift caused by the parity record (by equation 4.8) will cause contention (not necessarily with x) with $B-1$ LFSs active in bevy G_a.□

Sequential Write The obvious algorithm for a simple write with parity maintenance is at least a factor of two slower than the write without parity maintenance. The operation can, however be parallelized enough to reduce the slowdown to about the time for an IPC operation and two bitwise XORs.

The obvious algorithm for a write without parity maintenance is:

```
if the write is for part of a sector
    read sector x into buf
    move new into buf at the right offset
else
    move new into buf
write buf into sector x
```

The obvious algorithm for a write with parity maintenance is:

```
read record x into buf
read parity record into buf3
XOR buf with new at the right offset giving buf2
XOR buf2 with buf3
write buf3 into the parity record
write new into record x
```

Written this way the algorithm entails two reads and two writes with no parallelism. The two reads, however, can easily execute in parallel. The writes can also execute in parallel, but there is an apparent exposure to failure between the two writes; i.e., if one write succeeds and the other fails, parity is incorrect. The exposure is covered by insisting that the file system must know if it experiences a failure. Disk drives provide this notification with a read-after-write protocol. I/O processors can fail invisibly, but this is unlikely and we ignore it.

Provided that we can rely on a failed disk to recover itself from the redundant data in the rest of the bevy, either the data write or the parity write operation can fail without causing data loss. We consider two cases:

- If the data write fails but the parity write succeeds, the error recovery system will recover the data record from the bevy. The recovered data record will have the value it would have had if the data write had succeeded.
- If the data write succeeds but the parity write fails, the error recovery system will recover the parity record from the bevy. The recovered parity record will have the value it would have had if the parity write had succeeded.

The parity placement rule allows p consecutive writes without writing any parity record more than once. Consecutive writes, therefore, cannot produce concurrent updates of a parity record. Random writes can call for as many as $B - 1$ concurrent updates of a parity record. The parity updates will serialize, but they will not expose the system to unrecoverable errors. Any single failure can be recovered by the normal error recovery algorithm. We consider two cases:

- If a single data write fails the parity will reflect the updated data and the data will be recovered to its state after the update.
- If the parity write fails it will be recovered to its state after all the parity updates.

A write can be divided into two parts, one on the processor with the parity disk, the other on the processor with the data disk. These parts are:

```
             Data                                     Parity
read record x into buf                   read parity record into buf
XOR buf with new giving buf2
Send buf2 to parity processor
write new into record x                  receive buf2 from data processor
                                         XOR buf with buf2
                                         write buf2 into parity record
```

Note that the maximum path length is (read,XOR,IPC,XOR,write). This is the best performance a PIFS with parity support can achieve on writes. We will call a write with parity that updates the parity disk concurrently with the data disk a write with *full parallelism* to indicate that the operation is as parallel as possible.

Parallel Write A parallel write may be a write through the parallel open interface or it may be driven by a tool. In either case, let us say the parallel write updates m records from a PIFS file concurrently. Since a PIFS is optimized for sequential access, the canonical parallel write is for m *consecutive* records.

There are two measures of parallel-write performance: time for a parallel write, and maximum true parallelism. These measures interact. If the upper bound on full parallelism is $p/2$, then a write for more than $p/2$ records cannot run its data and parity updates in parallel. A write for more than $p/2$ records will execute more slowly than a write for fewer than $p/2$ records.

A trivial upper bound on write parallelism is set by the number of available processors, p. We can reduce this upper bound on parallelism with an observation. One record's data processor may be the same as *another* record's parity processor. Since every write involves two concurrent operations, p processors can support no more than $p/2$ update/parity-update operations in parallel.

Theorem 2 *The PIFS parity placement rule supports full parallelism for $p/2-1$ concurrent writes of consecutive records.*

By reasoning similar to that of the proof of lemma 4, any group of $p/2$ consecutive records will be stored on $p/2$ distinct processors. Moreover, the corresponding parity records will be stored on $p/2$ distinct processors that overlap at most one of the processors writing the data. A PIFS with parity therefore supports full parallelism for any write of $x \leq p/2 - 1$ consecutive records.□

All parallel writes that require more than $p/2-1$ processors will sequentialize in the LFS such that some data and parity updates do not execute in parallel.

Tools The performance degradation associated with parity maintenance is plainly visible to tools. A large class of tools for PIFSs without parity support can run with almost no IPC. The requirements of parity maintenance shrink this class substantially. Most tools that write to a parallel interleaved file will have to communicate with remote processors. A simple implementation of a tool that updates a file will include a remote parity update for every data update. This will force even simple tools to contend with communication bandwidth. A tool that simply copies a file without modification is an exception to this rule. Since the data records are not changed, the parity records remain unchanged and the entire file can be copied without regard for parity.

A tool can make some useful optimizations on the standard parity-preserving operations. If it is willing to buffer a sufficient amount of output, a tool can write the $B-1$ data records that share a parity record with only one write to that parity record.

Theorem 3 *A tool that writes a group of data records and their parity records concurrently has no unusual exposure to failure.*

The file system with parity is recoverable from any single failure within a bevy and its companion bevy. Two concurrent failures constitute an unrecoverable error. So assume that a single write per bevy-pair fails.

If any single write in a bevy pair fails and all the others succeed, recovery will repair the failed write.

So a write of $n(B-1)$ data records and their n parity records will recover to correct data even if there is one failure per bevy pair.$\square$

On computers that can send large messages almost as quickly as small messages, tools can reduce IPC overhead by building bundles of data for other processors. When parity records are involved, the parity records can be bundled with data records.

4.1.4 Recovery Protocol

A PIFS with parity implemented degrades gracefully as disks fail. It will continue to run even with a failed processor in each bevy pair, but each failed bevy will lose all parallelism. When a disk or processor fails, every read or write in its bevy involves all the processors in that bevy. Every read to a failed disk becomes a *recovery read,* a parallel read to the entire bevy plus one read in the companion bevy followed by a parity calculation to regenerate the lost record. Parity writes to the damaged disk are ignored, but data writes entail a recovery read followed by an ordinary parity record update.

When the damaged disk is repaired or replaced it can be restored online with a tool that runs on the damaged bevies. The tool does a lock and a recovery read for each record (including parity records) on the new disk, then writes the new value without updating parity information kept on other disks and unlocks the record.

4.1.5 The Selection of Bevy Size

Since the probability of data loss is the probability of two concurrently failed disks in companion bevies, reliability increases as B decreases, but storage overhead for parity also increases as B decreases. In addition, smaller values of B give better performance for concurrent writes, and faster recovery from failure.

A PIFS with parity will only fail if two drives in a pair of companion bevies are out of service concurrently. If we suppose that failed drives are repaired once a week, and use 60,000 hour (357 week) MTTF drives, the probability of one drive in a bevy pair failing in a week is: $(1-(1-1/357)^{2B}) = (1-0.9972^{2B})$. The probability of data loss is therefore $P_{dl} = (1-0.9972^{B})(1-0.9972^{(2B-2)})$

which gives a mean time to data loss (MTTDL) of $1/P_{dl}$ weeks. For a bevy of four drives, that works out to a MTTDL of 96 years for 1/3 extra disk storage. For a bevy of 25 disks the MTTDL is 111 weeks for four percent extra space. This MTTDL contrasts well with the 4 week MTTDL for a 100-disk system with no parity protection, and is about a third of the 357-week MTTF of a single drive.

Given the $(p/2) - 1$ bound on full parallelism for parallel writes, the loss of B processors of parallelism for each failure, the higher reliability of smaller bevies, and the rapid decrease in additional hardware cost as bevy size increases, bevies should probably be no larger than 10 processors. A 10-LFS bevy would have excellent reliability by conventional standards: 19-year MTTDL if disk replacement and recovery takes no more than a week.

4.2 File System Maintenance

Hardware and administrative changes frequently affect a file system. The configuration of a file system should be flexible enough to gracefully adjust to a mutable environment. The storage capacity of a typical file system expands rapidly. Old technology hardware is incrementally replaced with newer technology. File system parameters are tuned to improve throughput or optimize crucial applications. A usable file system must make such activities convenient. It must support partial reconfiguration without dumping and restoring the entire file system.

4.2.1 Changes to p

The PIFS design allows casual addition of disks and processors. It also supports on-line migration from a processor that will be removed.

The PIFS directory entry for each file includes a list of the processors the file uses. This list is constructed when the file is created and used whenever the file is opened. The set of disks available to new files can be controlled by the system administrator.

When the file system gains a new processor the system administrator can simply add that processor to the list of processors available for file allocation. From that time on files that are created can use that processor. The processor will gradually come toward full utilization. If this gradual process is unsatisfactory the system administrator can copy some files. When a file is copied it will read from the old processor set and write to the new processor set.

Removing a processor from the available processor list will only affect file creation. The processor will not be included in any new creations and the processor will gradually drain. When only long-lived files remain the system administrator can copy the remaining files and decommission the processor.

The PIFS create operation can be instructed to allocate various classes of files across different subsets of the available processors. This capability would let files that are frequently used together occupy disjoint sets of processors, and let the system administrator lighten the load on less powerful processors. Different user communities could occupy their own sets of processors while making the entire set of processors available for the jobs that need full I/O parallelism.

The location and span of files within a large PIFS can be changed with mechanisms analogous to those used for adding and removing processors from the file system. When the allocation rules for a class of files are changed, files in the class will migrate to the new allocation. The residue can be moved with copy operations.

4.2.2 Changing Disks

Actual disk drives are not really a concern at the PIFS level since they are handled by the LFSs. The only constraint a PIFS puts on disk changes is that the LFS must not lose data in a reconfiguration. Disk changes may move files within or between disks, or even cause a file to span multiple disks. The PIFS knows LFS files by their LFS names, and it knows records by their location within LFS files so only the names and internal structure of LFS files must remain fixed.

4.2.3 Backup and Recovery

A PIFS with parity support can be unusually reliable, but it would be foolish to rely too heavily on the file system's integrity. The probability of failure cannot be reduced to zero. Like any file system, a PIFS should be backed up regularly. This is, of course, an operational tradeoff. Several hours of backups, cubic yards of storage space, and plenty of administrative worry may be too much trouble when the file system will run for decades without data loss. The main reasons for backups in such an environment are user carelessness combined with file system hostility (mistakenly deleted files), and major site disasters. No parity system will protect against an earthquake that drops a building on the disk farm.

The backup operation is an appropriate application for a tool. The operation can be phrased as "take every record in the file system and put it somewhere safe;" that is, there are no inter-record dependencies. A backup tool could therefore run at the parallelism of the file system. If every LFS includes a backup device, backup can run entirely at the LFS level in time n/p where n is the total size of the PIFS storage and p is the number of LFSs.

Recovery is equivalent to backup. Recovery from dismounted media, however, is not likely to be convenient. A backup from a large conventional file system will span many tapes but the operator need only mount the tape that actually contains the file in question. A PIFS restore will call for at least one tape on every processor that has a tape drive. The solutions to this problem involve hardware or administrative policies. If restoration of a file from backup was only provided in case of data loss, the operation would be rare and it would move a large amount of data (enough to restore a bevy). Very large capacity, inexpensive backup devices such as WORM drives, might keep a large history of backups on line. Restores from the recent past could read from the online backup volumes without operator assistance.

A comparatively low performance backup would read files sequentially and write all the records for a file onto one tape (or more if the file ran off the end of a tape volume). This would slow the backup by passing almost every record through IPC, but it would not call for an unusual number of tape mounts to restore a single file.

4.3 Portability

One of the design goals for PIFSs was that they be adaptable to any sound MIMD architecture, and we believe that we achieved that goal. We support that assertion by briefly discussing the implementation issues for a PIFS in several hardware and software environments. We have experimental support for the portability of the PIFS concept in that the Bridge implementation depends on no architectural feature that is not common to parallel computers.

4.3.1 Software Environments

A PIFS consists of multiple processes on specific processors. At least some of the processes should be protected and privileged. These are not unusual demands to place on an operating system for a MIMD architecture.

To the extent that I/O hardware is localized to processors, the PIFS and tools must be able to name the processors and bind processes to them. This may force the PIF Server and tools to put hooks into the kernel. Operating systems use the process as an abstraction of a processor and tend to hide the names of the physical processors. This gives the operating system the freedom to schedule processes without considering any binding the process may have to a processor. The underlying assumption is that processors are equivalent. From the viewpoint of a PIFS this is an invalid assumption. Processors are characterized by their distance from each I/O device and their distance from other PIFS processes.

A PIFS requires a command that starts a process on a named processor and locks it on that processor. If the operating system does not provide that service to user processes, then the PIFS must be a part of the operating system.

Protection and security require some operating system support. Privileged components of the file system should be able to act on behalf of user processes. When the PIF Server forwards a command to an LFS, the command should be treated as if it came from the user process, not the PIF Server. Tools should be able to access LFS files either on behalf of the process that invoked the tool or as the owner of the tool.

If the PIF Server and certain tools are trusted by the system resource protection mechanism, they can implement appropriate security. Capability-based systems meet a PIFS's security requirements easily. Unix-like setuid mechanisms are also sufficient.

4.3.2 Hardware Environments

The PIFS design was created with three different parallel architectures in mind: non-uniform memory access (NUMA) shared memory multiprocessors, uniform memory access (UMA) bus-based parallel architectures, and message-based architectures such as hypercubes. This concern is reflected mainly in the PIFS's concentration on locality. A PIFS treates interprocessor communication as a scarce resource. This contributes to the scalability of the design; it also makes the PIFS design suitable for machines with less facile IPC than the NUMA Butterfly.

The Bridge implementation of a PIFS uses message passing constructs for all its IPC. We chose to use this approach because it makes IPC clear and because it is efficient on all parallel architectures. For NUMA, a PIFS was designed to minimize remote references (and implicitly memory contention). For UMA, the performance role of remote memory references in NUMA is replaced with cache misses and cache coherency traffic. These are not usually as important as remote references are for NUMA, but they are not negligible especially in aggregate. Many UMA machines are bus-based, and a bus-based machine is limited by its bus bandwidth. By minimizing cache misses and coherency operations, a PIFS minimizes bus traffic.

A PIFS is constructed of large, long-lived components that use large-grained parallelism. For the architectures we have in mind, support for fine-grained parallelism relies on fast thread creation and very fast IPC. A PIFS benefits slightly from fast process creation and substantially from very fast IPC, but it does not rely on them as a design based on fine-grained parallelism would.

4.4 The Local File Systems

A parallel interleaved file system is designed to work with an existing local file system. Since the PIF Server offers transparent access to every feature of the LFS that does not expose the parallel nature of the PIFS, a PIFS can be superimposed on an LFS almost invisibly. A PIFS makes few requirements of the LFS, but the PIF Server and tools must support logical record structure if the LFS does not, and LFS support for variable-length logical records is very helpful.

4.4.1 The Importance of Logical Records

Files structured in terms of logical records are crucial to a PIFS. Fixed length logical records are easy for the file system to support and sufficient for a PIFS. Variable-length logical records are useful and aesthetically appealing, but a PIFS (like a conventional file system) functions adequately without them.

A PIFS positions data according to the LFS's unit of allocation. This favors the file systems found on MVS, VMS and other operating systems designed for commercial work. It places the Unix file system and other similar designs at a serious disadvantage. Consider a program that operates on lines of data. If the file is written as an array of lines, the program can ask for a group of lines and have them delivered to different processes. If the file is written as an array of bytes or other fixed-size units, some entity must scan the file to find line terminators, and the PIF Server and tools cannot locate lines without reading the file.

Fixed-length records can be implemented in an LFS or simulated by the PIF Server on top of an LFS that does not implement logical records. The simulation by the PIFS requires two changes to a non-record-oriented interface:

- The file creation request must specify the record length. That datum must be saved as a characteristic of the file and recalled whenever the file is opened.
- Read and write must restrict data length to an integral number of records.

If these changes to the interface are not acceptable, the PIFS can use a default logical record length and split the I/O stream into logical records without regard to semantics. A PIFS with such useless logical records would function, but tools would be seriously hindered.

Variable-length records are aesthetically appealing and useful. The logical record structure of a file is chosen when the file is created. Typically, the size of the logical records can be fixed or allowed to vary within some range. Fixed-size records are easy for the LFS to manage, but they work best when logical units of data are of a fixed size. Lines of text are an excellent example of units of data that have a variable length. There is a long tradition of 80-character lines based on one of the original fixed-size record media (cards). Variable-length records accommodate long lines without wasting space on filler for all the shorter lines. This filler wastes disk space and I/O bandwidth.

For sequential access, a file of variable-length records may be more efficient than a file of fixed-length records, but for random access variable-length records require some form of index. The physical disk address of a variable length record depends on the length of the preceding records. The file system requires at least a table of the number of records per disk track to locate a track for the disk hardware to search.

The length of a variable-length logical record is part of the information content of the record. Parity records and recovery programs must consider this length value. A parity record must be longer than the maximum-length data record by the length of the length specifier. This is usually 2 bytes, so a parity record for variable-length records would have a length of the maximum data record length plus two bytes.

The designer of an application chooses efficient use of storage and I/O bandwidth, or fast random access for each file the program will use. The tradeoffs for a PIFS are the same as for a conventional file system.

4.4.2 Security

Our PIFS design has no direct provision for security, but it supports any security mechanism supported by the LFS. This is an important feature for two reasons: (1) The PIFS is automatically compatible with any existing security. (2) Tools are not a security exposure and can access LFSs without special security considerations. The only security measure required of the PIF Server is that it pass the source of each I/O request to the LFSs.

Each LFS is a self-sufficient file system. Any security implemented at the LFS level will apply to the components of parallel interleaved files stored on each LFS. If each LFS component of a parallel file is protected, and each access to the LFS reflects the true origin of the request, LFS security will be effective.

If security were localized in the PIF Server, tools would have to be "well behaved" in ways that are hard to enforce. Convention requires that tools open, create, and delete parallel interleaved files through the PIF Server. If LFSs have no protection mechanism they cannot defend against improper access by tools, so we are forced to trust tools or force them through an interface that implements protection. Protection based on trust is not protection, so secure protection can reside in the LFSs or in an interface that lies between the LFSs and tools. From the point of view of a PIFS, a layer between the LFSs and tools is indistinguishable from LFSs that implement protection.

4.5 Disk Drives

The PIFS strategy involves large numbers of I/O processors and attached storage devices. The performance of the storage devices should be matched to the performance of the LFSs and the aggregate storage capacity of the devices should meet the users' needs. Multiprocessor computers with many processors tend to use powerful microprocessors. The processors in current multiprocessors match, or slightly outpace, the best PC-type hard disk drives. If technology continues on its present trends, within a few years a node will require several such disks or a small RAID. Coincidentally, the class of disk drives that balance best with an LFS also have the best price/performance. When performance is measured in terms of capacity and transfer rate, price performance favors the middle of the performance range.

Table 4.1 shows price performance information for a variety of representative disk drives. The figures in this table are estimates, mostly taken from sales literature and phone calls to salesmen, but they are accurate enough for our purposes. The figure at the bottom of the table plots data rate per dollar against the cost of a single device (and its associated controller). The variation in cost is enough that the log of the price is used on the cost axis. The circled point represents the CDC Wren IV disk drive.

The Wren IV and the dozen or so other drives in and slightly below its class fall at the knee of the price performance curve.[1] Although the high-priced drives offer poor price performance compared to less expensive devices, simple high-performance disk drives are good for single-processor file systems. Their primary (and sufficient) features are that they are easy to manage and reliable. A PIFS, however, cannot use a monolithic disk drive. Each PIFS processor requires at least one disk drive, and unless each processor supporting an LFS was in the mainframe class, very high performance drives would balance poorly with the processors. Wren-class drives match well with a typical LFS and offer the best available price performance.

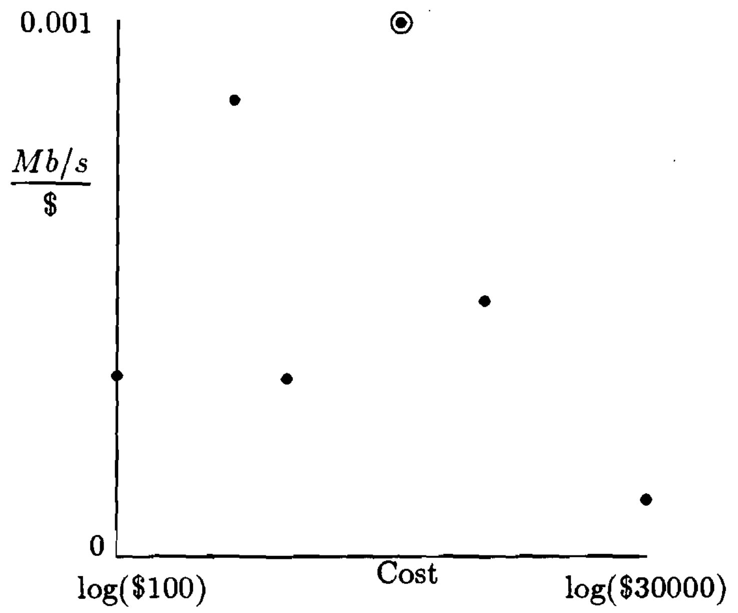
Historically, the performance of processors has improved faster than disk drives. We expect that in time a typical processor's I/O requirements will move out of the range of a mid-priced disk drive. This does not necessarily move a

[1]The storage array price performance part of table 4.1 shows that the devices that offer a good data rate per dollar also have a good storage capacity per dollar. The laser disk is much the best device by this measure, but its data rate and seek rate rule it out.

Table 4.1: Price Performance for Assorted Disk Drives

Single Devices				
Device	Price	Transfer rate	Latency	Capacity
IBM 3380 with a Portion of a 3880	28000	3 Mb/s	17+8 ms	1260 Mb
Fujitsu Eagle with SMD controller	5000	2.4 Mb/s	18+8 ms	575 Mb
Wren IV with embedded controller	2000	2 Mb/s	15+8 ms	340Mb
Laser disk drive	600	0.2 Mb/s	500 ms	600 Mb
Seagate hard disk with a minimal controller	350	0.3 Mb/s	45+8 ms	20 Mb
Floppy drive with WE Chip	100	35 kb/s	95+100 ms	660 kb

Storage Array Price/Performance based on a $12,000 storage system				
Device	Price	Transfer rate	Latency	Capacity
3380/3880	28000	3 Mb/s	17+8 ms	1260 Mb
2 Eagles	10000	4.8 Mb/s	17+8 ms	1150 Mb
6 Wren IVs	12000	12 Mb/s	15+8 ms	1800 Mb
20 Laser disks	12000	4 Mb/s	500 ms	12000 Mb
34 Seagates	11900	10.2 Mb/s	85+8 ms	680 Mb
120 Floppy drives	12000	3.5 Mb/s	95+100 ms	100 Mb



PIFS off the knee of the price/performance curve. RAIDs can match the highest price/performance drives to the data requirements of the faster processors.

4.6 Summary

In this chapter we considered issues that would affect a production parallel interleaved file system.

A PIFS is record oriented. Either the LFS must support logical records, or the PIF Server must add that support to the file system interface. Support for fixed-length logical records is relatively painless at the PIF Server level, but many file systems support logical records. Variable-length record support is more difficult to implement than fixed-length records. Though a PIFS does not require them, variable-length records conserve disk space and I/O bandwidth, and they are more aesthetically pleasing than fixed-length records with filler.

We established that a parity scheme would make storage on a PIFS more reliable than an ordinary file system with conventional disks. We discussed file system maintenance procedures and showed that a PIFS can be reconfigured and tuned about as easily as a conventional file system. A brief discussion of the PIFS design's dependencies on its hardware and software environment demonstrated that the PIFS design is appropriate for any reasonable parallel architecture. We showed that a PIFS easily inherits the full power of the file system security mechanism supported by its LFS. Finally we showed that the disk drives that meet the storage and performance requirements of a PIFS are also a good price/performance choice. As microprocessors evolve these drives may need to become RAID devices, providing higher bandwidth and storage capacity, and potentially solving the PIFS reliability problem in hardware as well.

5 Tools

The *tool* mechanism gives programs efficient, low-level access to a parallel interleaved file system's storage. Tools are the essential reason a PIFS scales well, and a tool's special position in the file system does not require arcane parallel algorithm techniques or unusual programming discipline.

Our argument for the practicality of tools is based on experimental evidence supplemented by analysis. Specifically, we built three substantially different tools and designed several others. Each tool uses a small variation on the most standard algorithm for the problem, and each tool scales within the range we could test (2 to 32 LFSs). Our analysis shows that they could be expected to scale far beyond that range.

The appropriate measures for our experiments are: (1) did the obvious algorithm adapt easily to a tool implementation, (2) does the resulting tool perform well, and (3) does the tool scale well? Our ground rules for tool design and implementation were:

- Use an ordinary algorithm with as few adaptations to parallelism as possible.
- Fix major performance bugs but do not *tune* the tools.

We chose to implement tools for three well-understood problems: copy, sort, and matrix transpose. The problems cover a range from obvious candidates for implementation as a tool to a unusual selection. The problems best suited to implementation as tools treat logical records as independent entities. These tools always give $O(p)$ speedup for problems that are large enough that run time dominates startup cost. From this class we chose *copy*. We chose file sorting as the function for a tool that exercises relationships between records. Copying and sorting are obvious benchmarks for a file system. Sorting is so basic to file systems that tools could be judged entirely on their support for file sorting. According to Knuth's volume on sorting and searching, [Knuth, 1973, page 3], sorting averages over 25 percent of running time across all mainframe computers and often reaches over half the computing time at particular sites.

Matrix transpose on an array of bits is not a common file operation, but we chose to create a transpose tool because we expected it to challenge the tool interface with extreme volumes of IPC.

In each case our tool used an obvious algorithm and scaled well. Our copy tool is an almost-unchanged version of the obvious read/write loop. It achieves linear speedup until performance becomes so good that non-parallel startup time becomes noticeable. The sort tool is based on a simple merge sort. Its speedup curve is linear through the range we measured, and we extrapolate good speedup beyond 160 processors. The matrix transpose tool uses an iterative version of the standard recursive algorithm. It gives good speedup through the range of measurement and should continue to benefit from additional processors until startup time dominates run time.

5.1 The Tool Interface

The tool interface consists of two PIF Server operations, access to the LFSs, and a set of conventions.

Tools use a PIF Server operation that returns IPC handles for all the LFSs, and another PIF Server operation that returns placement information for a PIFS file. Together, these operations return enough information for a tool to open a file at the LFS level. Once it has that access it must follow some guidelines to prevent damage to the PIFS structure.

- A tool must call the PIF Server to create and delete PIFS files. It must not use the LFS delete or rename operations to alter the characteristics of a file that is part of the PIFS.

- A tool must never set the number of records in a LFS constituent of a PIFS file such that it contains fewer records than both its neighbors or more than one fewer records than the LFS before it in the PIFS file. This prevents "gaps" in the PIFS file.

- Any PIFS files resulting from a tool's operation must be robust under a PIFS copy operation. Since a copy operation may change the placement of a PIFS file and the number of LFSs participating in the file, this means that records must not have a strong binding to an LFS and they must not depend on the locality of other records.[1]

[1] A tool that compressed each local component of a PIFS without recording the context for the compression would leave a file that was not robust under copy. Copy onto a different number of LFSs would leave a file that would not uncompress successfully.

A tool may create and delete LFS files that are not part of the PIFS.

Tools tend to follow a standard outline:

```
Connect to the PIF Server
Get the LFS Handles from the Server
Get info on the tool's input from the Server
Create output files if any and get info on them
For each LFS participating in a file
    Fork a process on the LFSs local processor
        Perform the bulk of the tool's processing
            in the local processes
Wait for all the processes to complete
Collect and return any results from the local processes
```

5.2 Copy Tool

Copying a file is the simplest example of a large class of file operations. These are the operations that can be stated:

> Record number x in the output file is function F of record x in the input file.

For copying, F is the identity function. Other tools in this family include:

Transform where F maps strings to other strings with the restriction that strings may not span records.

Compress where any compression algorithm is applied independently to each logical record producing a smaller output record.

Scan where the input file is tokenized into the output file. This type of scanner is limited to languages in which tokens do not span records (lines).

Search for keys that do not span records.

All the tools in this class require time $= O(n/p + \log p)$ where n is the size of the file, p is the number of processors, n/p represents run time, and $\log p$ represents startup time.

Other operations such as *count* and *summarize* are closely related to copying. They extract some small amount of information from records and produce no file as output. If the information is very small compared to the size of the file, these tools can run in time $O(n/p + \log p)$, but tools that return a substantial volume of information may require $T = O(n/p + n \log p)$ to pass the file and return $O(n)$ information up a tree.

5.2.1 Design

For our copy tool we started with the ordinary file copy algorithm:

```
open infile
open outfile
for each record $i$ in infile
    read record $i$ from infile into buffer
    write from buffer into outfile record $i$
close infile
close outfile
```

This algorithm converts into a tool by starting a tool process local to each LFS and using those processes to copy the records on their local LFS.

```
                                                        Time
open infile                                             $C_{open}$
create outfile                                          $C_{create}$
for each LFS start a process                            $C_{fork}$
    [In the processes]
    let L_infile be the local component of infile
    let L_outfile be the local component of outfile
    for each record $i$ in L_infile                     $C_{loop}$
        read record $i$ from L_infile into buffer       $C_{read}$
        write from buffer into L_outfile record $i$     $C_{write}$
for each LFS wait for process termination               $C_{join}$
close infile                                            $C_{close}$
close outfile                                           $C_{close}$
```

5.2.2 Analysis

The expected execution time for this algorithm is:

$$T = C_{create} + C_{open} + 2C_{close} + p(C_{fork} + C_{join}) + (n/p)(C_{read} + C_{write} + C_{loop}).$$

It could be reduced slightly by starting the processes more efficiently, but the improvement would be trivial for substantial files and p in the range of our experiment.

Table 5.1 contains constants for the analysis of tools. The values in the column labelled "measured" were either taken from [Dibble, 1986] or measured

Table 5.1: Constant Values for Simple Tool Operations

Name	Value in ms		
	Measured	Tuned	Optimized
C_{create}	689	689	100
C_{open}	100	100	30
C_{close}	0	0	10
C_{fork}	15	15	15
C_{join}	3	3	3
C_{read}	18	10	3
C_{write}	44	29	5
C_{loop}	0	0	0

Table 5.2: Calculated versus Measured Copy Tool Performance for a 20M File

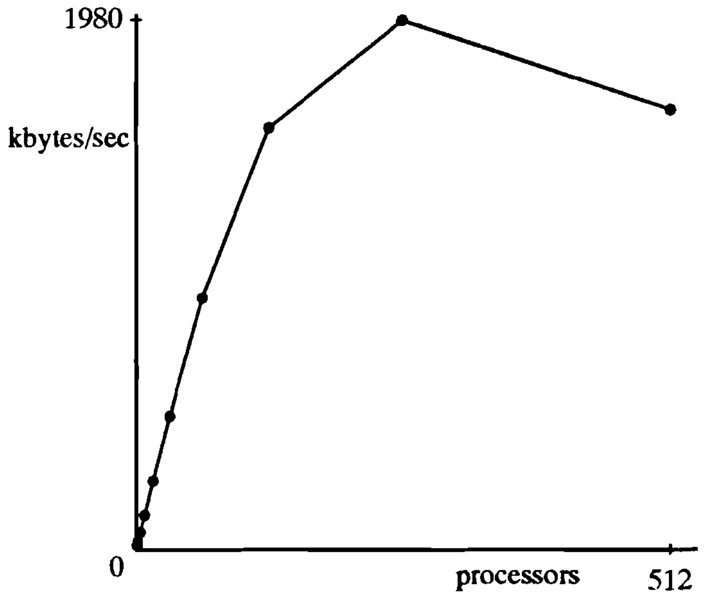
p	Expected		Measured	
	Time (sec)	Rate (k/sec)	Time (sec)	Rate (k/sec)
1	1271	16.1		
2	635	32.3	625	32.8
4	318	64.4	316	64.8
8	160	128.0	159	128.6
16	80	256.0	81	252.9
32	41	499.5	42	489.2
64	22	940.3		
128	13	1573.8		
256	10	1977.4		
512	12	1640.4		

in our sorting tool. The last two columns in the table will be used later. The second column contains estimated constants for a tool with better buffering than the copy tool implementation, and the third column contains estimates for a high-performance LFS. Substituting the constants from table 5.1 column one into the above equation gives:

$$T = 689 + 100 + 0 + p(15 + 3) + (n/p)(18 + 44)$$

with the resulting times shown in table 5.2 and figure 5.1.

Figure 5.1: Predicted Copy Tool Performance



5.2.3 Measurements

The measured performance of the copy tool is also shown in table 5.2.[2] These copy rates are within about 2 percent of the predicted rates. The error is mainly attributable to imprecise prediction of startup cost. The Chrysalis fork operation does not have simple performance characteristics. Our startup costs varies by at least a factor of two without obvious explanation. This imprecision is not noticeable in the ten minute run time for a two-processor copy, but amounts to about half a second for 32 processors, which is about one percent of the run time.

Figure 5.2 plots the expected performance of copytool as a dashed line and the measured performance as a solid line. The graph shows that the implementation closely matches our predictions, and scales well.

5.2.4 Extrapolations from Copytool

The EFS local file system works best when it is asked to read or write a single file sequentially. When it is asked to switch between two files, as this copy tool requires, its read and write times increase by about 30 percent. We speculate that the copytool could be optimized by causing it to read several records, then

[2]Throughout this chapter measurements are accurate to well within a millisecond. Many measurements of single operations vary by 10 to 100 percent; for these values we use an average over enough samples that three runs do not vary in the least significant digit reported.

Figure 5.2: Predicted versus Measured Performance for Copy Tool

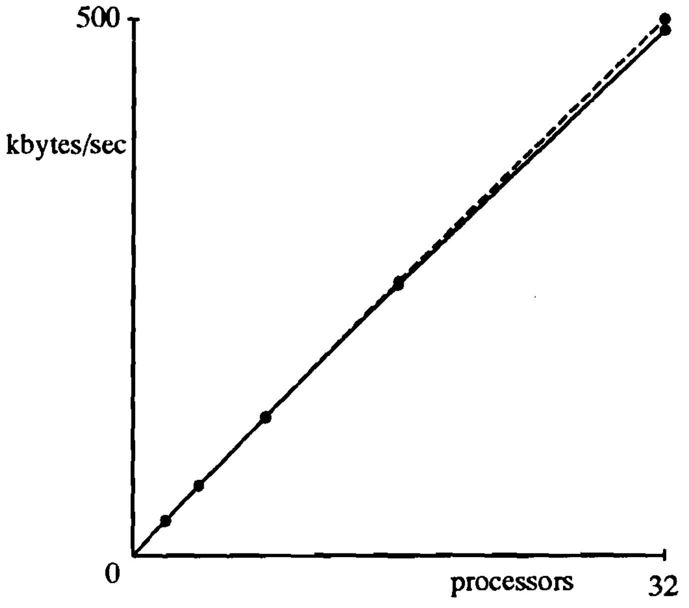


Table 5.3: Predicted Optimized Copy Tool Performance for a 20M File

Optimized Tool			High-Performance LFS			Maximum Performance		
p	Time (sec)	Rate (k/sec)	p	Time (sec)	Rate (k/sec)	p	Time (sec)	Rate (k/sec)
1	800	25.6	1	164	124.9	1	164	124.9
2	400	51.2	2	82	249.5	2	82.1	249.5
4	201	102.1	4	41	497.5	4	41.1	498.3
8	101	203.0	8	21	986.8	8	20.7	991.1
16	51	400.8	16	11	1921.6	16	10.4	1961.3
32	26	775.0	32	6	3515.3	32	5.34	3835.2
64	14	1420.2	64	4	5330.6	64	2.80	7319.5
128	9	2194.4	128	4	5514.3	128	1.54	13333.3
256	9	2404.6	256	5	3808.1	256	.914	22407.0
512	12	1770.9	512	10	2118.8	512	.612	33464.1
						1024	.470	43574.5
						2048	.408	50196.1
						4096	.386	53057.0

write them. The best time EFS can manage for reads is an average of about 10 ms; for writes EFS can reach 29 ms. The projected performance for that optimization is given in the optimized tool section of table 5.3 using constants from the second column of table 5.1. A moderately high performance file system can read about a megabyte per second and write at about 667 kilobytes per second. The file creation time for such a file system is no more than 100 milliseconds, and the open time for Bridge would decrease due to faster access time for the Bridge directory. If such a file system were used, the copytool would be expected to reach the predictions in the High-Performance LFS section of table 5.3, using constants from the third column of table 5.1.

The predictions for the high-performance file system show that startup time becomes a noticeable performance problem for 16 and 32 processors, and even at that our analysis assumes that the computer can start processes without contention for some resource. A copy tool with such a fast LFS and such comparatively small files (20 megabytes) should probably start its processes in parallel. A tool that used a $\log p$ startup algorithm would continue to profit from additional parallelism past 2048 processors. Predictions for a tool of this design are given in the section of table 5.2 labelled *Maximum Performance.*

Tools Similar to Copytool

We designed a number of tools to test the boundaries of the copytool class. File compression was one of the more interesting exercises.

File compression poses a challenge for a PIFS tool. Record-by-record compression is in the same class as copying a file, but it doesn't work well for the best compression algorithms or for LFSs without variable-length records. Run length compression and adaptive Huffman coding [Gallager, 1978] work well in a simple tool, but Lempel-Ziv [Welch, 1984] coding works best with large amounts of input data. A single record is not enough.

For the compression techniques that work best with lots of input data, a compression tool can use the entire local portion of a file. Each record must still be mapped to one output record, but the compression algorithm can use context from other records in the LFS file. Each output record in such a compressed file must identify the records on which it depends, in case the file is reorganized.

An adaptive compression tool can process an LFS file by compressing each record in the context of the previous record. This will give it enough input data to build useful compression tables, but copying the output file to a different number of LFSs may make the file decompress incorrectly. This difficulty can be countered by adding links to the compressed data, or by saving in the file the value of p for the file when it was compressed. If the decompression tool can calculate the PIFS record number of the next record compressed by the same tool component, a decompression tool will be able to decompress the file after arbitrary copying. In the usual case file decompression will be a local operation. Copied files may not decompress locally, but they will decompress correctly.

5.3 Sort Tool

File sorting is an obvious choice for a PIFS tool. Sorting is a heavily-studied problem with great practical importance. A sort tool also represents an interesting class of file operations: tools with data-dependent IPC. The $O(n \log n)$ sorting algorithms compare each record to $\log n$ other records. The pairings are determined by the data, so by supplying a file of random data, we get largely random comparisons, and many of the comparisons involved in sorting a PIFS file will compare records on different processors.

Our implementation of a sorttool uses a simple version of the standard mergesort algorithm. Our analysis shows that the tool should run in $T = O((n/p) \log(n/p))$, and our measurements show that the implementation meets the prediction.

Our sorttool design relies on the large difference between IPC speed and I/O speed. It is reasonable to assume that there will be a large difference, though not necessarily as large as in Bridge. We used our analysis to predict the behavior of sorttool in a number of hardware environments:

- With fast IPC, sorttool obtains nearly linear speedup past 160 processors and continues to benefit beyond 512 processors.
- With slow IPC, sorttool's speedup falls off around 30 processors.
- With fast IPC and a fast LFS, we can expect good speedup past 40 processors.
- With a fast LFS and slow IPC, our sort algorithm is not useful. It serializes at about four processors.

In accordance with our ground rules, we implemented the standard file sorting algorithm with the minimum possible adjustments to the tool environment. The result was good speedup, but unexceptional absolute performance. At the end of this section we suggest improvements to the algorithm that would give much better absolute performance with a fast LFS and fast IPC and still give speedup past 16 processors.

5.3.1 Design

An astonishing number of parallel sorting algorithms have been proposed [Bitton *et al.*, 1984]. Most are ill-suited for external sorting because they access data randomly, read the data too often, or require a very large number of processors. Among the few algorithms specifically designed for external sorting, several require special-purpose hardware, assume a very small number of processors, or have significant phases during which only a fraction of the processors or disks are active [Bitton *et al.*, 1984; Kwan, 1986]. In a recent paper Beck, Bitton, and Wilkinson [Beck *et al.*, 1988] detail their construction of a functional parallel external sort. They chose an algorithm similar to ours: local quicksort followed by parallel mergesort. Since they used comparatively few processors (five), they were able to pipeline the entire multi-phase merge with one disk read and one write. This gave them excellent performance, but poor speedup past three processors and no speedup of the merge stage in the worst case.

For our algorithm we chose a more or less conventional merge sort because it is easy to understand and has a straightforward parallelization. Our primary objective was to demonstrate that Bridge can provide significant speedups for common file operations. Rather than invest a great deal of effort in constructing the fastest possible sort, we deliberately chose to limit ourselves to an implementation with reasonable performance that could be derived from its sequential counterpart with only modest effort. We conjecture that most I/O-intensive applications can be implemented on Bridge with straightforward parallel versions of conventional algorithms; our merge sort constitutes a case in point.

A sequential external merge sort makes no unusual demands on the file system (no random access, indexing, etc.) and runs in $O(N \log N)$ time. Given a parallel merge algorithm, a log-depth parallel merge sort is easy to write. With p processors and N records a parallel merge sort concurrently builds p sorted runs of length N/p. It then merges the sorted runs in a $\log p$ depth merge tree. Pseudo-code for this algorithm appears in figure 5.3. The first phase of the algorithm sorts the records on each LFS independently. The second phase merges the sorted records in neighboring pairs of LFSs. Assuming for the sake of simplicity that p is a power of two, the final phase merges the records from two collections of LFSs, each consisting of $p/2$ processors.

Since the final merge phase must read and write the entire file with whatever degree of parallelism the merge algorithm provides, the time to run the merge algorithm on the entire file places a limit on the performance of the merge sort as a whole. Before the last pass, parallelism can come from both the merge algorithm (running within a subset of nodes) and from the structure of the merge sort itself (running in multiple subsets simultaneously). At depth t in the merge tree, the merge sort algorithm runs 2^t merges in parallel, each of which uses $p/2^t$ processors. If a merge on k processors can be fully k-way parallel (as

```
In parallel perform local external sorts on each LFS.
Consider each resulting file to be "interleaved"
across only one LFS.
x := 2
while (x <= p)
    Merge pairs of files in parallel
    Consider the new files to be interleaved
        across x processors
    Discard the old files in parallel
    x := 2 * x
```

Figure 5.3: Merge Sort Pseudo-Code

we argue below for reasonable values of k), then each merge phase as a whole will be p-way parallel. The entire merge sort will display nearly linear performance improvements as nodes are added to Bridge.

5.3.2 Merging Parallel Interleaved Files

A parallel interleaved file can be viewed as a whole or, at the other extreme, as p sub-files, each local to a node. It may also be regarded as some intermediate number of sub-files, each of which spans a non-trivial subset of the file system nodes. The merge algorithm takes two sub-files, each spread across k nodes, and produces a single sub-file spread across $2k$ nodes. To do so it employs two sets of reading processes (one set for each of the source sub-files, one process per node) and one set of writing processes (again, one process per node).

The algorithm passes a token among the reading processes of the two source sub-files. The token contains the least unwritten key from the other source sub-file and the location of the process ready to write the next record of the output sub-file. When a process receives the token it compares the key in the token to the least unwritten key among its source records. If the key in the token is greater than or equal to its local key, the process sends an output record to the appropriate writing process, and forwards the token to the next reading process in its sub-file. If the key in the token is less than the local key, the process builds a new token with its own key and address, and sends that token back to the originator of the token it received.

Special cases are required to deal with termination, but the algorithm generally follows the outline above. Figure 5.4 contains more detailed pseudo-code.

The parallelism of the merge algorithm is limited by sequential forwarding

```
token {
      WriteAll
      Key
      Source
      Number
      }

Setup for the merge                                                    $k_1$
Read a record                                                          -
If this process initiates the merge                                    -
      Build a token {false, key, MyName, 0}                            -
            where key is the first key in the local file               -
      Send the token to the first reading process for the other file   -

Loop
      Receive token                                                    $k_2$
      If (token.Key $\geq$ record.key and not EOF) or token.WriteAll   -
            Increment token.Number                                     $k_3$
            Send token to next reading process for this file           -
            Send an output record to the writing process               $k_4$
                  on LFS (token.Number $-1$) mod $p$                   -
            Read a new record                                          $k_5$
      Else
            Build a token {EOF, file key, MyName, token.Number}        $k_6$
            Send the new token to token.Source                         -
While not EOF

If (not token.WriteAll)                                                $k_7$
      Build a token {true, MAXKEY, MyName, token.Number}               $k_6$
      Send the token to old token.source                               -
```

Figure 5.4: Merge Pseudo-Code

of the token. On at least every other hop, however, the process with the token initiates a disk read and write at the same time it forwards the token. For disk sorting on a machine like the Butterfly, we will show that the token can undergo approximately two hundred fifty hops in the time required for the parallel read and write. This implies that the sequential component will be entirely hidden by I/O latency on configurations of well over 100 processors. Performance should scale almost linearly with p within that range.

5.3.3 Analysis

Merge

The parallel merge algorithm is a close analog of the standard sequential merge. The token is never passed three times without and intervening write, and all records are written in nondecreasing order. The program therefore writes all of its input as sorted output and halts.

For the purposes of this analysis, let p be the number of nodes across which the output sub-file is to be interleaved and let N be the number of records in this file. Let us also refer to source and destination sub-files simply as "files." Each source file will of course be interleaved over half as many nodes as the destination file, and will consist of half as many records. Moreover, the merge steps that make up an overall merge sort will often manipulate significantly fewer records than comprise the entire file, and will use significantly fewer nodes.

We will call the sequential part of the algorithm its *limiting section.* The rest of the code can execute in parallel with disk I/O. The critical code is:

```
receive token                                                        $k_2$
if (token.Key $\geq$ record.key and not EOF) or token.WriteAll   case 1   -
      Increment token.Number                                         $k_3$
      Pass token to next process                                     -
      ...
else                                                     case 2
      Build a token {false, file key, MyName, token.Number}          $k_6$
      Send the new token to token.Source                             -
```

There are two cases in the loop, one taking time $T_{act} = k_2 + k_3$ and the other taking time $T_{pass} = k_2 + k_6$. Since the first case will be executed N times before all the records are written and the algorithm terminates, the total time used by that code will be $T_{act}N$. The second case is executed whenever the token passes from one file to the other.

A "run" is a string of records merged from the same file. A crossover stands between runs. If C is the number of crossovers in the merge, the total time used by case 2 is $T_{pass}C$.

To analyze the behavior of the algorithm as a whole, we must consider the extent to which the limiting section can execute in parallel with disk I/O. The second case of the loop has no parallel part, but case 1 includes both a read and a write: $T_{read} = k_5$ and $T_{write} = k_4$. The limiting section has the potential to become significant when a process of a source file finishes reading its next record before the token returns, or when a process of the destination file finishes writing a record before being given another one. The time required for the token to return to the same reading process can be as small as $T_{act}p/2$, or as large as $T_{act}(p/2 + N/2) + T_{pass}\min(C, p/2)$, since it is possible for the entire other source file to be traversed before returning. Similarly, the time that elapses between writes to the same output process can be as small as $T_{act}p$, or as large as $T_{act}p + T_{pass}\min(C, p)$. On average, the time to complete either a read or a write "circuit" should be $p(T_{act} + T_{pass}C/N)$. The extent to which individual circuits deviate from the average will depend on the uniformity of the distribution of crossovers.

We want to discover the number of processors that can be used effectively to sort. We must therefore determine the point at which I/O begins to wait for the sequential token passing. Average case behavior can only be used with care because an unusually brief circuit saves no time (I/O is still the limiting factor), whereas an unusually long circuit loses time by allowing the sequential component to dominate. Fortunately, we can make the fluctuations negligible in practice by allowing source file processes to read ahead and write behind.

Since the output file is interleaved across the same disks as the input files, we will obtain linear speedup so long as every disk is kept continually busy reading *or* writing records that need to be read or written. A source file process whose disk has nothing else to do should simply read ahead. Finite buffer space will not constitute a problem until the limiting section begins to dominate overall. In our implementation, the timing anomalies caused by uneven crossover distribution (on random input data) are rendered negligible with only one record of read ahead.

Execution time for the merge algorithm as a whole can be approximated as

$$T_{merge} = T_{fixed} + \frac{N}{p}T_{delete} + \max\left(NT_{act} + CT_{pass}, \frac{N}{p}T_{disk}\right), \qquad (5.1)$$

where $T_{disk} = T_{read} + T_{write}$, and T_{fixed} is overhead independent of p. Each pass of the merge sort algorithm should delete its temporary files; since the delete operation for our LFS takes time k_{10} per record, $T_{delete} = k_{10}$. T_{fixed} includes the time required within each process for initialization and finalization. It also includes the time T_{eof} required in one of the token circuits to recognize the end of the first source file and build a WriteAll token. Per-phase initialization time is k_1. $T_{eof} = k_2 + k_7 + k_6$. If we let k_{11} be per-phase termination time, we have $T_{fixed} = k_1 + (k_2 + k_7 + k_6) + k_{11}$.

The merge algorithm will display linear speedup with p so long as p is small enough to keep the minimum token-passing time below the I/O times; in other words, so long as

$$T_{read} \geq T_{act}\frac{p}{2} \text{ and } T_{write} \geq T_{act}p,$$

i.e.

$$p \leq p_{max} = \min\left(\frac{2T_{read}}{T_{act}}, \frac{T_{write}}{T_{act}}\right) \tag{5.2}$$

If crossovers are close to uniformly distributed, the algorithm should actually display linear speedup so long as p is small enough to keep the ***average*** token-passing time below the I/O time; in other words, so long as

$$T_{disk} \geq p\left(T_{act} + \frac{C}{N}T_{pass}\right),$$

i.e.

$$p \leq \overline{p_{max}} = \frac{T_{disk}}{T_{act} + \frac{C}{N}T_{pass}} \tag{5.3}$$

To find an expression for $\overline{p_{max}}$ independent of C, we must determine the expected number of crossovers in a file. This can be calculated by dividing the number of possible occurrences of a crossover by the number of possible file interleavings.[3] Given source files *file1* with n_1 records and *file2* with n_2 records and $N = n_1 + n_2$, there are $\binom{N}{n_1}$ possible interleavings of the files. To identify a particular crossover in a particular interleaving, we note that the crossover can take place in any of $N - 1$ positions between records, and can be from *file1* to *file2* or from *file2* to *file1*. Since it determines the source file of two records, the crossover can be embedded in $\binom{N-2}{n_1-1}$ different interleavings. There are therefore $2(N-1)\binom{N-2}{n_1-1}$ different occurrences of crossovers among all the different interleavings, or an average of

$$\overline{C} = \frac{2(N-1)\binom{N-2}{n_1-1}}{\binom{N}{n_1}} = \frac{2n_1\binom{N-1}{n_1}}{\frac{N}{N-n_1}\binom{N-1}{n_1}} = \frac{2n_1n_2}{N}\ .$$

In our case, where $n_1 = n_2$, $\overline{C} = N/2$. Putting this back into equations 5.3 and 5.1 yields

$$\overline{p_{max}} = \frac{T_{disk}}{T_{act} + \frac{1}{2}T_{pass}} \tag{5.4}$$

[3]Thanks to Ron Loui for this insight.

and

$$\overline{T_{merge}} = T_{fixed} + \frac{N}{p}T_{delete} + \max\left(NT_{act} + \frac{N}{2}T_{pass},\ \frac{N}{p}T_{disk}\right), \qquad (5.5)$$

which we will henceforth use as our value for T_{merge}.

Merge Sort

The local external sorts in the first phase of the merge sort are ordinary external sorts. Any external sorting utility will serve for this phase. Standard external sorts will run (in parallel with each other) in time $O(\frac{N}{p}\log\frac{N}{p})$. We can approximate this as

$$T_{local} = C_{local}\frac{N}{p}\left(1 + \log\frac{N}{pB}\right),$$

where B is the size in records of the in-core sort buffer. The 1 inside the parentheses accounts for one initial read and write of each record, used to produce sorted runs the size of the buffer. Internal sort time is negligible compared to the cost of I/O for merging.

Referring back to figure 5.3, the merge algorithm executes $\log_2 p$ phases, for $x = 2, 4, 8, \ldots, p$ (again assuming that p is a power of two). Phase x runs p/x merges, each of which uses x processors to merge Nx/p records. The expected time for phase x is therefore

$$T_x = T_{fixed} + \frac{N}{p}T_{delete} + \max\left(\frac{Nx}{p}T_{act} + \frac{Nx}{2p}T_{pass},\ \frac{N}{p}T_{disk}\right).$$

If $T_{startup}$ is the time required to create $2p$ processes and to verify their termination, then the expected time for the merge sort as a whole is

$$\begin{aligned} T_{sort} &= T_{startup} + T_{local} + \sum_{x=2,4,8,\ldots,p} T_x & (5.6)\\ &= T_{startup} + T_{local} + \log p\left(T_{fixed} + \frac{N}{p}T_{delete}\right) & (5.7)\\ &\quad + \sum_{x=2^i\ 2\le x\le p_{max}} \frac{N}{p}T_{disk} + \sum_{x=2^i\ p_{max}<x\le p} \frac{Nx}{p}T_{act} + \frac{Nx}{2p}T_{pass}. \end{aligned}$$

If p is small enough that I/O always dominates, this is

$$T_{sort} = T_{startup} + T_{local} + \log p\left(T_{fixed} + \frac{N}{p}(T_{disk} + T_{delete})\right).$$

Otherwise we have

$$
\begin{aligned}
T_{sort} &= T_{startup} + T_{local} + \lfloor \log \overline{p_{max}} \rfloor \left(T_{fixed} + \frac{N}{p}(T_{disk} + T_{delete}) \right) \quad (5.8) \\
&\quad + \frac{N}{p}\left(T_{act} + \frac{1}{2}T_{pass} \right)\left(2p - 2^{\lfloor \log \overline{p_{max}} \rfloor + 1} \right).
\end{aligned}
$$

Startup

Time is required to create the $2p$ processes used for local sorting and for merging, and to detect termination when they are done. Let $T_{create} = k_8$ be the time required to create a process and to notice its completion. If startup proceeds sequentially we will have $T_{startup} = 2pT_{create}$; if startup proceeds in a tree, we will have $T_{startup} = T_{create}\log(2p) = T_{create}(\log p + 1)$. In either case, the time required to start a few hundred processes will be insignificant in comparison to the time required to read and write the records of a large file several times. In our implementation of Bridge, $T_{startup}$ and T_{fixed} both make a negligible contribution to T_{sort}. Below $\overline{p_{max}}$ we have

$$
\begin{aligned}
T_{sort} &\approx T_{local} + \frac{N}{p}\log p\,(T_{disk} + T_{delete}) \quad (5.9) \\
&= \frac{N}{p}\left[C_{local}\left(1 + \log\frac{N}{pB} \right) + \log p\,(T_{disk} + T_{delete}) \right]. \quad (5.10)
\end{aligned}
$$

In our implementation we obtain less than ideal speedup with increasing p primarily because $C_{local} < T_{disk} + T_{delete}$.

5.3.4 Performance

Given appropriate values for constants, our analysis of the sort tool can be used to predict execution time (T_{sort}) and level of available parallelism ($\overline{p_{max}}$) for a wide range of processors, disks, and architectures. In an attempt to evaluate the accuracy of the prediction, we have measured the sort tool's performance on our implementation of Bridge.

Actual and predicted performance figures are shown in table 5.4. The better-than-linear performance "improvements" with increasing p in the local sort are not remarkable; they reflect the fact that each individual processor has fewer records to sort. The predicted ratios of merge time to total sort time shown in figure 5.6 illustrate the effect of the small amounts of local data.

Our local sort algorithm is relatively naive: a simple two-way external merge with 500-record internal sort buffers. Initial runs are sorted internally with quicksort. The local sort runs at about a quarter the speed of the Unix sort

Table 5.4: Sort Tool Performance (10 Mbyte file)

Processors	Merge Phases		Local Sort		Merge Sort Total		
	Minutes		Minutes		Minutes		Rate
	Meas.	Pred.	Meas.	Pred.	Meas.	Pred.	(k/sec)
2	7.8	7.68	19.6	19.58	27.4	27.26	6.26
4	7.6	7.68	7.6	7.83	15.2	15.51	11.00
8	5.7	5.76	2.7	2.94	8.4	8.70	19.62
16	3.8	3.84	1.0	0.98	4.8	4.82	35.41
32	2.4	2.40	0.3	0.24	2.7	2.65	64.52
64		1.44		0.12		1.56	109.21
128		0.84		0.06		0.90	189.28
256		0.51		0.03		0.54	318.06
512		0.33		0.02		0.35	491.29

utility. Presumably one would want to employ a high-performance local sort for production use, but since we are interested primarily in demonstrating that the concept of parallel interleaved files extends well to very large numbers of nodes, there was no need to tune the local sort for our experiments. A highly tuned local sort is of limited value even for an optimized sort tool. When p is larger than 8 or 16, the local sorts make almost no contribution to total elapsed time (see figure 5.6). From a practical point of view, the extra code space required for a fancy local sort would also have used memory resources that we needed for our buffers and simulated disks. Finally, the advantages of a multi-way merge would largely be offset by degraded read and write times, since EFS uses a track buffering scheme tuned for sequential access.

Our predicted performance figures differ only slightly from measured performance between 2 and 32 processors. A detailed examination of timing data suggests that the remaining inaccuracy stems from minor contention for the local file systems that is not accounted for in the analysis.

Every process may have a read and a write in progress concurrently. The local file systems, however, can only service one I/O request at a time. The performance predictions assume that I/O requests are distributed uniformly across the LFSs, but this is not exactly true. Write requests cycle through all the LFSs in a set order; read requests depend on crossover distribution and so are somewhat random. Occasionally a read and a write will collide at an LFS and one of the requests will wait. If the merge algorithm blocked immediately for each read and write, these collisions could degrade performance badly. We therefore block for reads only when the data is actually used and for writes only when a subsequent write is issued. This strategy reduces the program's sensitivity to collisions. Additional buffering would be likely to reduce it even further.

Figure 5.5 plots predicted and actual performance figures up to 32 processors

Figure 5.5: Predicted Versus Actual Performance

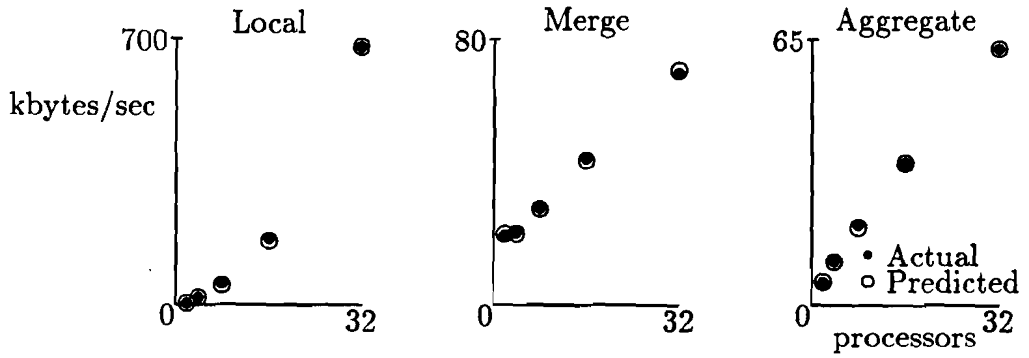
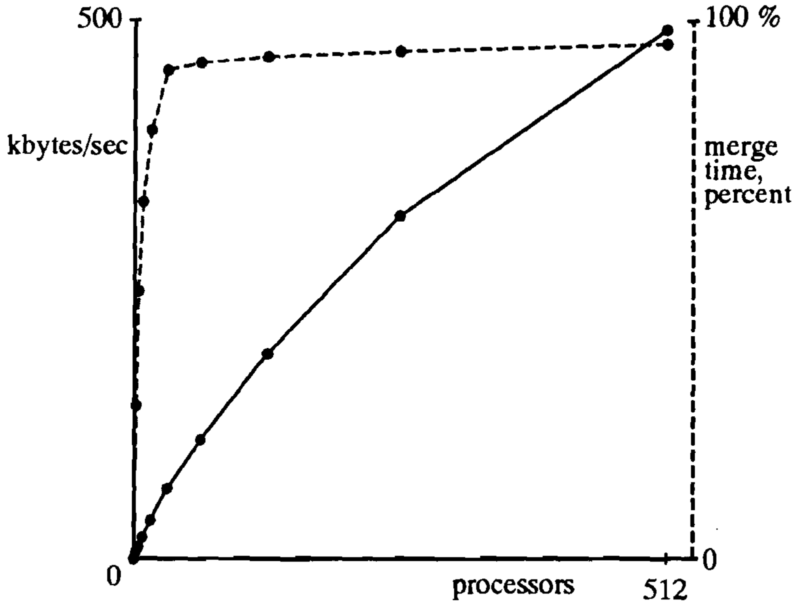


Figure 5.6: Predicted Aggregate Performance



for the local sorts, the merge phases, and the overall sort tool. Figure 5.6 extends these graphs with predicted performance on larger numbers of nodes. The dotted line in the second graph plots the percentage of total execution time devoted to parallel merge phases (as opposed to local sorts). Speedup begins to taper off noticeably beyond the center of the graph, where I/O ceases to dominate sequential token passing. Performance continues to improve, but at a slower rate. The merge that uses 256 processors to merge two files will run at its IPC speed. The earlier stages of the merge and the local sorts will, however, run with p-way parallelism. This should cause the algorithm to show some improvement with thousands of processors even though (as shown in the following section) the last stages of the merge will reach a speedup of only about 160.

Table 5.5: Constant Values for Sorttool

Constant	Value	Source
k_1 – merge phase setup time	20000 μsec	measured (avg.)
maximum	30951 μsec	measured
minimum	116 μsec	measured
k_{11} – merge phase termination time	0 μsec	hidden in start of next phase
k_2 – receive token and following tests	253 μsec	measured
k_3 – update and forward token	28 μsec	measured
k_4 – write a record (T_{write})	45000 μsec	measured
k_5 – read a record (T_{read})	25000 μsec	measured
k_6 – build and send a token	31 μsec	measured
k_7 – simple *if*	45 μsec	measured
k_8 – start a process (T_{create})	48000 μsec	measured
k_9 – end a process	300 μsec	measured
k_{10} – delete a record (T_{delete})	20000 μsec	measured
C_{local}	45900 μsec	measured
T_{disk}	70000 μsec	$k_4 + k_5$
T_{act}	281 μsec	$k_2 + k_3$
T_{pass}	284 μsec	$k_2 + k_6$
T_{fixed}	20300 μsec	$k_1 + k_2 + k_6 + k_7 + k_{11}$
p_{max}	160	see equation 5.2
$\overline{p_{max}}$	165	see equation 5.3 or 5.4

5.3.5 Values for Constants

The figures in table 5.5 were obtained by inserting timing code in the merge program. The timing statements inevitably introduced additional overhead, but not enough to make a difference in performance. The predicted times in tables 5.4 and 5.6 were calculated with equation 5.8 and the constants from table 5.5.

Perhaps the most important figures in table 5.5 are the derived values for p_{max} and $\overline{p_{max}}$. With the I/O, communication, and computation times found in our implementation, and with fewer than 160 processors in use, there is no way for a process to complete an I/O operation before being asked to perform another one.

5.3.6 Extrapolation from Sorttool

Slow Communication

Equation 5.4 indicates that the number of processors that can be used effectively to sort depends on the ratio of disk speed to processor and communication speed. Very fast disks or very slow communication would cause $\overline{p_{max}}$ to drop. To illustrate this effect, consider the implementation of Bridge on a collection

Table 5.6: Theoretical Merge Sort performance with Slow Communication

Processors	Merge	Local Sort	Total	Rate
2	7.68 min	19.58 min	27.26 min	6.26 k/sec
4	7.68 min	7.83 min	15.51 min	11.00 k/sec
8	5.76 min	2.94 min	8.70 min	19.62 k/sec
16	4.15 min	0.98 min	5.13 min	33.28 k/sec
32	3.24 min	0.24 min	3.48 min	49.02 k/sec
64	2.73 min	0.12 min	2.85 min	59.89 k/sec
128	2.45 min	0.06 min	2.51 min	68.07 k/sec
256	2.29 min	0.03 min	2.32 min	73.48 k/sec
512	2.21 min	0.02 min	2.22 min	76.75 k/sec

of workstations with processor performance comparable to the Butterfly and a file system similar to EFS, but with communication time increased by a factor of 25, to about two milliseconds per message.

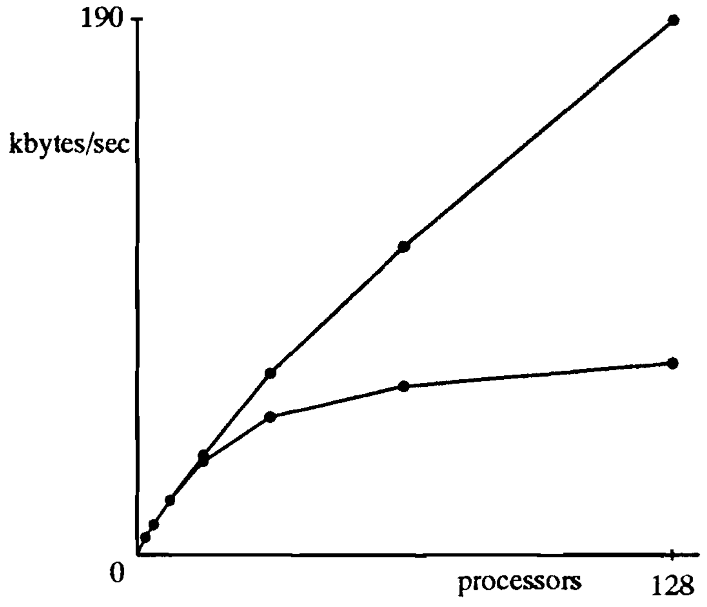
Equation 5.10 indicates that for small numbers of processors the hypothetical system should perform only slightly worse than our Butterfly version. The 4 ms increase in T_{write} has relatively little effect. However, T_{act} and T_{pass} will increase to 4120 and 4130 μsec, respectively, driving $\overline{p_{max}}$ down to 12. Predicted performance figures are shown in table 5.6 and figure 5.7. Beyond about 32 processors the serialization of later merge phases prevents the system with slow communication form obtaining additional speedup. As in the case of fast communication, $\overline{p_{max}}$ provides a very conservative estimate of available parallelism; continued speedup of early merge phases allows the system to take some advantage of additional processors well beyond $\overline{p_{max}}$.

Very Large Files

The local sort phase will scale with p so long as the number of records per processor does not become ridiculously small. Even so, as shown in figure 5.6, a 10 megabyte file is not large enough to let the local sorts contribute significantly to the total sort time when there are more than about eight processors. For much larger files, the effect of the local parallel sort is visible further out on the curve, but the program still suffers a major performance loss when the merge step sequentializes.

Figure 5.8 shows the predicted performance of the sort tool on files of size 10 Mbytes, 100 Mbytes, one Gbyte, and 10 Gbytes. The solid lines plot performance, as in earlier figures. The dotted lines use the scale on the right-hand margin of the graph, and plot the percentage of total time consumed by the parallel merge phases. The top pair of lines are the same as in figure 5.6.

Figure 5.7: Performance with Slow Communication versus Fast Communication



Merge rate (kilobytes/sec) is independent of file size, so the effect of file size on the overall rate of mergesort is entirely due to the $n \log n$ performance of the local sort phase. The local sort phase makes mergesort slower for large files, and causes the local sort to take a larger percentage of the run time for larger files.

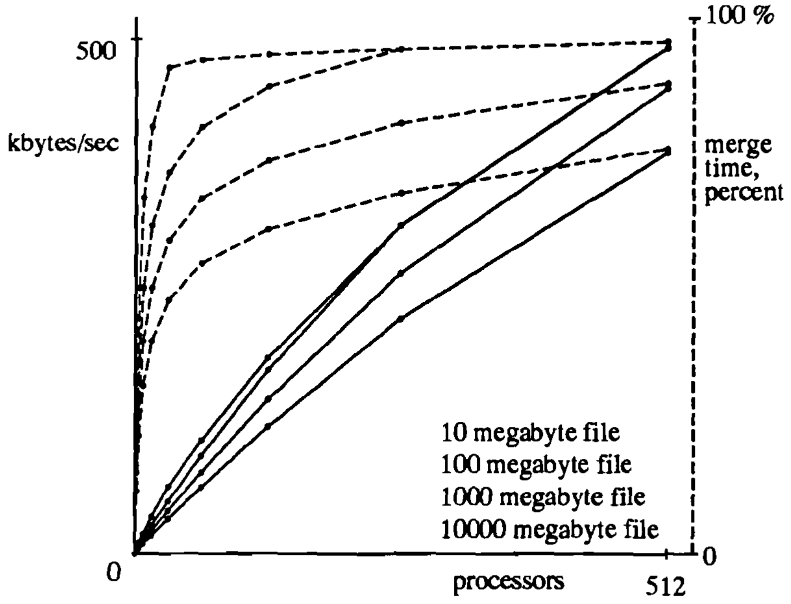
The local sort uses a 500 kilobyte buffer. The time to sort the buffer is not visible on the graph because it is so small in comparison to the time to read and write. The lines describing the performance of the mergesort for 10 and 100 megabyte files therefore meet at 128 processors. At that point the local sorts complete with no local merging. Their I/O time is linear in the file size, so the sort rate (and consequently mergesort rate) is effectively independent of file size.

High-Performance Sorttool

We can project from sorttool's performance to the performance of a highly tuned tool with a high-performance LFS. We will make this extrapolation in two steps: first we will simply modify the constants so they reflect a high-performance LFS, then we will investigate the effect of an improved algorithm.

We will use the estimated figures for LFS performance given in table 5.1 and additional figures in table 5.7. The new figures are reasonable estimates for a good (but not exceptional) file system, and the IPC times are halved, a reasonable guess for a tuned sort tool and a processor that could sustain the file

Figure 5.8: The Effect of File Size on Performance



The curves nearer to the top of the graph represent smaller files.

Table 5.7: Estimated Constants for a Tuned Sorttool

Name	Time (ms)
C_{delete}	100 per file
T_{act}	140
T_{pass}	142

system we are assuming. With these new values $\overline{p_{max}}$ becomes

$$\overline{p_{max}} = \frac{8000}{140 + 142/2} \approx 37.$$

Since $T_{startup}$ does not depend on I/O speed, and C_{local} scales with I/O speed, we substitute $C_{\text{local}} = T_{\text{disk}}$ (because a high performance LFS would not give the buffered reads of the local sort stage a great advantage), and $T_{\text{delete}} = 0$ into equation 5.10.

$$T_{sort} \approx 8000\frac{N}{p}\left(1 + \log\frac{N}{pB} + \log p\right).$$

At 32 processors and a 10 megabyte file, this sorts about a half a megabyte per second. This a very respectable rate for a disk sort, but 32 processors is not the ceiling we would like for scalability.

Improved Sorting Algorithms

The performance of the local sort phase can be improved by simply converting it to a multi-way merge. The best we can do is one read and one write per record for the local sort:

```
While not end of file
    Read a buffer-full of input and sort it
    Write the buffer
Start the parallel merge phase
Buffer the first record from each sorted run
Use the input buffers as the source for the merge phase
```

This improved local sort algorithm would substantially improve the local sort phase, but it does nothing for $\overline{p_{max}}$. With 32 processors, the sort would run at about 3/4 of a megabyte per second. Performance gains fall off rapidly after 64 processors (see figure 5.9). Going from 256 processors to 512 processors barely adds ten percent to performance.

The $P_{max} = 37$ limit appears to be an inescapable restraint on token-passing sort algorithms on our hypothetical high-performance hardware. Below p_{max}, however, we can continue to improve the performance of the sort.

The parallel merge can be expected to work better if it can run without intermediate files. The merge passes run concurrently, with each merge pass handing its output to the next pass. Beck, Bitton, and Wilkinson [Beck *et al.*, 1988] used a variation of this algorithm that arranged the merge processes in a tree. Files were at the bottom of the tree and sorted records flowed from the root. Unfortunately, this strategy suffers a bottleneck at the root. Our algorithm avoids the bottleneck by parallelizing each merge phase.

A pipelined version of the sorttool seems promising. The parallel merge algorithm used in the sort tool can also be used with the passes running concurrently. The output from the local sort need not be written to a file. The local sort can fill a buffer that will serve as input for the local representative of the first pass of the parallel merge. Likewise, each pass of the parallel merge except the last can pass its output though a buffer to the next pass. The final pass must finally write the data to record the sorted file.

Together with the optimized local sort, the pipelined parallel merge would cut the mergesort to $2n$ reads and $2n$ writes. All the intermediate reads and writes would be replaced with messages between sort passes. If we use k_6 and k_2 from table 5.5 for intermediate write and read respectively, each partition of each pass of the parallel merge becomes essentially sequential but very fast. The passes run concurrently with a pipe-filling delay.

Figure 5.9: Sorttool Performance with Preliminary Optimizations

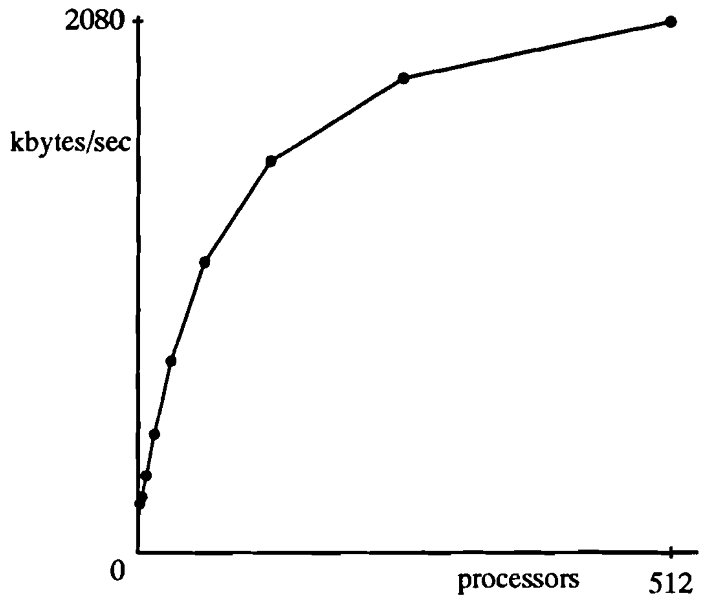
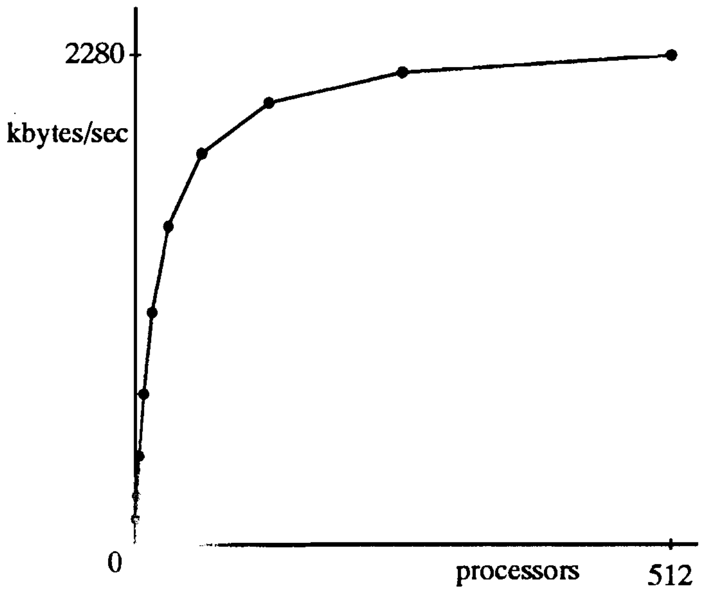


Table 5.8: Sorttool Performance with Pipelined Algorithm

Procs	Run Time	k/sec
1	173 sec.	118.7
2	91 sec.	226.1
4	50 sec.	412.7
8	29 sec.	702.6
16	20 sec.	1083.1
32	14 sec.	1485.3
64	11 sec.	1823.8
128	10 sec.	2058.2
256	9 sec.	2199.6
512	9 sec.	2277.7

Figure 5.10: Sorttool Performance with Pipelined Algorithm



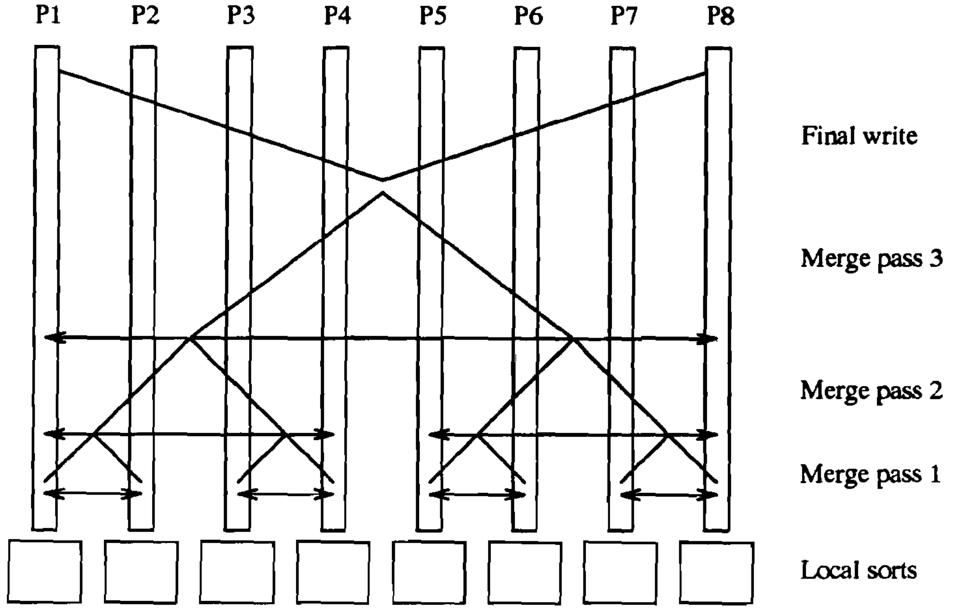
A sort tool th [illegible]ses this algorithm should be fast and scale well up to the point where the [illegible]orts a file in $T_{act} + CT_{pass}/N$ per record. Only the local sort phase and t[illegible]al merge pass do any I/O. This reduces read and write to messages in the sa[illegible] performance class as the token-passing messages, and the merges will sequentialize on token passing. There will, however, still be multiple parallel merges in each pass except the last. These parallel merges will give the first pass $p/2$-way parallelism and pass x will have $p/2^x$-way parallelism.

All the merge passes will run in parallel with one another, and, since the final pass has no parallelism, the final pass will only wait for data during the local sorts and while the first record is passed up through a $\log p$-depth tree of merge passes. For large p, the running time of this sort tool would be:

$$\begin{aligned}
T_{\text{sort}} &= T_{\text{localsort}} + T_{\text{priming}} + T_{\text{finalpass}} \\
T_{\text{localsort}} &\approx \frac{N}{p}(T_{read} + T_{write}) \\
T_{\text{priming}} &\approx T_{read} + (\log(p) - 1)(T_{act} + CT_{pass}/N) \\
T_{\text{finalpass}} &\approx NT_{act} + CT_{pass} \\
T_{\text{sort}} &\approx \frac{N}{p}(T_{read} + T_{write}) + T_{read} \\
&\quad +(\log(p) - 1)(T_{act} + CT_{pass}/N) + NT_{act} + CT_{pass}
\end{aligned} \tag{5.11}$$

with results shown in figure 5.10.

Figure 5.11: Pipelined Mergesort



The tight bottleneck imposed by the final merge pass will probably make the processor contention from the other passes negligible, so it is not reflected in equation 5.11.

5.4 Transpose Tool

We chose to implement file copy and file sort tools partly because they perform heavily used operations, but we felt that our good results might, in part, result from selecting problems that were known to be efficient operations on conventional file systems. We wanted to build a tool that challenged the tool concept, and matrix transpose seemed a good candidate. Matrix transpose on disk files is not a heavily-used file operation, but it is potentially useful. Matrix transposition is a useful mathematical operation, and it is also used to rotate images by 90 degrees. A 2500 dpi bitmap image for a typesetting machine uses almost 7.5 megabytes per 8 by 10 page. That is large, but possible to manipulate in RAM. A color image could easily be ten times the size of a monochrome image, and even today 2500 dpi is ordinary resolution and more than twice that resolution is available—few computers would casually transpose a 75 or even a 30 megabyte array.

Matrix transpose is a simple operation as shown in figure 5.12. Our preliminary analysis showed that the obvious parallelization of that matrix transpose algorithm would run in $T = O(n^2/p)$ where n is the number of records (rows)

Figure 5.12: Standard Transpose Algorithm

```
for x = 0 to bound
    for y = 0 to bound
        tmp = array[x,y]
        array[x,y] = array[y,x]
        array[y,x] = tmp
```

in the file. That parallelization involves reading a row and writing each element of the row to the appropriate record: n writes for each of n rows. On further thought we settled on a $T = O(n \log n)$ algorithm.

5.4.1 Design

The non-locality of the iterative matrix transpose algorithm shown in figure 5.12 was hard to map efficiently onto a file, but the recursive transpose algorithm shown in figure 5.13 has better locality. We converted the recursive transpose algorithm to an iterative form and adjusted it slightly to adapt it to largely sequential file access. Figure 5.14 shows the input and the output of the transpose algorithm on a representative 4×4 array. The double lines show the sub-matrix size the algorithm is considering.

The conversion of the algorithm in figure 5.13 to a file processing tool depended on the following observations:

- Each row of the matrix can be represented by a logical record.
- Transpose pass k divides each record into 2^k sections (which we will call grains). Half of these grains will remain stationary in this pass. The other half of the grains swap with grains in a different record.
- A record (row) can be divided into its stationary and moving grains without reference to any other record.
- Each Record in the output of a transpose pass is the result of the local stationary grains, and the moving grains from a single record (that may be remote).

Each local component of the transpose tool follows the outline in figure 5.15.

The tool does not continue with the file-based transpose all the way down to a grain length of one. When the grain size, b, is small enough that local memory

Figure 5.13: Recursive Matrix Transpose

```
Transpose(lowx, lowy, highx, highy)
    midx = (lowx + highx)/2
    midy = (lowy + highy)/2
    if(highx > lowx)
        Divide the input matrix into quadrants.
        Swap the upper right quadrant with the lower left quadrant
        and transpose all four quadrants.
        SwapBlock(midx, highx, lowy, midy,
            lowx, midx, midy, highy)
        Transpose(lowx, lowy, midx, midy)
        Transpose(midx, lowy, highx, midy)
        Transpose(lowx, midy, midx, highy)
        Transpose(midx, midy, highx, highy)

SwapBlock(lowx1, highx1, lowy1, highy1, lowx2, highx2, lowy2, highy2)
    y2 = lowy2
    for y1 = lowy1 to highy1 − 1
        x2 = lowx2
        for x1 = lowx1 to highx1 - 1
            swap matrix[x1,y1] with matrix[x2,y2]
            x2 = x2 + 1
        y2 = y2 + 1
```

Figure 5.14: Recursive Transpose Illustration

Pass One

$$\begin{array}{|c|c|c|c|}\hline 1 & 2 & 3 & 4 \\ \hline 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ \hline 13 & 14 & 15 & 16 \\ \hline \end{array} \Longrightarrow \begin{array}{|c|c||c|c|}\hline 1 & 2 & 9 & 10 \\ \hline 5 & 6 & 13 & 14 \\ \hline\hline 3 & 4 & 11 & 12 \\ \hline 7 & 8 & 15 & 16 \\ \hline \end{array}$$

Pass Two

$$\begin{array}{|c|c||c|c|}\hline 1 & 2 & 9 & 10 \\ \hline 5 & 6 & 13 & 14 \\ \hline\hline 3 & 4 & 11 & 12 \\ \hline 7 & 8 & 15 & 16 \\ \hline \end{array} \Longrightarrow \begin{array}{|c||c||c||c|}\hline 1 & 5 & 9 & 13 \\ \hline\hline 2 & 6 & 10 & 14 \\ \hline\hline 3 & 7 & 11 & 15 \\ \hline\hline 4 & 8 & 12 & 16 \\ \hline \end{array}$$

will fit an entire recordlength by b band of submatrices, the tool completes the transpose in one pass by fully transposing each grain in memory.

5.4.2 Analysis

The only dependencies between the parallel components of the transpose tool that show in figure 5.15 are the synchronization calls. These are barrier synchronization calls that cause all the tool components to operate in step. The analysis of the algorithm is, therefore, straightforward:

$$\begin{aligned} T_{tpass} &= \frac{n}{p}(k_1 + k_2 + k_3 + k_4 + k_5 + k_6 + k_7 + k_8) + 2k_{12} \qquad (5.12) \\ T_{trans} &= (\log n - \log b)T_{tpass} + \frac{n}{pb}(bk_9 + k_{10} + bk_{11}) \end{aligned}$$

Although the outline of the tool shows little explicit interaction between tool components, the algorithm does cause interprocessor disk contention. The operations on the lines labelled k_6, k_9, and k_{11} may be directed to a remote processor. They may therefore contend with the local reads and writes of the tool component on that processor as well as the operations from any other tool components targeting that remote processor.

Our implementation makes some effort to smooth LFS response time. It includes a dispatcher that allows each tool component to service its sequential I/O buffer and information requests from other tool components and only wait for sequential I/O when buffers are empty (or full). The read at k_6 is not sequential, and since it is a random read, it is not buffered.

Provided contention among remote tool components for access to an LFS is *fair* in the formal sense (that is, the number of times a process waiting for access

Figure 5.15: Transpose Tool Outline

This is a local component of a transpose tool that transposes an n-column matrix.

```
k = columns / 2
while k $\geq b$
    for all local records do
        Read record                                                   $k_1$
        Split into stationary and moving parts with grain length $k$  $k_2$
        Write the stationary part into temp1                          $k_3$
        Write the moving part into temp2                              $k_4$
    sync()                                                            $k_{12}$
    for all records in temp1
        Read temp1                                                    $k_5$
        Read the corresponding moving part                            $k_6$
        Join parts of grain length $n$                                $k_7$
        Write the result                                              $k_8$
    k = k / 2
    sync()                                                            $k_{12}$
```

Transpose the remaining grains in memory.
A band is b *consecutive rows of the PIFS file*

```
for BandStart = 0 to Columns by (b * NumberOfLFSs)
    for i = 0 to b -1
        read a row of the band                                        $k_9$
    Transpose the grains in the band                                  $k_{10}$
    for i = 0 to $b - 1$
        write a row of the band                                       $k_{11}$
```

to an LFS can be passed by another process is bounded) k_6 can be expected to remain roughly constant with changes in p. When p increases, more processes contend for each LFS, but there are also more LFSs available. This assumes that remote accesses are randomly distributed among the LFSs, which they are not. In our implementation, when b is a multiple of p, p components will attempt to read from the same remote LFS. They are, however, quickly separated as they are serialized whenever they contend for an LFS.

The local transpose at the end of the algorithm would appear to have a serious contention problem, but we were able to reduce the contention to insignificance. The algorithm includes a loop that loads the transpose buffer and another loop that saves it back to the LFSs. If each component filled its buffer starting at entry 0, contention would be maximized because a group of *grainsize* processes would read from a common set of processors in the same order. By cycling through the buffer with each tool component directing its initial read to a different processor the transpose array is not filled in the expected order but the reads and writes are spread evenly across the local file systems. For instance, processor zero might load eight records in the order (0,1,2,3,4,5,6,7) which would access processors zero through seven in order. Processor one would load its records (1,2,3,4,5,6,7,0) which would start at processor one. This rotation keeps each processor uniformly loaded.

5.4.3 Measurement of Transtool

We tested our implementation of the transpose tool, *Transtool,* on 4096 × 4096 arrays of bits. We ran several trials at 32, 16, and 8 processors. We found a consistent super-linear speedup (see figure 5.16), but we were able to trace this to the odd properties of EFS. The basic algorithm has linear speedup as our analysis predicted.

Using our implementation, we measured values for the constants in equation 5.13 (see table 5.9), and used these to produce predicted performance figures as shown in table 5.10

Two properties of EFS give Transtool superlinear speedup: its $O(n)$ access time for random access, and the strong influence of the EFS caches on performance for very small files. EFS maintains files as linked lists. Random access is implemented as a sequential search through the linked list. EFS will accept a hint and bypass the search if the hint is correct, but without a hint, random access is an $O(n)$ [illegible]tion. Transtool uses random reads and random writes:

- k_6 is a ran[illegible]d
- k_9 is a rando[illegible]

Figure 5.16: Measured Transtool Performance

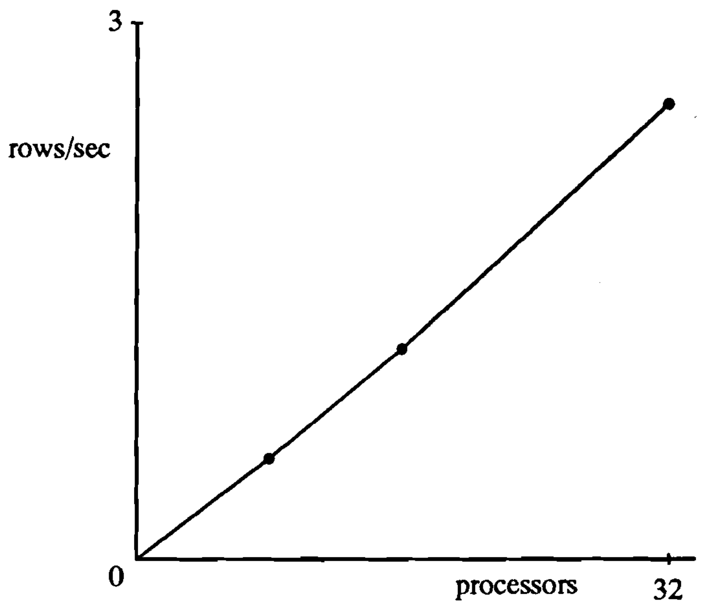


Table 5.9: Constants for Transtool

Constant	$p = 8$	$p = 16$	$p = 32$	Average
k_1	13 ms	11 ms	9 ms	11 ms
k_2	716 ms	668 ms	679 ms	688 ms
k_3	24 ms	27 ms	26 ms	26 ms
k_4	24 ms	27 ms	26 ms	26 ms
k_5	4 ms	4 ms	4 ms	4 ms
k_6	478 ms	311 ms	213 ms	$125 + .69(n/p)$ ms
k_7	707 ms	698 ms	695 ms	700 ms
k_8	8 ms	7 ms	7 ms	7 ms
k_9	31 ms	37 ms	41 ms	36 ms
k_{10}	384 ms	372 ms	383 ms	380 ms
k_{11}	167 ms	179 ms	153 ms	166 ms
k_{12}	5280 ms	19747 ms	7998 ms	11008 ms
T_{trans}	7294 s	3481 s	1605 s	

Table 5.10: Predicted vs Measured Transtool Performance

	$p = 8$	$p = 16$	$p = 32$
Measured time	7294	3481	1605
Estimate using average times	7328	3479	1793
Percent error	0.05 %	−0.05 %	11.7 %
Estimate using measured times	7262	3479	1626
Percent error	−0.4%	−0.06 %	0.1 %

- k_{11} is a random write

For k_9 and k_{11}, we were able to give EFS a useful hint most of the time. For k_6 we failed to produce good hints. The time for that read therefore depends on the size of the local file, n/p.

A 4096 × 4096 array of bits only uses 2132 blocks of file storage. When Transtool runs on 32 processors, the file only uses 66 blocks per processor. A substantial part of that file will stay in EFS's caches. This gives anomalously good performance at 32 processors. If a larger array were used the cache effect would disappear.

Our predicted performance is within a twentieth of a percent of the measured time for 8 and 16 pr sors. At 32 processors we missed by almost 12 percent, but we attribute th he EFS cache effect. Using I/O times that depend on p instead of avera s reduces the error at 32 processors to only one tenth of a percent.

5.4.4 Extrapolation from Transtool

We used the analysis and measurement of our transpose tool as the basis for some conjectures about better transpose tools. First we simply projected the performance of Transtool for up to 512 processors on a 40k × 40k-bit matrix. It was slightly superlinear (see table 5.11 and figure 5.17). As discussed in section 5.4.3, this superlinear performance is to be expected. Next we considered the possibility that Transtool's success is an artifact of EFS. We re-evaluated the Transtool's performance equation using the performance constants from table 5.1 extended to cover the transpose tool (see table 5.12). Transtool's superlinear performance was eliminated by the ordinary file system, but the new figures were still linear pa 12 processors, and the tool continued to benefit substantially from new pro past 4096 processors (see table 5.13 and figure 5.18).

From thi we conclude that Transtool's superlinear behavior is indeed an a ut the performance and scalability of a transpose tool will rema d with a typical LFS.

Table 5.11: Transtool Performance Extrapolated

Processors	Time	Rows per second
2	3,223,798 s	0.013
4	888,547 s	0.046
8	263,559 s	0.155
16	86,725 s	0.472
32	32,223 s	1.271
64	13,450 s	3.045
128	6184 s	6.624
256	3080 s	13.297
512	1661 s	24.658

Figure 5.17: Transtool Performance

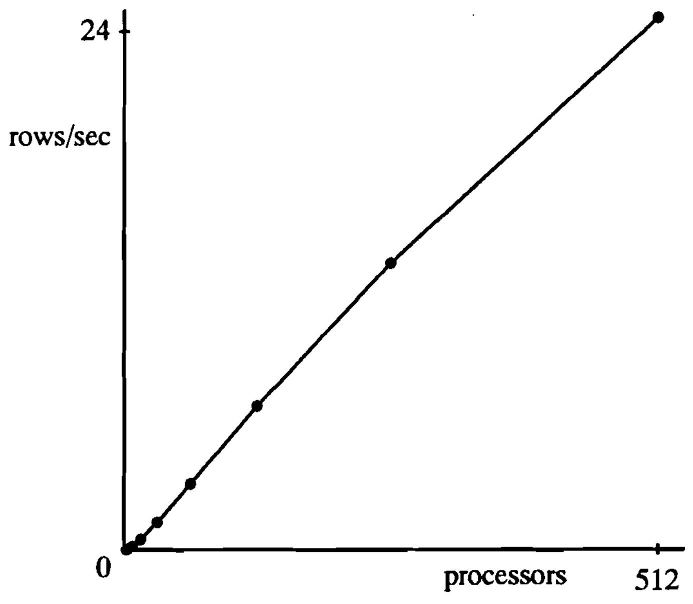


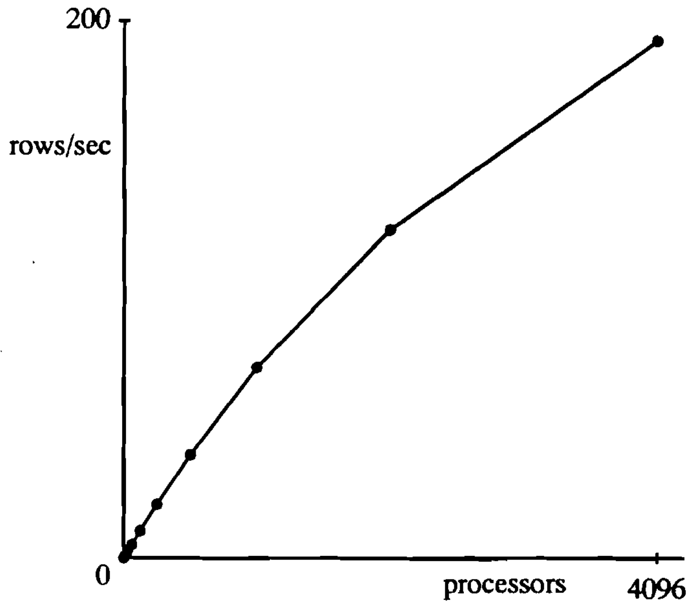
Table 5.12: Tuned Transtool Constants

k_1	3 ms
k_2	600 ms
k_3	5 ms
k_4	5 ms
k_5	3 ms
k_6	5 ms
k_7	600 ms
k_8	5 ms
k_9	5 ms
k_{10}	300 ms
k_{11}	7 ms
k_{12}	3000 ms

Table 5.13: High-Performance Transtool Extrapolated Performance

Processors	Time	Rows per Second
1	503,135 s	0.081
2	251,613 s	0.163
4	125,851 s	0.325
8	62,971 s	0.650
16	31,530 s	1.299
32	15,810 s	2.591
64	7950 s	5.152
128	4020 s	10.189
256	2055 s	19.932
512	1073 s	38.191
1024	581 s	70.468
2048	336 s	122.041
4096	213 s	192.469

Figure 5.18: High-Performance Transtool Performance



5.5 Experience with Tools

We have written and evaluated three tools. Our most important discovery is that tools are a useful vehicle for high-performance file system operations. In every case we were able to use the low-level structure of PIFS files to good advantage and get good speedup through our target range of parallelism. Each tool used a minor modification of the standard algorithm for the operation in question.

5.5.1 Design

Three aspects of the tool interface make tools easy to design:

1. Disk I/O is very slow compared to IPC and computation.
2. The PIFS structure lets any tool component use trivial arithmetic to map from local address to PIFS address and back.
3. Each LFS is independent of all the other LFSs. The only connection is the formal relationship maintained by the PIF Server.

Sorttool relies heavily on the first aspect of the tool interface. The tool has a significant non-parallel component. Sorttool's algorithm is

$$T_{sort} = O(\overbrace{(n/p)\log(n/p)}^{\text{parallel part}} + n),$$

where the first term is the parallel part of the mergesort and the second term is the non-parallel part (see equation 5.8). However, the constant for the first term includes disk reads and writes, and the constant for the second term is dominated by IPC. The tool can use a substantial number of processors before its sequential portion has any effect.

Both sorttool and transtool rely on the second aspect of the tool interface. In sorttool the addressing is implicit in the token-passing trick used to locate the next output LFS. In transtool each component of the tool freely reads and writes records throughout the PIFS file.

Every tool takes advantage of the third aspect of the tool interface. The copytool and all the other tools in that family are particularly good examples of programs that use each LFS independently. The components of tools in that class communicate only with the master process that starts all the tool components.

5.5.2 Construction

We chose to implement our tools under the Chrysalis Operating System [BBN, 1987] and in the C language. We also set rigorous performance goals (by analyzing the tools before we measured them) that required close attention to detail. Copytool was written and debugged quite quickly. The other two tools consumed man months.

Tool construction is not inherently difficult. Construction of programs that run reasonably efficiently in a real-time environment with mechanical (or simulated mechanical) components *is* inherently difficult, particularly when the program is required to meet exact performance specifications.

A good tool cannot ignore the performance difference between I/O and computation. Like the file system itself, a tool should consider optimized access patterns, caching, buffering, and asynchronous activities. One job of the file system, and by extension, of tools, is to get the best possible performance out of a disk drive that runs in parallel with the processor and has complicated timing properties. Tool construction involves considerations like:

- I can buffer up to ten read requests from various processors. How can I sequence them to prevent cache flushes? Will that reordering cause deadlock?

- How can this tool equalize I/O load with a minimum amount of synchronization?

- Why is this read taking ten times as long as it should?

The PIFS structure never hindered our tool construction. For tools the PIFS is primarily represented by a set of constraints. Though these constraints were convenient for tool design, they were largely irrelevant for implementation.

5.5.3 Analysis

Our research is focused more on systems issues than theory, but we find nonetheless that the mathematical analysis of algorithms can play an important role in this work. In the case of the merge sort tool, our analysis has proven extremely successful. Informal analysis guided our initial choice of algorithm. Detailed analysis uncovered important flaws in our implementations, and yielded trustworthy estimates of the number of nodes that could be utilized effectively. An informal analysis of our sorting algorithm suggested that it would parallelize well despite its sequential part. Further analysis confirmed that conclusion, but only a full analysis, including all constant factors, could show the range over which we could expect the algorithm to scale.

Our sorting algorithm is *not* parallel under asymptotic analysis. It is, however, simple and it is parallel over the range of parallelism that we have chosen to address. There are numerous truly parallel sorting algorithms, but they don't have the simplicity of our merge sort. Some of these parallel algorithms would use more than $(n \log n)/p$ reads; others would use more than p processors; others are too casual about access to non-local data.

Our analytical predictions accurately match the experimental results we report in this dissertation, but our first experiments with sorttool and transtool fell well below the predicted performance. For sorttool there were wide variations in read times and there was less parallelism than expected. This cast doubt on our analysis. When we included a simple model of contention for disk drives in our analysis, we obtained a much better match with the experimental data. Unfortunately, the equations became much more complex than those in section 5.3.3. We then collected more timing measurements, which confirmed that contention was a serious problem. Alerted to the problem, we were able to implement a simple read-ahead scheme that eliminated almost all of the contention, thereby improving performance and matching the predictions of the simpler version of the analysis.

The implementation of transtool was seriously short of the predicted performance until we noted that the hints we were providing for one read were incorrect. Those incorrect hints changed a read operation from a constant time operation to an operation that depended on the size of the local file. We could not supply a useful hint, so we were forced to alter the analysis to reflect the implementation.

A tool is hard to analyze for much the same reasons that it is hard to implement. I/O operations are a significant part of the running time of the tool. They cannot be ignored or easily approximated. An analysis may be seriously incorrect if it incorrectly estimates the LFS's behavior.

Optimization by exhaustive analysis is too painful for us to recommend as everyday practice, but in one instance analysis helped us uncover and repair a serious performance problem, and in another instance the comparison between predicted and measured performance located a problem we were not able to repair.

Disagreement between the analysis and our initial performance results alerted us to a performance "bug." Agreement between the analysis and our final performance results told us that the problem had been fixed. Actual identification of the problem—disk contention, or failed hints—was a hit-or-miss affair. High-quality performance monitoring and analysis tools [Fowler *et al.*, 1988] would have helped us find it sooner. Such tools are very good at identifying which portions of a program are not performing as expected. They are of limited help, however, if the programmer does not know what behavior to expect. An analysis such as ours can help by providing a model of what the program should be doing. It plays the role of a lower bound on run time—a self-sufficient benchmark against which to compare empirical results. When those results fail to match the analysis, there is either a problem with one's understanding of the algorithm (as reflected in the analysis) or with one's realization of the algorithm in code.

6 Conclusion

Our parallel interleaved file system design meets our stated goals. It is compatible with the standard file system interface, it performs and scales beyond our expectations, and it has, or can support, the features that separate a production-quality file system from a research test bed.

Our parallel interleaved file design makes the following contributions to the field of computer science.

- We have shown that a parallel file system can balance the processor power on large, general-purpose MIMD computers. In general, and within limitations imposed by IPC performance and algorithm design, the performance of a PIFS improves linearly with the parallelism of the file system.

- We introduced the concept of a file system tool. Our experimental tools and calculations based on our measurements and projections show that the performance of tools based on "obvious" algorithms scales linearly with the file system's parallelism. Based on this evidence, we state that tools are easy to design and are an effective mechanism for applications that require high-performance I/O.

- We showed that straightforwared interleaving is a practical record distribution strategy. Interleaved files have excellent performance characteristics whether they are accessed from a tool or an ordinary program, and a system management on a parallel file system based on interleaved files is easy.

- We developed an error recovery scheme tuned for a parallel file system. The design reflects the failure modes and performance characteristics of a PIFS.

- We showed that file system maintenance and administration is not significantly more difficult for a PIFS than it is for a conventional file system.

Conventional file systems and other parallel file systems are inappropriate for a large-scale MIMD computer. Uniprocessor file systems constrain the file system to the performance of a single processor node. Other parallel file systems seem inappropriate for more than about ten processors. They are a clear improvement over a conventional file system, but a parallel file system must be judged in large part by its scalability.

The Intel parallel file system [Pierce] and other parallel file systems currently under development implement parallelism at the lowest level of the file system. Disks on multiple processors are treated as a single disk and a single file system allocates sectors from the disks. File creation and deletion for this file structure is not dependent on the number of processors; it may even benefit slightly from additional processors. However, although the Intel approach works well for file creation and deletion, it gets this performance by sacrificing some read/write performance.[1] These file systems use a free list, so their allocation becomes random. This will usually behave well, but not as well as the PIFS's interleaving. Random placement makes no guarantee that consecutive records will not fall on the same processor. A tool can read a file with records placed pseudo-randomly, but it will require a map of the file. Writing such a file is difficult since the next record's location is decided by the free list, a structure which is not defined at the lowest level of the file system.

Our parallel interleaved file system is practical and scalable. It has demonstrated the ability to handle 32 I/O processors with little degradation. Analysis shows that it should effectively use well over 100 processors if it has high-speed IPC. This shows that the PIFS design may be suitable for MIMD computers with 1000 processors or more.

Even for substantially more than 100 file system processors, a PIFS imposes at most a small constant performance cost on I/O operations. Parallel file system operations overshadow the overhead with a performance improvement that may be as high as $O(p)$. Even for operations with no explicit parallelism the overhead can be outweighed by the file system's internal parallelism.

A file system tool has access to the file system at a level that permits significant optimizations that other file system interfaces do not support. The standard file system interface, our parallel-open interface, and the various parallel interfaces used by other parallel file systems all fail to give programs full access to the power of the file system. Only the tool interface reveals the locations of records to an application in a form that the application can easily use.

[1]I am not suggesting that other parallel file system implementors decided to balance I/O performance against file management performance. It is more likely that they chose the file system structure that most effectively gave them a parallel Unix-like file system.

6.1 Future Work

We have answered the most important questions about parallel interleaved file systems, but several open questions remain.

- Our choice of the Elementary File System as Bridge's LFS, and our insistence on leaving the EFS program interface intact, support our argument that a PIFS is not dependent on the idiosyncrasies of a popular file system. We achieved excellent results with the spartan set of file operations EFS has in common with typical file systems, but it would be interesting to test the performance of a PIFS with a file system designed as an LFS.

- Our original goal was to show that a 32-processor file system was practical. We now know that our goal was much too low. Analysis suggests that a machine with fast IPC can support more than 100 processors in the file system. This result should be tested experimentally.

- We know that tools are hard to write. We guess that they would be easier to write in a programming language designed specifically for parallel computers.

All these questions suggest directions for future work. Our analysis found that a PIFS would work well with 128, 512, and even 4096 processors, but our tests were constrained to 32 processors. For completeness sake our analysis should be tested with an implementation on at least 128 processors. The hardware requirements for this research are obviously extensive.

The PIFS analysis should be further checked by implementation with physical disk drives and several different LFSs.

We have ignored system throughput in favor of single-thread performance. There are a host of interesting questions relating to concurrent use of a PIFS by many unrelated processes; for instance, can the PIF Server improve performance by considering disk head position and LFS caching when it dispatches I/O requests to LFSs?

File system tools have proven difficult to write. This makes them impractical for tasks that are not I/O bound and time-critical. If tools were easier to write, they might be more generally useful. A library of functions for tool construction would be helpful, but library functions tend to be somewhat general. How much would loss of detailed control of parallelism hurt tool performance? Are explicitly parallel message-passing programming languages such as Lynx [Scott, 1987], SR [Andrews, 1982], and CSP [Hoare, 1985] suitable for tool construction?

Bridge used a slow LFS and never ran on high-performance processors. It might not contribute much to computer science research, but it would be very interesting to see how fast a **PIFS** would run a sort with 128 100 mhz 88000 processors and matching **RAID** storage.

Bibliography

[Andrews, 1982] G. R. Andrews, "The Distributed Programming Language SR—Mechanisms, Design and Implementation," *Software—Practice and Experience*, 12:719–753, 1982.

[Bashe *et al.*, 1981] C. J. Bashe, W. Buchholz, B. V. Hawkins, J. J. Ingram, and N. Rochester, "The Architecture of IBM's Early Computers," *IBM Journal of Research and Development*, 25(5):363–375, September 1981.

[BBN, 1986] "Butterfly$_{TM}$ Parallel Processor Overview," Technical Report 6149, Version 2, BBN Laboratories, June 1986.

[BBN, 1987] BBN Advanced Computers Inc., *Chrysalis Programmers Manual*, April 1987.

[Beck *et al.*, 1988] Micah Beck, Dina Bitton, and W. Kevin Wilkinson, "Sorting Large Files on a Backend Multiprocessor," *IEEE Transocations on Computers*, 37(7):769–778, July 1988.

[Bitton *et al.*, 1983] Dina Bitton, Haran Boral, David J. DeWitt, and W. Kevin Wilkinson, "Parallel Algorithms for the Execution of Relational Database Operations," *ACM Transactions on Database Systems*, 8(3):324–353, September 1983.

[Bitton *et al.*, 1984] Dina Bitton, David J. DeWitt, David K. Hsaio, and Jaishankar Menon, "A Taxonomy of Parallel Sorting," *ACM Computing Surveys*, 16:287–318, September 1984.

[Boral and DeWitt, 1983] H. Boral and D. J. DeWitt, "Database Machines: An Idea Whose Time Has Passed: A Critique of the Future of Database Machines," Technical Report 288, Technion, August 1983.

[CDC, 1986] CDC, "Hydra Parallel Transfer Disk Drive," 1986.

[CDC 88, 1988] Control Data Corporation, *Product Specification for Wren IV SCSI Model 94171*, c edition, April 1988.

[cray, 1988] "The Cray Y-MP Computer System," February 1988.

[cyberplus] "Cyberplus Software Summary".

[DeWitt *et al.*, 1986] David J. DeWitt, Robert H. Gerber, Goetz Graefe, Michael L. Heytens, Krishna B. Kumar, and M. Muralikrishna, "GAMMA: A High Performance Dataflow Database Machine," Technical Report 635, Department of Computer Sciences, University of Wisconsin – Madison, March 1986.

[DeWitt *et al.*, 1987] David J. DeWitt, Shahram Ghandeharizadeh, Donovan Schneider, Rajiv Jauhari, M. Muralikrishna, and Anoop Sharma, "A Single User Evaluation of the Gamma Database Machine," In Kitsuegawa and Tanaka, editors, *Database Machines and Knowledge Base Machines*, pages 370–386. Kluwer Academic Publishers, 1987.

[Dibble *et al.*, 1988] 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 Computing Systems*, pages 154–161, June 1988.

[Dibble, 1986] Peter C. Dibble, "Benchmark Results for Chrysalis Functions," Technical Report BPR 18, University of Rochester Computer Science Department, December 1986.

[DPT, 1989] DPT, "Distributed Processing Technology Advertisement," *Byte*, 14(6):75, June 1989.

[Ellis and Dibble, 1987] Carla Schlatter Ellis and Peter C. Dibble, "An Interleaved File System for the Butterfly," Technical Report CS-1987-4, Dept. of Computer Science, Duke University, January 1987.

[Ellis and Kotz, 1989] Carla Schlatter Ellis and David Kotz, "Prefetching in File Systems for MIMD Multiprocessors," In *Proceedings of the 1989 International Conference on Parallel Processing*, August 1989.

[Floyd, 1989] Richard Allen Floyd, *Transparency in Ditributed File Systems*, PhD thesis, University of Rochester, 1989.

[Floyd, 1986] Rick Floyd, "Short-term File Reference Patterns in a UNIX Environment," Technical Report 177, Department of Computer Science, University of Rochester, March 1986.

[Flynn and Hadimioglu, 1988] R. J. Flynn and H. Hadimioglu, "A Distributed Hypercube File System," In *The Third Conference on Hypercube Concurrent Computers and Applications*, January 1988.

[Fowler *et al.*, 1988] Robert J. Fowler, Thomas J. LeBlanc, and John M. Mellor-Crummey, "An Integrated Approach to Parallel Program Debugging and Performance Analysis on Large-Scale Multiprocessors," In *ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 163–173, May 1988, In *ACM SIGPLAN Notices*, 24(1), January 1989.

[Freeman and Perry, 1977] Donald E. Freeman and Olney R. Perry, *I/O Design: Data Management In Operating Systems*, Hayden Book Company, 1977.

[Gallager, 1978] Robert G. Gallager, "Variations on a Theme of Huffman," *IEEE Transactions on Information Theory*, IT–24(6):668–674, November 1978.

[Gamerl, 1987] Michael Gamerl, "Maturing Parallel Transfer Disk Technology Finds More Applications," *Hardcopy*, 7(2):41–48, February 1987.

[Garcia-Molina and Salem, 1988] Hector Garcia-Molina and Kenneth Salem, "The Impact of Disk Striping on Reliablility," Technical report, Princeton University Department of Computer Science, January 1988.

[Gibson *et al.*, 1989] Garth A. Gibson, Lisa Hellerstein, Richard M. Karp, Randy H. Katz, and David A. Patterson, "Failure Correction Techniques for Large Disk Arrays," In *Third Internation Conference on Architectural Support for Pragramming Languages and Operating Systems*, pages 123–132, April 1989.

[Gurwitz *et al.*, 1986] R. F. Gurwitz, M. A. Dean, and R. E. Schantz, "Programming Support in the Cronus Distributed Operating System," In *Proceedings of the Sixth International Conference on Distributed Computing Systems*, pages 486–493, May 1986.

[Hillis, 1985] W. Daniel Hillis, *The Connection Machine*, The MIT Press, 1985.

[Hoare, 1985] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.

[IBM, 1988] International Business Machines, *IBM 360/370 Principles of Operation*, 1988.

[Katz *et al.*, 1988] Randy H. Katz, John K. Ousterhout, David A. Patterson, and Michael R. Stonebraker, "A Project on High Performance I/O Subsystems," *IEEE Database Engineering Bulletin*, 11(1):40–47, March 1988.

[Knuth, 1973] Donald Knuth, *The Art of Computer Programming*, Addison Wesley, 1973.

[Kwan, 1986] Sai Choi Kwan, *External Sorting: I/O Analysis and Parallel Processing Techniques*, PhD thesis, University of Washington, January 1986.

[Lampson, 1983] Butler W. Lampson, "Hints for Computer System Design," *Operating Systems Review*, 17(5):33–48, 1983.

[Mallett and Smith, 1989] Mark Mallett and Bud Smith, "Sun's SPARCstation 1 and 3/80," *MIPS*, June 1989.

[Manuel and Barney, 1986] Tom Manuel and Clifford Barney, "The Big Drag on Computer Throughput," *Electronics*, 59:51–53, November 1986.

[Masters, 1987] Del Masters, "Improve Disk Subsystem Performance With Multiple Serial Drives In Parallel," *Computer Technology Review*, pages 76–77, Summer 1987.

[McKusick *et al.*, 1984] 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.

[Ousterhout *et al.*, 1985] J. Ousterhout, H. DaCosta, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson, "A Trace Driven Analysis of the UNIX 4.2 BSD File System," *Proceedings of 10th Symposium on Operating Systems Principles, Operating Systems Review*, 19(5):15–24, December 1985.

[Parnas and Siewiorek, 1975] D. L. Parnas and D. P. Siewiorek, "Use of the Concept of Transparency in the Design of Hierachically Structured Systems," *Communications of the ACM*, 18(7):401–408, July 1975.

[Patterson *et al.*, 1988] David A. Patterson, Garth Gibson, and Randy H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," In *Proceedings of the 1988 ACM SIGMOD Conference*, pages 109–116, June 1988.

[Pierce] Paul Pierce, "A Concurrent File System for a Highly Parallel Mass Storage Subsystem," In *The Fourth Conference on Hypercube Concurrent Computers and Applications*.

[pyramid] "Pyramid Sales Literature".

[Rinko-Gay and Varhol, 1989] William L. Rinko-Gay and Peter D. Varhol, "SCSI: The Next-Generation Disk Standard?," *MIPS*, June 1989.

[Salem and Garcia-Molina, 1984] K. Salem and H. Garcia-Molina, "Disk Striping," Technical Report 332, EECS Department, Princeton University, December 1984.

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

[Satyanarayanan, 1981] M. Satyanarayanan, "A Study of File Sizes and Functional Lifetimes," In *Proceedings of the 8th Symposium on Operating System Principles*, pages 96–108. ACM, December 1981.

[Schantz, 1984] R. Schantz, "Elementary File System," Technical Report DOS-79, BBN, April 1984.

[Scott, 1987] Michael L. Scott, "Language Support for Loosely-Coupled Distributed Programs," *IEEE Transactions on Software Engineering*, SE–13(1):88–103, January 1987.

[Siewiorek *et al.*, 1982] Daniel P. Siewiorek, C. Gordon Bell, and Allen Newell, *Computer Structures: Principles and Examples*, McGraw-Hill, 1982.

[Snyder, 1986] Lawrence Snyder, "Type Architectures, Shared Memory and the Corollary of Modest Potential," *Annual Review of Computer Science*, 1, 1986.

[Thomas and Toner, 1984] Bob Thomas and Steve Toner, "The Elementary File System," Technical Report DOS-79, Bolt Beranek and Newman, April 1984.

[TMI, 1987] "Connection Machine Model CM-2 Technical Summary," Technical Report HA87-4, Thinking Machines Inc., April 1987.

[Valduriez and Gardarin, 1984] Patrick Valduriez and Georges Gardarin, "Join and Simijoin Algorithms for a Multiprocessor Database Machine," *ACM Transactions on Database Systems*, 9(1):133–161, March 1984.

[Welch, 1984] Terry A. Welch, "A Technique for High Performance Data Compression," *IEEE Computer*, 17(6):8–19, June 1984.

[Witkowske *et al.*, 1988] A. Witkowske, K. Chandrakumar, and G. Macchio, "Concurrent I/O System for the Hypercube Multiprocessor," In *The Third Conference on Hypercube Concurrent Computers and Applications*, January 1988.