AN ABSTRACT OF THE THESIS OF

Jason A. Moore for the degree of Doctor of Philosophy in Computer Science presented on July 19, 1996.

Title: High-Performance Data-Parallel Input/Output.

Abstract approved: Redacted for Privacy

Michael J. Quinn

Existing parallel file systems are proving inadequate in two important arenas: programmability and performance. Both of these inadequacies can largely be traced to the fact that nearly all parallel file systems evolved from Unix and rely on a Unix-oriented, single-stream, block-at-a-time approach to file I/O. This one-size-fits-all approach to parallel file systems is inadequate for supporting applications running on distributed-memory parallel computers.

This research provides a migration path away from the traditional approaches to parallel I/O at two levels. At the level seen by the programmer, we show how file operations can be closely integrated with the semantics of a parallel language. Principles for this integration are illustrated in their application to C*, a virtual-processor-oriented language. The result is that traditional C file operations with familiar semantics can be used in C* where the programmer works—at the virtual processor level. To facilitate high performance within this framework, machine-independent modes are used. Modes change the performance of file operations, not their semantics, so programmers need not use ambiguous operations found in many parallel file systems. An automatic mode detection technique is presented

that saves the programmer from extra syntax and low-level file system details. This mode detection system ensures that the most commonly encountered file operations are performed using high-performance modes.

While the high-performance modes allow fast collective movement of file data, they must include optimizations for redistribution of file data, a common operation in production scientific code. This need is addressed at the file system level, where we provide enhancements to Disk-Directed I/O for redistributing file data. Two enhancements are geared to speeding fine-grained redistributions. One uses a two-phase, or indirect, approach to redistributing data among compute nodes. The other relies on I/O nodes to guide the redistribution by building packets bound for compute nodes. We model the performance of these enhancements and determine the key parameters determining when each approach should be used. Finally, we introduce the notion of collective prefetching and identify its performance benefits and implementation tradeoffs.

High-Performance Data-Parallel Input/Output

by

Jason A. Moore

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Completed July 19, 1996
Commencement June 1997

Doctor of Philosophy thesis of Jason A. Moore presented on July 19, 1996

APPROVED:

Redacted for Privacy

Major Professor, representing Computer Science

Redacted for Privacy

Chair of the Department of Computer Science

Redacted for Privacy

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for Privacy

Jason A. Moore, Author

ACKNOWLEDGEMENTS

First and foremost I must thank my major professor, Professor Mike Quinn. His insight, patience, and support have been critical in getting this work started *and* finished.

Thanks to Phil Hatcher for letting me run with his idea, and for providing great feedback on all of this research. Thanks, too, to my committee members, Dr. Bose, Dr. D'Ambrosio, Dr. Saletore, and Dr. Eleveld, for their support. Others who have helped see this research through include Mark Clement, who taught me most of my Unix knowledge (or at least showed me *how* to find it); Ken Ferschweiler and Steve Fulling, who rounded out my Unix education and provided desperately-needed system administration services; Bob Broeg, whose LaTeXskills are unequaled; and Larry Hsu, who gave me PA-RISC implementation lessons.

The United States Air Force Academy Department of Computer Science has generously supported me during my time at Oregon State. I anxiously look forward to repaying them.

I am indebted for life to Terri and the kids for putting up with my being too often at work (mentally if not physically). Most kids get to hear traditional children's songs, but Tory and Carmen had to listen to "The Buffering Song" while I mixed playtime with research. Natalie, in the womb for the stretch run, I hope you haven't been scarred for life! I love you all—you're a great family!

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

LIST OF TABLES

LIST OF TABLES (Continued)

HIGH-PERFORMANCE DATA-PARALLEL INPUT/OUTPUT

1. INTRODUCTION

1.1. Background

First-generation commercial multiple-CPU computers provided little support for parallel disk I/O, either in terms of a high-performance parallel disk system or a reasonable programming interface. Today, advances in disk arrays, coupled with the striping of data across powerful I/O nodes, provide the means for systems such as the Thinking Machines CM-5, Intel Paragon, Meiko CS-2, and IBM SP-2 to provide reasonable disk bandwidth to parallel applications.

Unfortunately, disk bandwidth is necessary, but not sufficient, to support parallel I/O operations. Existing parallel file systems are proving inadequate in two important arenas: programmability and performance. Both of these inadequacies can largely be traced to the fact that nearly all parallel file systems evolved from Unix and rely on a Unix-oriented, single-stream approach to file I/O. More researchers are agreeing that this approach is not ideal for supporting multiprocessor systems [80].

In this dissertation, these issues are addressed in the context of distributed memory parallel computers like the SP-2 and CS-2. The processors on such a machine are connected by a fast network, and parallel file access is provided by a subset of processors acting as *I/O nodes*. File data are striped across the I/O nodes,

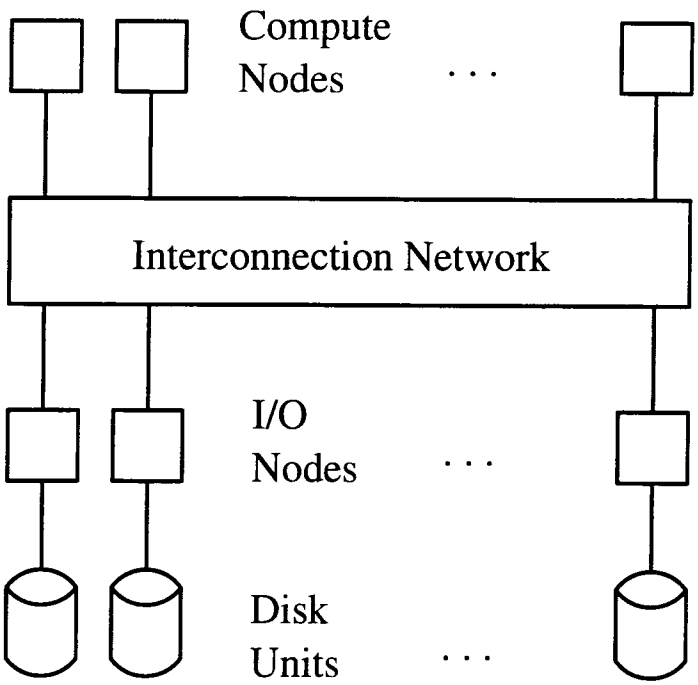


FIGURE 1.1. Organization of a distributed memory multicomputer in which I/O nodes and compute nodes share the same interconnection network.

which can communicate with each other and the remaining (compute) nodes using the parallel network. A generic example of such a system is shown in Figure 1.1.

Programming distributed-memory parallel computers, like those in Figure 1.1, has always been a notoriously difficult task; using file operations on parallel computers perpetuates this tradition. Difficulties can be found at two levels when parallel file systems are used. At the lower level, close to the operating system, the programmer uses system-level file calls comparable to the `read` and `write` calls of Unix. But, what does a `write` call issued by many processes or processors mean? Does the first `write` move a shared file pointer, subsequently impacting the next `write`, no matter which process issues it? Or, does each process have its own file pointer and operate on a file independently of others? Must all processes synchro-

nize before a file operation? Must all processes open and/or close a shared file synchronously? Will this operation work the same way on another parallel computer? Obviously, the semantics of a simple `write` is not concrete when moved from the straightforward sequential world of Unix to a parallel system.

One might be tempted to ignore the ambiguities of the low-level parallel file operations. After all, sequential C programmers tend to use C library functions like `fwrite`; not operating-system-specific functions like `write`. In a like manner, parallel programmers should use the portable file operations provided by their programming languages of choice.

At this higher level, the language level, the parallel programmer finds no relief. While many parallel languages have been developed to exploit and express parallelism in novel ways, their definitions typically ignore both the syntax and semantics of parallel I/O operations, which are essential in solving real-world problems. Therefore, the user of a parallel language is forced to use the file system of the underlying machine. This approach suffers on many fronts. First, the programmer is using a high-level parallel language to abstract away low-level architectural details. By using the system-level file operations, the abstraction is lost. Second, it may be awkward, if not difficult, to map the parallel abstraction of the chosen language to a single stream of bytes. Finally, the use of system-level file operations suffers from ambiguity and nonportability as described earlier.

One can conclude that parallel language designers ought to include I/O operations in their language definitions. Further, the syntax and semantics of I/O operations should fit the programming paradigm of the language, so the programmer does not move from one abstraction to another to perform I/O. These operations would enhance portability and abstract low-level file system details away from the programmer.

Although programmability is a major concern, parallel computers are used for speed. The programmer's I/O interface, while shielding system-level details, must not inhibit high performance. High performance can be achieved only if

1. Techniques are developed to translate high-level, programmer-friendly I/O calls into efficient, better yet optimized, native file-system calls.

2. The underlying file system moves from a myopic single-user view of file operations to a view that allows it to optimize, rather than hinder, parallel file operations.

Two inherent features of parallel file operations, the asynchrony of requests from multiple processes and distribution of data, make the latter particularly important. Out-of-order requests can turn latency-hiding techniques such as one-block read-ahead, which works well in sequential situations, into wasted reads. The data prefetch done after one compute nodes's request may be purged while other compute nodes are served, even if the prefetched data will be needed soon.

Further complications arise when files are accessed by programs using different data distributions. Such accesses account for a majority of production scientific programs' file accesses [56]. When a file written by a program using one data distribution is read by a program using another, the data must be permuted. The brute-force method of permuting, namely having each compute node read many small pieces of the file making up its local data, inherently requires many inefficient fine-grained file-system calls. More elegant approaches, which rely on the notion of *collective I/O operations*, have been proposed to speed up the permutation operation.

Collective I/O operations require that all compute nodes synchronize before the file system is contacted. The file operation to take place, typically transfer of a

logically contiguous file segment, is then viewed from a global point of view rather than from the local view of the individual processors and the portions of the file segment they must access. A file-system-independent approach to collective I/O is the two-phase access strategy [27], in which data are redistributed among compute nodes after a read or before a write. Each compute node then accesses a large, contiguous portion of the file.

Although two-phase I/O is a step in the right direction, the underlying file system must be aware of parallel optimizations like collective operations and actively support them. To quote Marc Snir of IBM, "We should stop thinking of I/O as a communication between a job and file. I/O is really communication between one job and another job" [80]. The I/O "job" referred to here must extract maximum performance from its I/O devices and pass that performance to the compute nodes by exploiting high-level knowledge (e.g., compute node data distribution) about file-system operations. Such knowledge can be inferred or easily described when operations are regular. Many studies of the I/O requirements of applications running on high-performance computers have found that their file operations are regular, array-oriented, and sequential, distributions notwithstanding [22, 36, 56, 63, 64, 77].

Disk-directed I/O [52] is the prototypical example of the "file as a job" approach to I/O. The file system running on I/O nodes optimizes disk accesses and, when necessary, permutes file data as it is transmitted to compute nodes. Although it performs well in many instances, disk-directed I/O suffers from high message-passing overhead and a lack of prefetching support. It does, however, provide a foundation upon which an efficient, proactive parallel file system can be designed.

This section closes with a merging of the earlier discussion of high-level languages with the notion of a knowledgeable, proactive, collective, parallel file system.

Clearly the run-time system for a parallel language must take advantage of advanced features of a parallel file system. In fact, as file systems becomes more complex, it becomes critical that implementation details are hidden from the programmer but exploited by the run-time system. The programmer then has portable, abstract access to I/O in which collective operations can be optimized on a large scale, prefetching takes into account the "big picture" of file operations, and expensive, fine-grained file system operations are avoided.

1.2. Research Contributions

This research addresses shortcomings of existing parallel I/O facilities at both levels described in the previous section. At the language level, design principles are presented for integrating I/O into parallel languages. The use of the principles is illustrated in their application to C*, a virtual-processor-oriented language. The limitations of the single-stream view of files for a language like C* are detailed. The use of *machine-independent modes* to support both high performance and generality are motivated. Modes change the performance of file operations, not their semantics, so programmers need not use ambiguous operations. An automatic mode detection technique is presented that saves the programmer from extra syntax and low-level file system details. This mode detection system ensures that the most commonly encountered file operations are performed using high-performance modes. The high-performance modes have been crafted to take advantage of collective file operations and related optimizations. Finally, virtual processor file operations, typically fine-grained by themselves, are combined into efficient large-scale file system calls.

Although these principles are demonstrated in the context of C*, their significance is farther reaching. The notion of language-specific, rather than machine-

specific, modes addresses the tension between the need for speed and the need for generality in file operations. File operation and mode design take into account a language's style of computation, its constraints, and common operations. Therefore, I/O fits the language's parallel abstraction. A parallel language must have this kind of support for I/O in order to be taken seriously and used for production programs. As long as I/O within parallel languages is ignored, the parallel software crisis will linger.

At the lower (file system) level, two major weaknesses of disk-directed I/O are addressed here. First, its performance suffers due to message-passing overhead when small records must be redistributed, despite the optimistic machine parameters given in [52]. This research shows that this problem cannot be solved simply by having I/O nodes use scatter and gather to send fewer messages, especially as disk speeds increase at their present rate. An alternative approach to redistributions of small records is provided in the context of disk-directed I/O, in which the two-phase access strategy is utilized. A model and empirical results are presented showing that this combination of disk-directed I/O and the two-phase access strategy is faster than traditional disk-directed I/O and disk-directed I/O with scatter and gather.

The second weakness of disk-directed I/O is its failure to perform any prefetching. Because it relies on collective I/O operations to get global information, the file system is in a strong position to aggressively prefetch for subsequent collective operations. Redistribution of data from I/O nodes to compute nodes is a major concern. Hence, pre-permuting prefetched data must be examined as an option to increase effective bandwidth. An open question is how properly to pre-permute data that have been prefetched. This issue is explored and several alternatives are modeled.

To support the models used in the discussion of disk-directed I/O, this research develops a detailed model of redistribution cost. In particular, the model focuses on the most expensive part of the redistribution, packet-building and unpacking costs. These are modeled independently of the parallel message-passing system in place. In the model, the impacts of relevant architectural features such as Translation Lookaside Buffer size and replacement policy, cache size, and compute node count are described qualitatively and mathematically. The model is accurate for a wide variety of redistributions and validated on three different architectures.

Not only does the redistribution model presented here provide a foundation for our work in permuting data for parallel I/O operations, it has significance across parallel computing. Redistributions are required frequently in languages like HPF, in which library subroutines are optimized for only a limited number of distributions, or when differently distributed arrays are combined in an operation. Because of the frequency of redistributions in this context, modeling their cost is critical. Data distributions also play a key role in the design of parallel programs which are composed of the composition of several tasks. Each of these tasks may be performed by several different algorithms, each optimized for a different data distribution. If a program designer, human or automated, wants to predict the optimal sequence of algorithms for a program, the time taken for redistributions between algorithms must be accounted for. This research is the first to examine, in detail, the parameters most directly influencing redistribution cost. From it, accurate models valid for a wide variety of machines can be built.

1.3. Organization of the Dissertation

In Chapter 2 previous work in parallel file systems, their use, and data distributions is discussed. A description of virtual processor streams within C* is presented in Chapter 3. Chapter 4 contains the detailed data redistribution model, whose results are then applied in Chapters 5 and 6 in evaluating alternatives to traditional Disk-Directed I/O for permuting data and prefetching. In Chapter 7 the results are summarized and future directions for this research are outlined.

2. RELATED WORK

Like parallel I/O systems on real machines, parallel I/O research is still maturing. Only within the last several years have the shortcomings of current parallel file systems sparked significant interest in the research community. This chapter briefly describes important contributions to the field related to this work and, when appropriate, differentiates this work from them. These contributions are broadly divided into the following areas: improving the programmer's interface, studying I/O needs of "real" parallel applications, improving redistribution of file data, and enhancing parallel prefetching. In addition, important work in the related area of array redistribution is discussed in this chapter.

2.1. Improving the Programmer's Interface

Until recently, virtually all parallel file systems proposed and implemented have given the programmer a view of a parallel file in which all processors (or virtual processors) access a single byte stream. These include the CM-5's sfs [6] [62], which provides an inflexible, single-stream approach for C* virtual processor I/O [94], Intel's CFS/PFS [2] [7] [45] [76], Bridge [29], the nCUBE-2 file system [23] [24], MasPar parallel I/O [68], and more [9] [28] [32] [100]. The programming interfaces for these systems, all vendor-specific, are typical Unix-like `read`, `write`, and `seek` environments, some with different *modes* for accessing files in parallel.

Modes give the programmer flexibility in selecting the type of processor interaction and file pointer management performed during file operations. Common access modes include *shared-pointer, shared-data* mode, in which all processors write

or read the same value; *synchronous* mode, in which a file operation consists of all processors synchronizing, then transferring data sequentially in processor number order; and *independent* mode, in which each processor has its own file pointer and can access the parallel file independently of other processors. The flexibility provided by modes is not necessarily a good thing; modes complicate the programmer's task with non-portable, low-level details. A major drawback of modes is that they render file operations ambiguous. That is, the same operation has different meanings depending on the mode selected. Hence, the use of modes hurts readability and maintainability of parallel programs relying on them.

The modes in Stream*, presented in Chapter 3, differ from the modes offered by these file systems. Stream* modes are machine-independent. Mode changes do not change the semantics of file operations. Stream* modes are managed by the run-time system, and the programmer need not be concerned with the underlying implementation.

As an alternative to the single byte stream approach to parallel file systems, Kotz proposes the idea of multifiles [51]. A multifile contains several subfiles, typically one per parallel process. Multifiles simplify the programmer's job, because processes neither worry about synchronizing for file operations nor positioning their data in a shared stream. PIOUS [67] provides multifiles as one option for implementing a parallel file. An obvious drawback to this approach is the dependence of the number of subfiles on the number of compute nodes used during file creation—what if more or fewer compute nodes are used to access the same file later? The approach proposed by Hatcher [41], which is the foundation for Stream* presented in Chapter 3, is a generalization of the notion of multifiles. With this approach, the number of Stream* subfiles, or data-parallel streams, is tied to the number of virtual processors. This number remains the same regardless of the number of physical processors

employed. Also, a generic implementation of multifiles does not support Stream* optimizations like the collection of many fine-grained writes into one contiguous file write. Therefore, using multifiles for data-parallel, virtual-processor-oriented streams would slow the file system to a crawl.

Other file systems provide different views of a parallel file. Vesta [13] [14] [17], a research project now implemented as IBM's PIOFS, allows a file to be opened using several views. These views are based on a structured, two-dimensional-array layout of data. That is, a stream of bytes is divided into a repeating pattern of bytes. Each of these repeating patterns contains records divided among the processors along one or two dimensions in a pattern such as Block or Cyclic. Each process' file operations access only those records mapped to it; the mapping is transparent to the programmer once the file has been opened and a specific layout specified. Although the view is that each process has its own stream, Vesta assumes all streams are the same size, and that all processes read or write the same amount of data in the same fashion—only one EOF marker is maintained for the file. In terms of abstraction, Vesta's file interface is simpler than that of traditional file systems at the expense of a complicated mapping specification when a file is opened.

MPI-IO [15] [16] takes a similar approach to Vesta, but MPI-IO's design is much more general. At the lowest level, a file is a sequential stream of bytes. Each process specifies a `filetype`, which is essentially a template that repeats itself on the stream of bytes. Within a process' template are data and holes; the data are readable and/or writeable, while the holes, presumably used by other processes, are ignored. MPI-IO is more general than Vesta, because each process' filetype is independent of others. Data accessed by different processes may partially or completely overlap. The generality of MPI-IO comes at the cost of a steep learning curve. The data layout specification is complex, and the number of available file

operations is overwhelming. Any gains from the parallel abstraction are lost in the details.

A system providing a high-level file abstraction is the Transparent Parallel I/O Environment (TPIE) [95]. TPIE is a set of C++ templates and libraries, where the user provides callback functions to TPIE access methods. TPIE has built-in stream handlers which perform scan, merge, distribution, sort, permute, batch filter, and distribution-sweep. The programmer-provided callback functions are applied to streams as streams are manipulated by the stream handlers. Parallelism is exploited with the distribution function, which divides data from a sequential stream among many processes. TPIE relies on a SPMD, or single thread of control, style of parallelism. The programmer consistently interfaces at a high level with stream operations. Unfortunately, whether or not the abstract interface scales efficiently to parallel machine is unknown, as the current implementation is for a uniprocessor with multiple disks.

2.2. I/O Needs of Parallel Applications

The optimizations we present in Chapters 3 and 5 are geared toward data-parallel applications using predictable, regular file accesses. Several recent studies show that file usage in the scientific parallel programming environment is predictable. A study of four parallel oceanographic applications [64] found that all file accesses were array-oriented and sequential. Scientific applications at Argonne National Laboratories [36], NASA Ames [63], and the San Diego Supercomputer Center [71] [72] accessed files in a similar manner. Six scientific codes studied by Cypher *et al* [22] running on an Intel Touchstone Delta, nCUBE-1, and nCUBE-2 perform sequential, matrix-oriented file operations. The NHT-1 Application I/O

Benchmark [31] emphasizes sequential array writes, while the file I/O used by the Perfect Benchmarks [79] is predominantly sequential.

A study by Crandall *et al* [20] finds that three scientific applications running on the Intel Paragon use less regular file accesses. However, the file usage of these applications is dictated by limitations of the underlying file system. The programmers take direct control of data placement on disk rather than relying on modes provided by the Paragon's PFS [45].

Traces of production parallel code running on NASA Ames's Numerical Aerodynamics Simulation (NAS) facility iPSC/860 [56] and the National Center for Supercomputing Applications' CM-5 [77] give a different characterization of file usage. Instead of large, sequential, array-oriented file accesses, many programs accessed files in small pieces. Further analysis of these accesses, however, shows that the fine-grained accesses occur with regular strides and nested strides. The application programs are manipulating arrays, but the data are laid out in the file with a different distribution than the desired distribution among compute nodes. The data cannot simply be transferred to each compute node as a contiguous block from the I/O nodes.

2.3. Redistribution of File Data

Several approaches have been presented to avoid the many fine-grained file operations often needed to permute data from the compute node distribution to the file distribution. The simplest of these by Nieuwejaar and Kotz [69] supports the use of *strided* and *nested-strided* file operations. With this approach, the runtime system can request an entire file block and sieve out unwanted data rather than requesting a portion of the same block once for each record residing in it.

This approach has the drawback that its interface, consisting of `strided_read` and `strided_write` calls, is somewhat low level.

Most other approaches to file data redistribution rely on *collective I/O*, in which all processors synchronize and make a single large request for data. This large request encapsulates both the entire file segment accessed and the associated redistribution. The run-time system can then optimize the file operation as a whole instead of performing many seemingly independent fine-grained operations.

Collective I/O has been implemented in several systems, including CM Fortran and C* [94] on the CM-5, and the nCUBE/2 [23] [24]. SPIFFI [33] is a research file system supporting collective I/O operations. Parallel I/O libraries allowing collective operations include VIP-FS [28], MPI-IO [16], the Syracuse HPF interface [8], Jovian [5], PETSc/Chameleon [36], nCUBE [23] [24], and Panda [84]. Not all of these systems address collective operations in the context of file data redistributions; those that do are described below.

The nCUBE/2 [25] supports redistributions using two mapping functions. Unfortunately, these functions only work when the number of array bytes, the number of compute nodes, and the number of I/O nodes are all powers of two. The first mapping specifies the data distribution on compute nodes, and the second specifies the distribution of the file among I/O nodes. The composition of these mappings is used to calculate the compute node and array offset for each byte in the file. The composite mapping also gives the number of consecutive bytes to transfer for a given compute node and array offset. Since the I/O nodes are given the mapping functions, this approach eliminates the overhead of many data requests. Details of how data are actually transmitted (one message for each set of bytes contiguous on both compute nodes and disk, or using scatter and gather) are not given.

A collective, file-system-independent approach to file data redistribution is the two-phase access strategy [8] [27]. Using this approach, the needed file segment is accessed in the distribution with which it is stored in the file (e.g., row-major order). The result is that only a few large-grained file operations are needed to request and transfer the data. The compute nodes then redistribute the data among themselves before a write or after a read. While a data redistribution can be a fairly expensive operation, it is much cheaper than many fine-grained file operations otherwise required to permute the data.

VIP-FS [28] supports both mapped file operations, like the nCUBE/2 but without power-of-two array size limitations, and the two-phase strategy with its collective operations. In addition, it uses the notion of *assumed requests* to minimize the number of file requests made on a slow network of workstations.

Jovian [5], while eventually planning on providing a global view of file operations, relies on compute nodes' local view to specify file operations. The interface consists of file operations for ranges of data and strided data. All file operations are collective, and a group of *coalescing processes* act as intermediaries between the application program and the file system. Each application process makes requests to the single coalescing process assigned to it. After requests have been passed to the coalescing processes, they sort and exchange requests so each coalescing process has all requests for a given I/O node (that is, there is exactly one coalescing process acting on behalf of each I/O node). Because requests are sorted, a single file block is requested at most once during an operation. Hence, fine-grained file requests are not eliminated (because they are communicated to coalescing processes), but they do not result in fine-grained disk requests.

Disk-directed I/O [52] relies on a collective interface. Compute nodes synchronize before requesting a file operation, and the I/O nodes, which are informed of

the data distribution, supervise the transfer of file data. The transfer is performed using a double-buffering scheme, in which one buffer is used for the current disk operation, and the other is used to send data to or collect data from the compute nodes. When operations on both buffers are complete, their roles are reversed. The I/O nodes sort the list of disk sectors accessed to ensure that the disk, typically the bottleneck, is used as efficiently as possible. The permutation of data from the disk distribution to the compute node distribution is done by having the I/O nodes requesting specific data from compute nodes for writes or sending data to compute nodes for reads. Because the I/O nodes sort the sectors before accessing them, the I/O nodes must have control over the order in which data are transferred between compute nodes and I/O nodes. Simulations of this scheme show that all but the finest-grained redistributions take essentially the time taken to access the disk. That is, the redistribution completely overlaps the disk access. The machine assumptions used for the simulations in [52] are somewhat optimistic, in that the message-passing latency is approximately 2 μs and bandwidth is 600 MB/s, and the implementation, due to its machine- and disk-specific nature, is nonportable. However, the potential performance of disk-directed I/O makes it a good starting point for a parallel file system.

A more abstract version of disk-directed I/O, one not relying on direct control of disks, is implemented in Panda [83]. Essentially, intermediate processes supplant the machine- and disk-specific operations performed by the disk-directed I/O nodes as defined in [52]. These intermediate processors direct the redistribution of data. Performance results show that nearly peak disk bandwidth is achieved, but challenging redistributions such as Block to Cyclic are not benchmarked.

On a related, more theoretical note, several researchers have explored the problem of reorganizing data on a striped file system. That is, the data are read

from one distribution and written in another. The class of redistributions for which this body of work applies can be described using bit-matrix-multiply/complement (BMMC) operations. Cormen [18] [19] analyzes BMMC redistributions with a goal of minimizing the number of disk accesses. More recent work by Wisniewski [99] analyzes *in place* BMMC redistributions of large files on a striped file system.

2.4. Array Redistribution

The previous section described recent schemes for permuting parallel file system data for use on compute nodes. Although our central concern is redistribution in the context of I/O, the results presented here are also applicable in the more mainstream context of array redistribution. Redistribution of data is important for many parallel applications. In many instances, data must be distributed one way to minimize communication in one portion of a program, and it must be distributed differently to minimize communication elsewhere. Although a redistribution may be expensive, its use may result in significant time savings by allowing the use of optimal algorithms for different steps in a program. In some systems, library functions require that input data have a particular distribution; in such cases a redistribution is required.

Johnsson and Ho [46] performed early work in data redistributions. Their approach is limited to power-of-two-sized arrays, one element per processor, on hypercubes. They do not address more general redistributions like those in HPF.

Thakur and Choudhary [90] [91] propose straightforward, cache-efficient methods for redistributing data. Their methods rely on direct communication between all processors rather than the forwarding of data through intermediate processors to reduce the impact of latency. They also provide a performance prediction

model for all-to-all communications [89] associated with such a redistribution. Ramaswamy and Banerjee [78] use PITFALLS, a notation for expressing intersections of array segments, to calculate the send and receive sets for redistributions. PITFALLS has more general applicability than the algorithms of Thakur and Choudhary, and the results in [78] show that PITFALLS is faster for more general redistributions. The authors make a reasonable case that PITFALLS is also faster for multi-dimensional array redistributions.

Kalns and Ni [48] propose a scheme in which processors are logically reordered during a redistribution to minimize the communication needed for the redistribution. They also have a set of library routines called DaReL [47] implemented in MPI to provide portable redistributions. In a similar vein, Kunchithapadam and Miller [59] attempt to optimally distribute data so that the redistribution has minimal cost. Their scheme requires an instrumented run that tells which data each processor sends to others. An offline graph coloring scheme is then used to place data on processors to minimize communication for subsequent runs. Another scheme for placing data in advance to minimize the cost of a redistribution is the *spiral mapping strategy* of Wakatani and Wolfe [98], in which each processor sends to and receives from the same set of processors, all of which are logically neighbors.

A group at Ohio State University has developed a virtual processor approach to general cyclic to cyclic redistributions [39] and a multiphase approach to redistribution [50]. These schemes rely on a tensor product algebraic notation [49] for expressing redistributions and evaluating their cost. The redistributions appear to be the same as those proposed by Thakur and Choudhary, but the complex algebraic notation limits their usefulness.

Wakatani and Wolfe [97] propose a stripmining scheme in which communication for redistribution of an array segment is overlapped with computation on an-

other array segment. As Chapter 4 of this thesis shows though, a compute-intensive expense of redistributions is building of packets, which cannot be overlapped with computation.

Lee *et al* [61] present communication-efficient algorithms minimizing disk operations for redistribution of out-of-core data. The redistribution follows three phases much like those of Thakur and Choudhary [91]. The phases are packet-building, communication, and unpacking. The out-of-core approach simply requires that all these phases take place in steps, each step working with slabs of the data that can fit in core.

2.5. Prefetching of Parallel File Data

Prefetching has been studied extensively for uniprocessor systems [86]. Unix implementations typically perform one block lookahead when reading files [3], while disk controllers fill their buffers with requested read data as well as subsequent sectors [57] [81]. Because of the striped nature of parallel file systems, and because processors can request file data in unpredictable sequences, traditional sequential prefetching methods cannot be naively migrated to parallel systems [52].

Kotz and Ellis performed early work on prefetching for parallel systems [54] [55]. Their studies concentrate on a synthetic workload consisting of eight different file access patterns. The prefetching mechanism chooses between one block lookahead, infinite block lookahead, or portion (access pattern) recognition. These studies were done before the file needs of parallel applications became reasonably well understood, so their conclusions, based on a synthetic workload, are of limited value.

Existing parallel file systems provide either a simple prefetching scheme used at all times, or they rely on prefetching hints from the programmer. In the former category are Intel's CFS and PFS [34] [70], which perform one block lookahead for read prefetching. A PFS substitute for the Paragon [1] generates asynchronous one-block lookahead requests after each user request. This style of prefetching is similar to that for the original PFS, and its success is mixed for a variety of file operations. SPIFFI [33] performs implicit prefetching by increasing the block size so out-of-order requests for data result in much surrounding data on either side automatically being pulled into the buffer cache. The latency of the first access of the block is not hidden by any prefetch operations.

More sophisticated systems let the programmer provide input on the prefetching done by I/O nodes. Vesta [14] provides explicit prefetch operations at the user interface. PPFS [44] advocates user control over prefetching, but the current interface only provides support for enabling and disabling prefetching. The Hurricane File System [58] lets the user specify sequential, bounded sequential, or general prefetching. *Transparent informed prefetching* (TIP) [37] [73] [74] [75], now built into the Scotch file system [38], lets the programmer or the system generate hints for prefetching. The PASSION project provides `PASSION_prefetch_read()` and `PASSION_prefetch_wait()` calls to support asynchronous requests for data [12] [88]. The ADOPT dynamic scheme [85] lets the user specify either a sequence of blocks that will be needed in order or a set of blocks that will be accessed together in parallel. MPI-IO lets the programmer specify *hints*, and these can be applied, depending on the implementation, to prefetching. Vitter [96] proposes a novel probabilistic prefetch scheme based on the Ziv-Lempel data compression algorithm.

None of the above schemes has been combined with a collective interface to support prefetching of data relative to the global view of the file access. A collective

interface prevents the I/O nodes' prefetching mechanism from being confused when seemingly random, fine-grained file accesses are performed. With a collective view, even if many small file operations are requested, I/O nodes can hold data they know will be requested during the operation. When the collective view is combined with redistribution information, prefetching can be more than simply reading the disk in advance of file operations. As Chapter 6 shows, progress on the redistribution can be made as well, resulting in significant gains in completion of a read with redistribution.

3. EFFICIENT DATA-PARALLEL FILES VIA AUTOMATIC MODE DETECTION [1]

Parallel computing offers great potential for speeding up a wide variety of applications. Unfortunately, parallel software has not matured enough to support mainstream developers and users. They will not embrace parallelism until the tedious low-level details of parallel programming are abstracted away from the programmer and high-level languages and environments support the construction of portable, high-performance applications. To this end, many parallel languages have been proposed. Few of these languages have been adopted by applications programmers.

One significant reason parallel languages have not been accepted is their lack of support for parallel I/O, which is critical for real applications. Parallel language designers must incorporate into their languages I/O operations which support high-level I/O integrated with the language's style of computation, which can be implemented with high performance, and which are portable. Although many portable parallel file systems have been proposed in recent years, their interfaces

---

[1]Much of the material in this chapter appears in the Proceedings of the Fourth Annual Workshop on I/O in Parallel and Distributed Systems and is Copyright ©1996 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that new copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request Permissions from Publications Dept, ACM Inc., Fax +1 (212) 869-0481, or `<permissions@acm.org>`. Used with permission.

are relatively low-level, and their details require significant programmer effort to master. A more abstract interface, perhaps built on top of one of these systems and utilizing its optimizations, is needed.

In this chapter, we describe a high-level and intuitive file interface for virtual-processor-oriented languages. The interface is presented in the context of the SIMD language C* [92]. The C* user's view is of an abstract parallel machine consisting of a front-end scalar processor combined with a back-end collection of virtual processors (VPs). The number of VPs matches the data-parallelism of a given program; each VP maintains its own element of every parallel variable. Parallel operations are programmed from the viewpoint of what each VP does with its data. Using the VPs' view makes parallel programming easier than less abstract methods [42]. We apply the same view to parallel files: a parallel file consists of one stream per VP, and each VP operates on its own stream within a file, as shown in Figure 3.1. Therefore, a parallel file may contain millions of streams, each under the control of a different VP. We call our implementation of this abstraction Stream* ("Stream-star"). In addition to maintaining consistency of the C* programmer's view, Stream* enhances programmability through its interface, which consists of parallel versions of familiar C file operations. At the VP level, these operations have the same semantics as their sequential counterparts.

The use of C file operations as the Stream* interface provides the programmer with great flexibility. Therefore, our implementation must support general file operations. However, parallel computers are used for speed, and support for general operations must not hamper the performance of frequently occurring, structured operations. Stream* addresses this dichotomy by accessing parallel files in three *modes* which use different techniques for managing VP file data. Two modes support simple, regular I/O operations and have little overhead, while a third mode

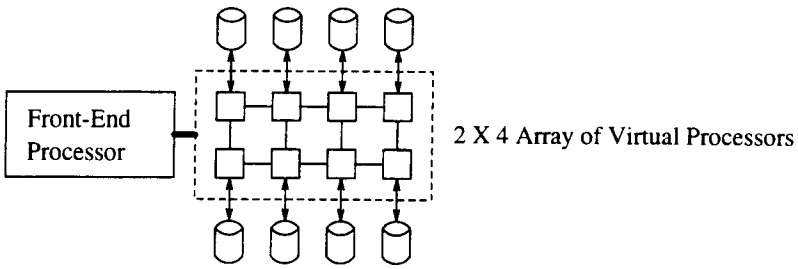


FIGURE 3.1. The C* model of computation combined with the data-parallel file abstraction in which each virtual processor in the SIMD machine's back end has its own I/O stream.

supports complex, irregular I/O operations. As an added advantage to the programmer, Stream* automatically selects the most efficient file mode to use at run time.

The selection of modes and the Stream* design were guided by the following principles, which can be applied to languages other than C* as well.

- Automatic mode detection by the run-time system maintains a high-level interface—the programmer is not burdened with specification of modes.
- Modes depend only on the program, not the underlying file system, so their management is completely portable.
- Modes change the performance of file operations, not their semantics.
- Common operations are performed using high-performance modes.
- High-performance modes are designed so they can exploit redistribution optimizations such as disk-directed I/O [52] and the two-phase access strategy [27], where available.

- Whenever possible, fine-grained VP file operations are combined into large-grained file system operations.
- The system must be able to read and write external files (e.g., from sequential programs or other data-parallel programs).

The remainder of the chapter describes how we have applied these principles to the implementation of Stream*. We first discuss related work and follow that with a brief discussion of C*. We then examine the choices available for C* I/O paradigms. Next is a general discussion of Stream* modes, followed by detailed descriptions of their implementation. Throughout this section, we illustrate design tradeoffs by presenting results from C* programs compiled and run using Stream*. We finish by describing how Stream* interfaces with external programs and how the C* programmer can, without knowing implementation details, ensure high performance modes are used.

3.1. Related Work

The first C* file system was built by Thinking Machines. Its interface, similar to that for VP-oriented CM Fortran on both the CM-200 and CM-5, includes limited functionality in which all VPs transfer data to or from a single stream [93, 94]. Virtual processor streams for C* were proposed by Hatcher [41], whose results with a general implementation are reported in [4]. Moore *et al* [65] point out the shortcomings of the single-stream approach to VP files and suggest the use of high-performance modes for parallel streams. Here we extend this work to include automatic mode detection, design details and tradeoffs, and optimizations.

Most proposed parallel file systems for MIMD machines give the programmer one view of a file, namely as a single stream of bytes. Some systems, including Vesta

[14], PIOUS [67] and others [51] support "multifiles", in which a parallel file is broken into multiple subfiles or segments, typically one per physical processor. The CM-200 supports *parallel files*, in which each physical processor accesses its own subfile [93]. The notion of parallel VP streams is a large-scale generalization of this idea, which can simplify I/O programming significantly. Unfortunately, one cannot simply implement VP streams as segments or subfiles on top of these systems. PIOUS would require the opening of many thousands of files, and array-oriented Vesta assumes all subfiles are accessed in the same manner, so it does not store EOF for each subfile. The parallel files of the CM-200 are too closely tied to the physical machine size to manage streams for a larger virtual machine. Further, if each VP explicitly accesses its own subfile, many fine-grained file operations are required. Performance suffers significantly when this approach is used.

Modes were introduced with the earliest parallel file systems [76, 100], and some more recent systems use modes [6, 45]. The systems provide a limited number of operations whose synchronization requirements and file access semantics vary depending on the current mode. The implementations of modes on these systems have several drawbacks. They are machine-dependent. They give a single file operation multiple meanings; this leads to poor code readability. Finally, modes complicate the parallel programmer's task with low-level details. In contrast, Stream* modes are machine-independent and affect only the performance of file operations, not their semantics within C*.

3.2. C*

3.2.1. Programmer's Model

The C* data-parallel language [92] is targeted for a logical SIMD machine consisting of a front-end sequential processor and a back-end processor array. C* programs contain scalar variables and parallel variables. Scalar variables reside on the front-end processor. Parallel variables are associated with a *shape*, whose left indices define the layout of virtual processors along any number of dimensions. The declarations in Figure 3.2(a) illustrate several language features. The shape `S` is logically laid out as a 256 $\times$ 128 array of VPs. Stored at each VP is a single integer `x` and an array `y` containing 10 doubles. The variables `i`, `sum`, and `scalarFile` are scalars. The pointer `parFile` is a scalar which will point to a parallel file variable when it is allocated upon opening of a file.

Sequential, or scalar, code is logically executed by the front-end processor. Parallel code, contained in the `with` statement, is executed by the back-end array of VPs. When not all VPs need to work, some can be *masked off* using the `where` statement; the division of VPs into active and inactive subsets forms the *context* of operations in the `where` statement. The sample code segment in Figure 3.2(b) demonstrates these features.

Finally, parallel data can be manipulated using two different types of functions. C* functions can be passed parallel arguments and can return a parallel variable. The context is passed to these functions when they are called, and synchronous SIMD operation is maintained while the functions are executed. The computation performed depends on the context when the function is called. The following examples include a prototype for function `foo` and a call to `foo` from within a `where` statement.

```
shape [256][128]S;

int:S      x;
double:S   y[10];
FILE:S    *parFile;
int        i;
double     sum;
FILE      *scalarFile;

.
.
.

sum = 0;                          /* Scalar code */
for (i = 0; i < 10; i++) {        /* Also scalar */
  with (S) {                      /* Perform in parallel */
    x += y[i];
    where (x > 100) {             /* Only some VPs will do this */
      x = x % 100;
    }
  }
}
```

FIGURE 3.2. Sample C* declarations followed by a code segment containing both scalar and parallel C* code.

```
double:S foo(double:S a, double:S b);

int main() {
...
  with (S) {
    where (x > 0) {
      y[2] = foo(y[0],y[1]);
    }
  }
...
}
```

The second type of function with which parallel data can be manipulated is an *elemental function.* Elemental functions are an important extension to C* formulated by ANSI X3J11.1 [40]. Elemental functions allow VPs to invoke scalar functions conceptually by passing individual elements of parallel variables as arguments. For example, the University of New Hampshire (UNH) C* compiler, with which this work is integrated, provides parallel access to the standard C library routine `sin` by providing it as an elemental function. The subtle difference between standard C* functions and elemental functions is that standard functions maintain SIMD-style synchronization, while elemental functions do not. The current context is passed to standard functions, which may increase the number of active VPs using the `everywhere` statement. With an elemental function, the current context determines whether or not individual VPs *enter* the function. Users can write their own elemental functions, but some operations must be avoided in elemental functions. In particular, an elemental function must operate only on its parameters, which must belong to the calling VP. An elemental function cannot have side effects on scalar data or the data of other VPs. A "single, shared byte stream" approach to C* I/O cannot support file operations in elemental functions, because the shared

file is a scalar. Since each VP has its own stream in Stream*, file operations within elemental functions can be supported.

3.2.2. C* Implementation on MIMD Machines

Although C* provides the programmer with a simple SIMD view of computation, the UNH C* compiler has efficient implementations for a variety of MIMD machines. We briefly describe how key SIMD features are modeled on a MIMD machine. For more details see [42, 60]. Scalar variables, and operations on them, are replicated on all compute nodes. Scalar I/O operations are intercepted so only one compute node physically performs I/O and broadcasts the results to the others. Parallel data are distributed among the compute nodes of a MIMD computer. Although the logical view of the data is that each VP holds its elements of parallel data, parallel data are grouped together by variable rather than by VP. That is, all the elements of parallel variable **x** assigned to a compute node are stored in contiguous memory. Parallel code is implemented through the use of *VP emulation*, in which each physical processor performs the operations of the VPs assigned to it. Standard C* functions are implemented using a single function call on each compute node, and VP emulation and synchronization are performed in the function. Elemental functions, on the other hand, are called once for each active VP being emulated by a compute node; VP emulation is performed outside of the function.

```
for (i = 0; i < 256; i++) {
  for (j = 0; j < 128; j++) {
    int temp = [i][j]x;          /* Convert parallel to scalar */
    fwrite(&temp,sizeof(int),1,scalarFile);
  }
}
```

FIGURE 3.3. Writing parallel data to a scalar file using standard C file operations. Extra code is required to work around the limitation that only scalars can be input or output.

3.3. Parallel I/O within C*

3.3.1. Scalar Files

Scalar files, including `stdin`, `stdout`, and `stderr`, are standard C files managed by the front-end processor in scalar code. The use of standard, scalar C operations for parallel I/O would let programmers use familiar, well-understood routines. Unfortunately, scalar file operations are not suited for parallel data. The code in Figure 3.3 shows how parallel data must be converted to scalar in order for standard C file operations to be used.

The code shown suffers from two drawbacks. First, each execution of the assignment `int temp = [i][j]x;` requires an expensive communication: one-to-all broadcast. Second, when I/O is performed serially by only one processor, available parallel I/O hardware facilities are wasted.

3.3.2. Parallel Data—Single Stream

Since standard sequential C file operations cannot realistically be used for parallel data in C*, a new programming interface must be introduced to support

parallel I/O. Typical parallel I/O systems support the notion of a single stream of data accessed by many processors; we first examine this approach, which assumes a scalar file accessed using special operations. These operations can provide the programmer with either a global view or the VPs' local view.

The global view, implemented by Thinking Machines for their C* I/O interface using `CMFS_write_file()`, `CMFS_read_file()`, and later `fwrite` and `fread`, transfers all VPs' values for a variable, regardless of the VPs' context (based on a `where` statement), to or from a single stream [94]. These operations can be performed in parallel at high speed, but they provide little flexibility and abandon the local VP view that makes programming easy in the first place.

If we take the local view into account, VPs may or may not read or write during an operation; each may transfer different amounts of data to or from the single stream. Although this approach is more flexible than the global view, it suffers from several drawbacks. First, it forces serialization of compute nodes during file operations, since the file offset for a given node is not known until those preceding it have completed reading or writing. Second, file operations cannot be used in elemental functions. Recall that elemental functions cannot have side effects; writing to or reading from a single, shared stream is a side effect. Hence, debugging messages cannot be written to a file from an elemental function. Moreover, the single stream does not allow us to identify the data written by a given VP. Finally, the single stream forces the programmer to take a global view of the file data, so the local view is only partly supported.

3.3.3. The Alternative—Parallel Streams

Although the single stream approach can provide fast file operations, its inflexibility limits its utility. Further, it does not fit in with the C* programming paradigm. In contrast, assume that a data-parallel file contains one stream for each VP. File operations can be programmed from the VPs' local view, file operations in elemental functions will not have side effects, and each VP can read from a file exactly the data it wrote earlier.

With this scheme, a data-parallel file contains one stream for each VP. Standard C file operations can be overloaded to allow the use of a parallel file pointer. When a parallel file pointer is passed to a file operation, each active VP performs the operation specified on its stream in the parallel file. Furthermore, the scalar C file operations can be used in elemental functions to allow VPs access to their individual streams. The result is that the programmer can use familiar file operations, and the compiler and run-time system work together to manage the parallel streams. The example in Figure 3.4 shows the familiarity of operations when parallel streams are used—the only difference from standard C operations is a shape pointer argument added to `fopen`, which lets an input file specify the shape at run time.

Despite the compelling reasons for using Stream* in data-parallel languages, parallel programmers will use Stream* only if it has high performance. In the next section, we describe strategies that make Stream* fast for most operations, especially those most commonly found in data-parallel applications.

3.4. Stream* Modes

In our Stream* virtual processor file implementation, we rely on three file modes. Two modes limit the available operations in exchange for speed, while one

```
shape [2][4]S;
int    scalarvar;
FILE:S *outfile, *errfile;
int:S  someint;

.
.
.

  with (S) {
    outfile = fopen("temp","w",&S);
    fwrite(&someint,sizeof(someint),1,outfile);
    fclose(outfile);
    where (someint < 0) {
      errfile = fopen("debug","a",&S);
      fprintf(errfile,"Error, someint was %d\n",someint);
      fclose(errfile);
    }
  }
```

FIGURE 3.4. Writing parallel data to parallel streams using familiar C operations.

mode supports more general operations. The restrictions on operations for the high-performance Stream* modes are shown in Table 3.1. We say an operation meeting the restrictions for a given mode are *compliant* with that mode. All three modes break the file into small (usually less than 64 bytes), fixed-sized VP blocks, from which VP streams are built. VP blocks, while much smaller, are analogous to disk blocks in a traditional file system, in which streams consist of an ordered collection of blocks. The Stream* modes lay out VP blocks in the file in different ways. The high-performance modes, which require that *all* VPs move the same amount of data during an operation, lay VP blocks out in a structured, array-oriented fashion that is independent of the number of compute nodes. That is, assuming V total VPs, VP v's stream consists of the vth block and every Vth block after that. Although a VP block logically has a small size, data are moved in large, contiguous chunks between the regularly laid out file and compute node memory.

The highest performance mode is No Buffering (NB) mode. In this mode, VP-level file operations are implemented in parallel using a single, collective file system operation. This operation moves VP data directly between the file system and the desired parallel variables on the compute nodes with no intermediate buffering. Figure 3.5(a) shows how the array-oriented layout of data on compute nodes matches the desired layout in the file. Note that, to maintain the regular file layout with no intermediate buffering, *all* NB-compliant operations must transfer the same amount of data.

Collective Buffering (CB) mode supports more general operations such as transfer of strided data, in which the layout of data in compute node memory does not directly match the structure of the file. In this case, a VP-level file operation is implemented by copying data between fixed-sized VP buffers on the compute nodes and the desired parallel variables. The CB buffers themselves are contiguous, so their

layout in memory matches the array-oriented layout of data in the file. Because the restrictions on CB guarantee that all VPs read or write the same amount of data during a given operation, all VP buffers become full (when writing) or empty (when reading) at the same time. An example showing the buffers filling with VP data during a write is shown in Figure 3.5(b)-(d). As with NB, all the data are moved between compute nodes and the file system in a single, collective large-grained operation that can take advantage of optimizations such as disk-directed I/O [52] and the two-phase access strategy [27]. Recall that the VP block size for NB is dependent on the amount of data written by VPs. The CB VP buffer size, and hence the VP block size in the parallel file, does not depend on the size of VP writes, although the amount of data written by each VP during the first write could be used to choose the CB block size. Since subsequent writes may move different amounts of data, this may not always prove the best heuristic. Other parameters an advanced version of Stream* might use to choose a CB VP buffer size include the amount of memory available and the optimal transfer size for the physical file system. Our current implementation defaults to an 8-byte CB buffer size whose value can be changed on a per-run basis via command-line arguments.

It is important to point out that Stream* does not suffer the problems of a general-purpose file system when buffering VP data on compute nodes. A more general file system must immediately perform writes to a parallel file to ensure that other processors see the update. If compute nodes cache prefetched data for reading, they must ensure the data remain consistent with the physical file contents. With Stream*, however, each VP accesses only its own stream, and only the compute node emulating a VP can update its blocks in the file system. Therefore, each compute node is guaranteed to have the most up-to-date information for the VPs it emulates.

Collective Buffering (CB) Mode Restrictions:
1. All VPs are active during every operation
2. All VPs transfer the same number of bytes *in a given operation*
3. No elemental file operations are performed
4. All VPs choose the same file offset *in a given seek operation*

No Buffering (NB) Mode Restrictions:
1. All of the above restrictions, plus
2. *Every* read/write operation transfers b_{NB} bytes per VP
3. VP data are contiguous (unstrided) in compute node memory
4. All seek offsets are an integral multiple of b_{NB}

TABLE 3.1. VP operations which allow use of Stream* high-performance modes.

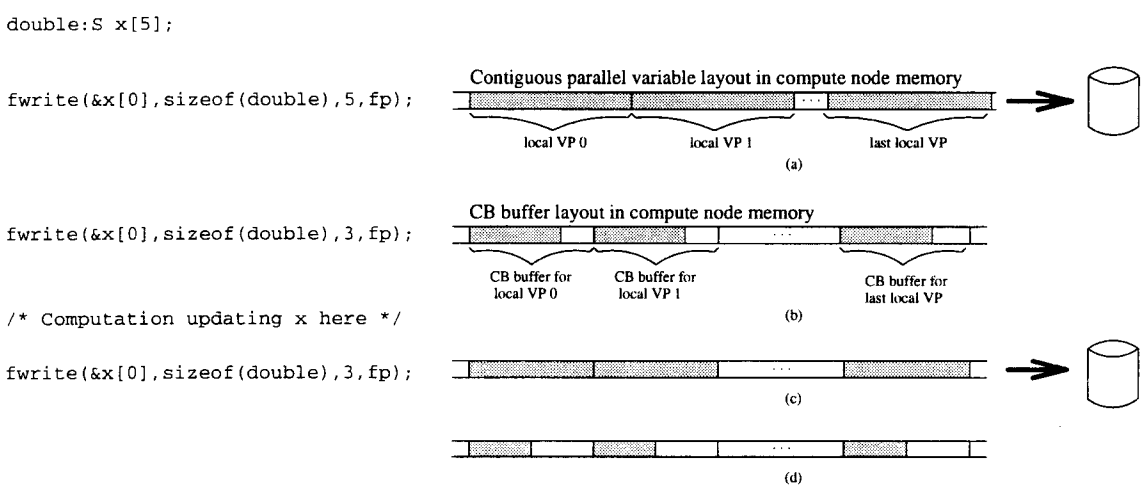


FIGURE 3.5. VP-level writes implemented using high-performance modes. (a) In NB, the parallel variable can be moved directly from its location in memory to the file system with a single, collective file system call. Collective Buffering (CB) mode is shown in (b)–(d). In the CB example, each VP writes two sets of 24 bytes (assuming 8-byte `doubles`) to its own 32-byte CB file buffer. The state of the CB buffers, each containing 24 bytes after the first write, is shown in (b). The second write takes two steps. The VP buffers are filled with 8 bytes each and written to the file system as a single unit, as shown in (c), then the remaining 16 bytes for each VP are copied into the buffers (d).

Independent Buffering (IB) mode is the most general, supporting any C file operations at the VP level. Because some VPs may be inactive during a file operation, or VPs may move different amounts of data, the collective, shared-offset, array-oriented implementation of NB and CB cannot be used for IB. Although the regular layout of data created by NB and CB eases the job of distinguishing individual VP streams, such a layout cannot be efficiently utilized in IB mode. VPs using IB may not fill their buffers at the same time or in any specific order. If, when IB is used, blocks making up VP streams are laid out in regular fashion as in NB and CB, each VP buffer must be written individually using an expensive, fine-grained file operation. An early prototype showed the cost of these operations to be prohibitive. Our solution, which combines fine-grained VP file operations into large-grained calls to the file system, writes VP data to the file on a first-come-first-served basis and manages it using a directory. A description of the implementation is in Section 3.6.

In our first VP file system implementation [65], the user wanting high-performance had to specify the desired mode when declaring a parallel file variable. This approach has several drawbacks. First, the programmer must understand implementation details to choose the correct mode. Second, unfamiliar syntax must be added to the language to specify mode hints. Finally, the single mode assigned to a file must be the most general with which it is accessed, even if many of the operations on the file could be performed using less general, higher performance modes. Our current design eliminates these drawbacks through the use of dynamic mode detection, which chooses the best possible mode for each operation based on its parameters and the current mode.

3.5. File Segments

Stream* dynamic mode detection uses a scheme that divides a parallel file into three distinct segments. The first segment contains VP data written using NB mode, the second contains data written using CB mode, and the last contains data written using IB mode. We name the segments after the modes in which they were written (NB, CB, and IB). The scheme assumes the program writing a file will use NB, the highest performance mode available. Recall from Table 3.1 that NB requires that all operations move the same amount of VP data. The first NB write establishes the VP block size b_{NB} for the NB segment of the file. If a program writes only one data type to the file, as is frequently done in data-parallel applications, the file will consist entirely of an NB segment. When a non-NB-compliant write operation, or an NB-compliant operation relying on a different value of b_{NB} is used, the NB segment of the file is complete. A transition to CB or IB takes place.

The CB segment of the file is written using a VP block size of b_{CB}, whose value is determined by the run-time system, until an operation requiring IB is encountered.[2] The remaining writes, assuming no backwards seeks, are performed using IB to the IB segment of the file, even if subsequent operations are NB or CB-compliant. A scheme could be developed to allow unlimited switching between modes, but the costs of this approach potentially outweigh the benefits. The volume of metadata describing the mode switches and VP activity in every IB segment potentially would be huge. IB's performance relative to the other modes' is good enough that the extra overhead for allowing more file segments is not justified. Finally, files used by most data-parallel applications would remain in NB or CB

---

[2]Note that operations allowed in NB mode form a subset of those allowed in CB.

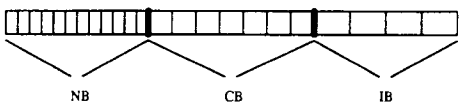


FIGURE 3.6. The segmented nature of a Stream* file. The first segment is written using NB mode, and VP blocks of size b_{NB} are regularly laid out in the file. The second segment is written using CB mode. This segment has a structure similar to that of NB; the VP block size b_{CB} may differ from that of NB. The last segment consists of VP blocks of size b_{IB} with an unstructured layout. The IB segment is read with two directory files which describe which blocks make up which VP streams.

modes during their lifetimes [22, 36, 56, 63, 64, 77], and Stream*'s prioritization of the highest performance modes matches their needs without becoming overly complex.

Associated with each Stream* data file is a second *metafile*, whose name is the concatenation of the data file name with `.meta`. This file contains information about the file such as b_{NB}, b_{CB}, b_{IB}, mode transition points, distribution information, and shape. Files whose IB segment is not empty also have files with suffixes `.first` and `.dir` to hold directory information. Their function will be described in section 3.6.2.2. In the remainder of the chapter, we refer to the meta files by their suffixes (e.g., the `.meta` file).

3.6. Implementation

Stream* is implemented as part of the C* run-time library. Machine-independent routines make up a majority of the Stream* implementation, while a few routines make calls to the machine-dependent file system. We assume the underlying file system manages a parallel file in the traditional manner, as a single stream of bytes. Therefore, Stream* can be implemented on top of PFS [45],

Thinking Machines' CMMD I/O [6], or the single stream views of more flexible file systems like PIOUS [67] and Vesta [14]. Knowledge of the underlying implementation of the file system (e.g., programmable I/O nodes, disk arrays, etc.) may be used to optimize the machine-specific portions of the Stream* run-time system, but our discussion is independent of the file system. All of the features of Stream* are implemented on the compute nodes.

For our experiments, we built a simple striped file system using varying numbers of nodes on a Meiko CS-2 multicomputer. Although a parallel file system is available with the CS-2, it is not installed on our machine. Instead, specific nodes are selected as I/O nodes, and they read from and write to their local SCSI disks using Unix file operations. File operations are requested and fulfilled through messages on the CS-2's low-latency, high-bandwidth network. For the experiments shown in subsequent sections, eight compute nodes were used with one to four I/O nodes and a striping unit of 32K bytes.

3.6.1. Opening a File

When a parallel file is opened, a parallel `FILE` variable is allocated and a pointer to the parallel variable is returned. A sequential C `FILE` has a corresponding Unix file descriptor, or `fd`, an integer. Our Stream* implementation also associates an `fd` with a `FILE` variable. The `fd` is an index into a structure containing the data needed to manage the parallel file. Our system reserves a fixed number (0-63) of `fd` values for sequential files, and `fd` values higher than that correspond to parallel files. The typical implementation of a `FILE` struct allocates a byte to the `fd` field, so 256 `fd` values are available to represent both sequential and parallel files, although the underlying operating system may not support that many open files simultaneously.

3.6.2. Writing

Although C provides several ways to write to a stream, our examples below are based on the parallel overloading of the function `fwrite`:

```
int:current fwrite(char:current *buf,  int:current size,
                   int:current nitems, FILE:current *fp);
```

Note that the C* keyword `current` matches the current shape, so the `fwrite` function can be used for any shape. Depending on whether or not all VPs are active, and depending on the parameters for an invocation of `fwrite`, this function can comply with any of NB, CB, and IB modes. The mode detection performed at the start of `fwrite` is based on the current mode, file segment, and the characteristics of the parameters. The goal is to use the fastest mode possible, with IB being selected if NB and CB tests fail. To use **NB or CB** for `fwrite`, the mode detection logic checks the following:

1. NB requires that the current mode be NB. CB allows a current mode of either NB or CB.

2. The file segment to which the write will occur must be compatible with the mode to be used. CB can be used to write to the NB segment of the file using a buffer size of b_{NB} (after a backwards seek, for example), and only in rare instances—the current implementation does not check for these—can NB be used to write to the CB segment of the file.

3. All VPs must be active. Two tests are used to determine VP context. The C* compiler emits code to manage a flag called `CS__everywhere`. The flag is true when VPs are not masked off by a `where` clause. If the flag is false, meaning a

`where` clause is in effect, each compute node can directly test whether or not all its VPs are active.

4. `size * nitems` is the same for all VPs.

In addition to the above, the following conditions must hold to use **NB**:

1. `size * nitems` is equal to b_{NB} for all VPs (if this is the first write to the file, b_{NB} is established for subsequent tests).

2. The stride of the parallel data being written is equal to `size * nitems`. That is, the data are contiguous in compute node memory. The UNH C* compiler stores the stride as part of each parallel variable.

Different compute nodes may get different results from the above tests. For example, on exactly one compute node, a VP may have performed a file operation in an elemental function. That compute node will come into the `fwrite` with a current mode of IB. All others will have a current mode of NB. To guarantee that all compute nodes are using the proper mode, a reduction is performed. The processors exchange the calculated mode, number of bytes transferred per VP, and all-active status of VPs with each other. After the reduction, all compute nodes agree on the mode. The reduction operation is cheap relative to file operations, and it can act as the synchronization for a collective file operation, since both NB and CB can take advantage of collective operations.

If NB is the agreed-upon mode, the compute nodes perform an efficient, array-oriented transfer of data directly from the parallel variable to the file system. If CB is chosen, the compute nodes copy from the specified parallel variable to the VP buffers. If the VP buffers become full, an efficient array-oriented transfer, like that for NB, moves data from the contiguous VP buffers to the file system. As shown

in Figure 3.5, the VPs may write more bytes than their buffers can hold. In this case, the buffers are filled to capacity and sent to the file system. This copy-and-write cycle continues until the VP buffers can store the remaining VP data.

3.6.2.1. Enhancing the Interface and the Performance

The version of `fwrite` presented above offers general functionality that may be rarely exploited. For instance, the `size` and `nitems` parameters are most often constants, with the `size` often denoted using the `sizeof` operator. With the general `fwrite` prototype, the programmer must cast constants to parallel values, e.g.:

```
fwrite(&parVar, (int:S)sizeof(double), (int:S)1, parFile);
```

Further, the run-time system must allocate and initialize parallel arguments for the call to `fwrite`. Finally, these arguments must be checked, element-by-element, during mode detection to ensure that all VPs write the same amount of data. To eliminate these frequently unnecessary costs, Stream* provides another overloading of `fwrite` (and, correspondingly, `fread`), in which `size` and `nitems` are scalars:

```
int:current fwrite(char:current *buf, int size,
                   int nitems,  FILE:current *parFile);
```

With this version, no parallel arguments must be built, and checking for size consistency among VPs is unnecessary. We expect this to be the normal usage of `fwrite`, so this is the version we use when comparing performance in subsequent experiments.

Because NB is performed using the fastest file operations, it is the standard against which other modes are measured. Figure 3.7 shows that the differences between NB and CB modes, despite the extra buffering required by CB, is negligible when repeatedly writing a simple parallel `double`. The results shown in Figure 3.7

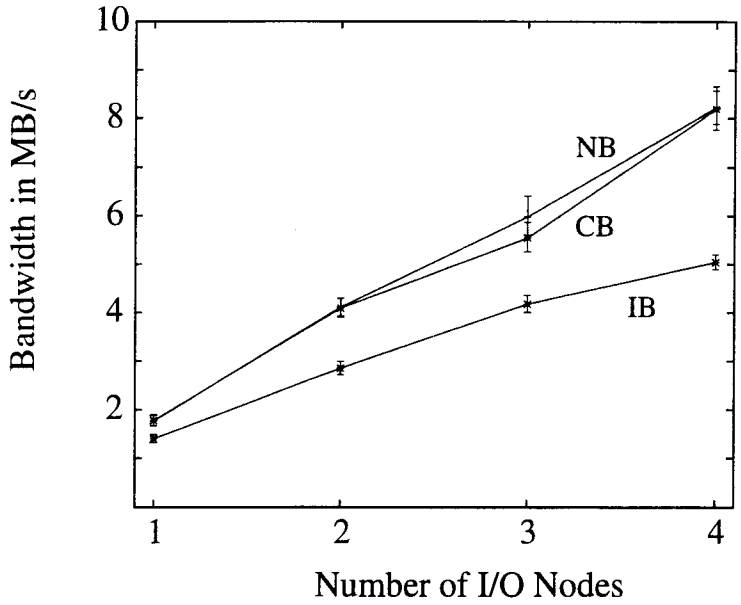


FIGURE 3.7. Comparison of NB, CB, and IB modes on the Meiko CS-2. 64K VPs each output 64 `double` values for a total of 32 MB. 95% confidence intervals are shown.

are with a CB block size b_{CB} of 8 bytes. A block size of 32 bytes takes more compute node memory and may result in inefficient cache usage, but the larger buffers result in fewer file operations. Our experiments show no statistically significant difference between CB performance with 8 and 32-byte buffers, so the current default b_{CB} is a space-saving 8 bytes. The mode detection cost of Stream*, that is, the bandwidth difference between Stream* with mode detection and without, is also negligible on all I/O node configurations. The largest difference between NB bandwidth with and without mode detection is 3.9%, while the standard deviations of the measurements are 5.6% and 6.0%.

3.6.2.2. Writing in IB Mode

If IB is the selected mode, a VP emulation loop performs the write for each active VP by moving data from the parallel variable being written to the VP's IB buffer. VPs may fill their compute-node buffers at different times, which is why they are managed individually, but our goal is to avoid fine-grained writes of individual VP blocks. Figure 3.8 shows the steps taken when a VP fills its buffer. The contents are moved to a *superblock*, a collection whose size would typically be the size of a striping unit in the underlying file system. Each physical processor maintains its own superblock for each file in IB mode. When a superblock is filled, the data contained in it are sent to the file system as a single chunk. VP directory information is also managed using the notion of superblocks. By combining VP file writes into superblocks, our implementation avoids many fine-grained writes of VP data, which would significantly degrade performance. To support IB superblock writes without processor synchronization (recall that the superblock may be filled from an elemental function, in which no synchronizations may occur), each compute node must know where in the IB segment of the file to write its data. We allocate superblocks in the data file and directory file to compute nodes in round-robin fashion; the pth processor among P total processors writes to the pth superblock, followed by every Pth superblock. This does mean that the structure of the data in the file changes with the number of physical processors, but alternative file layouts relying on virtual processor instead of physical processor configuration require expensive fine-grained file system operations to write VP blocks. This scheme may also result in "holes"—unused space—in the parallel file. In extreme cases, when one physical processor's VPs write much more than any others', file holes might waste considerable space. Since such programs will poorly balance the computational load as well as the I/O

	IB Block Size						
	64 bytes	32 bytes		16 bytes		8 bytes	
# I/O Nodes	Bandwidth	Bandwidth	% 64	Bandwidth	% 64	Bandwidth	% 64
1	1.37	1.17	85	0.85	62	0.61	45
2	2.89	2.36	81	1.87	65	1.06	37
3	4.14	3.44	83	2.55	62	1.63	39
4	4.96	4.47	90	3.59	72	2.42	49

TABLE 3.2. Comparison of achieved I/O bandwidth in megabytes per second when writing in IB mode with different IB block sizes. The % 64 column shows the percentage of the bandwidth achieved relative to when $b_{IB} = 64$. All bandwidth values are distinct at the 95% confidence level.

load, their general performance will be poor, and thus they should prove to be the rare case.

Figure 3.7 shows that the bandwidth achieved by IB is approximately 60% of NB's bandwidth. Note that the times shown include closing the file in IB, an operation with considerable overhead, to be discussed in Section 3.6.2.4. With a faster file system, the time for the extra work done on compute nodes limits the achievable bandwidth. This explains the flattening of the IB curve as the number of I/O nodes increases.

Several variables play a part in IB performance. The first of these is IB block size, b_{IB}. A large block size requires more compute node memory, but the buffer management overhead becomes a smaller percentage of work done on compute nodes. In the benchmark shown in Figure 3.7, $b_{IB} = 64$. Table 3.2 shows the relative performance with smaller b_{IB} values. We see a steady decline in performance as b_{IB} decreases to 32, 16, and 8. Our system defaults to a b_{IB} value of 64 bytes; the user can override this value on a per-run basis using a command-line argument. The cost of reading, discussed in Section 3.6.3, must be considered along with the time-space

tradeoff shown in Table 3.2 to select an appropriate b_{IB} value. Another variable is the number of data block pointers in a single directory entry. The benchmarks shown were run with only two pointers per directory entry. By increasing that number to six[3], the bandwidth increases on all I/O node configurations by approximately 7%. Increasing the number to fourteen provides negligible additional bandwidth.

Asynchronous I/O, in which file operations are overlapped with computation, can sometimes be utilized to increase performance. When asynchronous I/O is used, the latency, or time the run-time system spends buffering data, is of greater concern than bandwidth. The latencies for all three modes vary widely based on the number of VPs per compute node, the cache size, the VP buffer size, and the amount of data written by each VP. We give here a simple comparison of latencies based on the experiments described in Figure 3.7. NB latency is 0.000839 μs per VP, CB latency is 0.086 μs per VP, and IB latency is 6.550 μs per VP. These values reflect the cost of buffering data on the compute nodes and assume messages to the file system and subsequent disk operations take place in the background.

3.6.2.3. Mode Transitions During Writing

A mode transition can potentially take place on a subset of the compute nodes when they perform file operations from within elemental functions. The other nodes do not need to be informed of the transition as it occurs (in fact, they cannot be informed, since the run-time system cannot perform communication within elemental functions), since they will either transition on their own upon performing elemental file operations, or they will be informed of the transition at the

---

[3]With 2 other values in a directory entry, these block pointer counts (2, 6, and 14) ensure a power-of-2 total size in bytes.

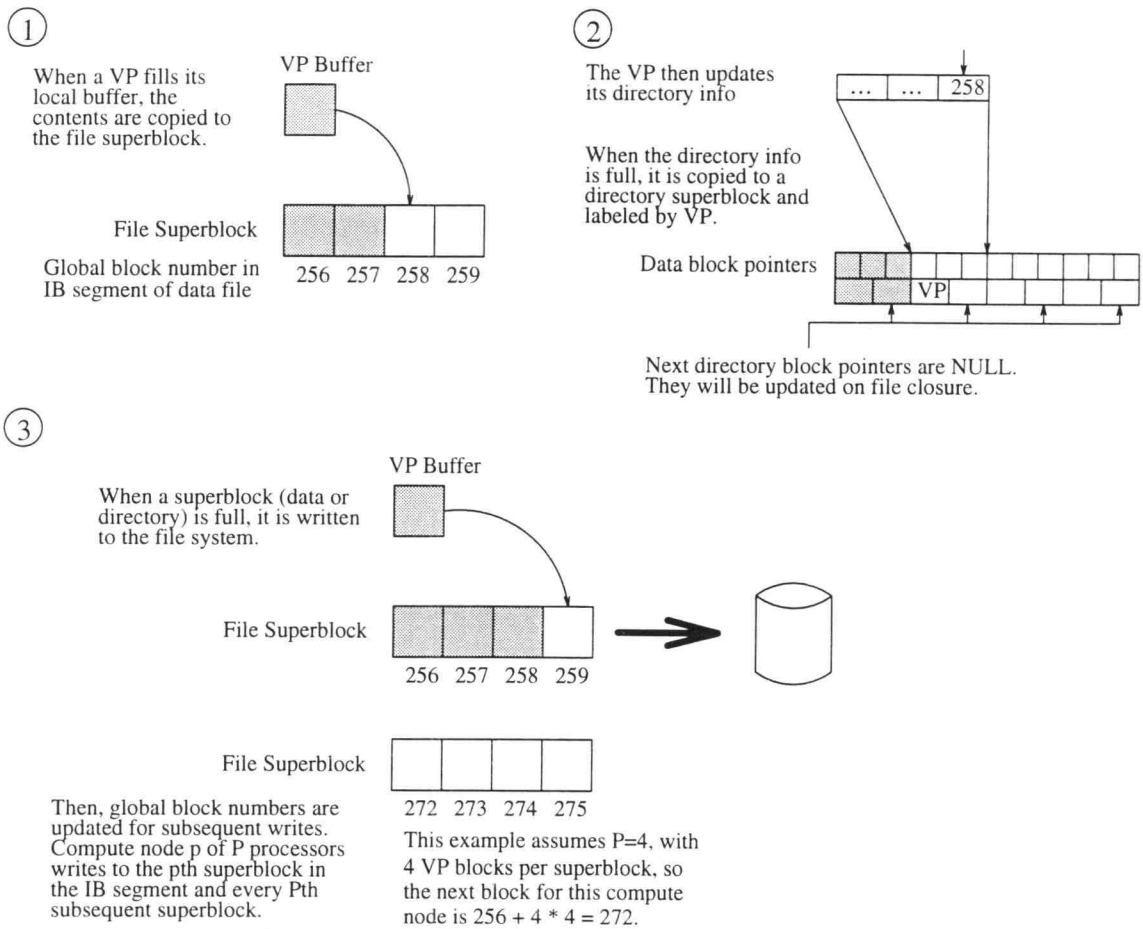


FIGURE 3.8. Using superblocks for VP writes in IB mode. Each VP has its own active file buffer. When it is filled, the buffer is copied to the superblock. When the superblock is filled, it is written as a unit to the file system. The VP keeps track of which data blocks its data has been written to, and this directory information is written to the directory file using superblocks.

next non-elemental file operation. Non-elemental file operations are preceded by a reduction to determine the mode for the operation. If some compute nodes are in IB mode, the remaining nodes will transition after the reduction and prior to performing the requested operation. If no compute nodes are in IB mode, but the requested operation requires IB mode, the reduction results in all compute nodes transitioning at the same time. A transition to IB may require flushing and deallocation of CB buffers, if they are not empty. Then, IB buffers, superblocks, directory superblocks, and VP directories must be allocated. The parallel `FILE` variables, used to manage individual VP streams, are initialized. The IB transition is now complete, and the requested operation is performed, either for an individual VP (in an elemental function), or in a VP emulation loop in a non-elemental function. The simpler transition from NB to CB entails allocation of CB buffers.

3.6.2.4. Closing a File

When a file is closed, housekeeping must be performed. Regardless of the mode(s) used to write a file, the sequential `.meta` file is written by compute node 0. A file closed while in CB must have its VP buffers flushed if they aren't empty. The `total_CB_data` field in the `.meta` file lets the reader know how many bytes of the last CB block are valid.

A file written using IB requires more work on closing. First, the VP buffers are flushed to superblocks. Then, the superblocks are written to the parallel file. Note from Figure 3.8 that the directory information written so far contains VP values but NULL `next` values. These are updated by reading directory superblocks in reverse order, updating `next` pointers, and writing the data back again. A linked list is formed, with the head of each list stored in the `.first` metafile. Although this

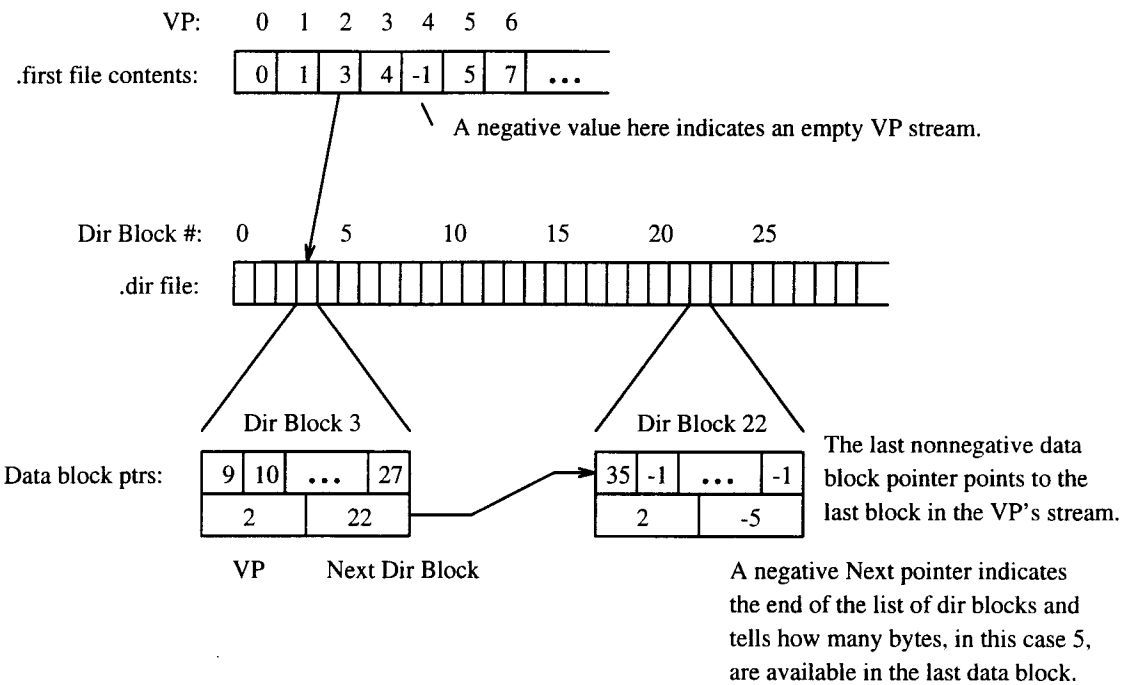


FIGURE 3.9. Directory structure for VP data blocks in a single file. The `.first` file contains a pointer to the first directory block for each VP. Each directory block contains pointers to a single VP's data blocks in the IB segment of the data file. The `Next` pointer in a directory block is used to build a linked list containing all of a VP's directory blocks.

directory patching step incurs extra overhead, it is performed using large-grained file operations. An earlier prototype updated the directory in a small-grained fashion as VP blocks are written; this approach reduced effective bandwidth by an order of magnitude. One might consider omitting the patching step, since the VP values in the directory blocks allow the streams to be reconstructed again. However, we chose to eliminate the need for a reading program to search the directory. As parallel applications' file needs become more well-understood, we may find it necessary to change the directory structure entirely, or perhaps to use a doubly-linked list. All the performance figures presented for IB include the extra cost of updating the directory upon closing of the file.

3.6.3. Reading

A file may be opened for reading using one of two `fopen` overloadings:

```
FILE:current *fopen(char *name, char *type);
FILE:void    *fopen(char *name, char *type, shape *s);
```

The first of these requires that the existing file be of the same shape as `current`. The second, the only Stream* operation whose usage is not analogous to that of traditional C, returns a parallel FILE variable whose shape is defined at run time from the `.meta` file. The `void` shape of the return value specifies that it can match any shape, while the actual shape of the file opened is returned via the `s` parameter.

The mode detection performed for reading is essentially the same as for writing described in section 3.6.2. NB can be used to read only from the NB segment

of the file[4]. CB can be used to read data from the NB and CB segments, because their structure is identical aside from the VP block size. IB can be used to read from any of the file segments. The VP block size for each segment is read from the `.meta` file, so the VP buffers on the compute nodes exactly match the VP blocks on disk.

Reads in NB move data directly from the file system to parallel variables. In CB, data are moved to VP buffers on the compute nodes; VP-level reads then move data from the buffers to parallel variables. As shown in Figure 3.10, bandwidth achieved by CB scales well with the number of I/O nodes but slightly lags NB bandwidth. IB is slower due to the extra buffer manipulation on the compute nodes as well as the extra reads required to get directory information.

3.6.3.1. Reading in IB Mode

Reading in IB mode was designed to emphasize movement of collections of VP blocks rather than individual blocks between compute nodes and the file system. The design is based on the fact that most programs read a file in the same way it was written. In this case, VP blocks written to the file system in the same superblock will be needed in the reading program at approximately the same time. Therefore, when a VP block (data or directory) is needed from the file system, a superblock containing the desired block and subsequent blocks is read. Like all prefetching schemes, this one may actually hurt performance when a particularly ill-behaved read pattern is used. However, if no prefetching is done, or if each prefetch reads a

---

[4]If the VP block sizes for the NB and CB segments are identical, NB could be used to read the CB segment. We have not implemented this optimization.

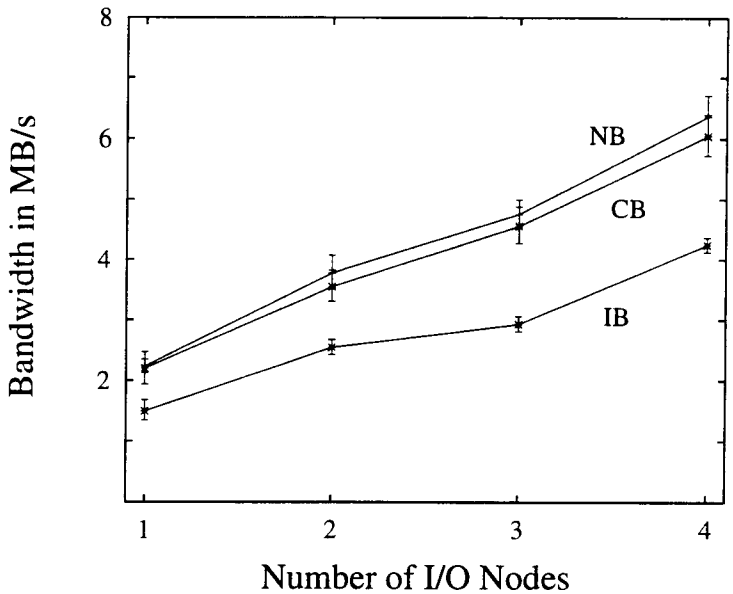


FIGURE 3.10. Comparison of NB, CB, and IB modes for reading on the Meiko CS-2. 64K VPs each read 64 `double` values for a total of 32 MB. 95% confidence intervals are shown.

single VP block from the file system, file accesses are fine-grained and guaranteed to be slow. We feel that prefetching based on VP write patterns is a good heuristic for avoiding fine-grained file accesses.

The reading process for IB mode is detailed in Figure 3.11. Several variables impact the IB reading performance. As with writing, one of these is b_{IB}. Table 3.3 shows the relative performance when several values of b_{IB} are used. Reducing b_{IB} has an impact on bandwidth, but not as much as for IB writes. The size of a directory entry affects reading performance as it does for writing. That is, moving from two to six data block pointers per directory entry increases performance by about 6%, but increasing that value to fourteen data block pointers makes negligible additional impact. Finally, the number of read buffers impacts performance. For the results presented in Figure 3.10, the compute nodes use only two read buffers

	IB Block Size						
	64 bytes	32 bytes		16 bytes		8 bytes	
# I/O Nodes	Bandwidth	Bandwidth	% 64	Bandwidth	% 64	Bandwidth	% 64
1	1.50	1.49	99	1.21	81	0.94	63
2	2.54	2.07	81	2.15	85	1.51	59
3	2.93	2.45	84	2.05	70	1.58	53
4	4.24	3.97	94	3.52	83	2.67	63

TABLE 3.3. Comparison of achieved I/O bandwidth in megabytes per second when reading in IB mode with different IB block sizes. The % 64 column shows the percentage of the bandwidth achieved relative to when $b_{IB} = 64$. Bandwidth values are distinct at the 95% confidence level for 3 and 4 I/O nodes.

of 32K bytes each. During the reads, one buffer holds the directory information while another holds data. Because the data are read exactly as written, no more read buffers are needed. In other cases, more read buffers may be needed to cache data for extended periods. Unfortunately, our current implementation uses a simple linear search of buffers to satisfy a given request; the search becomes expensive as the number of buffers increases. We shall update the search to a more efficient scheme; until then, our implementation handicaps reads relying on more than a few read buffers.

3.6.3.2. Mode Transitions During Reading

Mode transitions can either be forced by the segmented nature of the file (e.g., when the CB segment has been exhausted, a transition to IB must occur), or through the parameters of the requested operation. For example, data in the NB segment of a file may be read using CB if the parallel variable being filled has a stride. A transition to CB requires allocation of CB buffers on the compute nodes. The size of the CB buffers is based on the b_{CB} value stored in the `.meta` file. If the

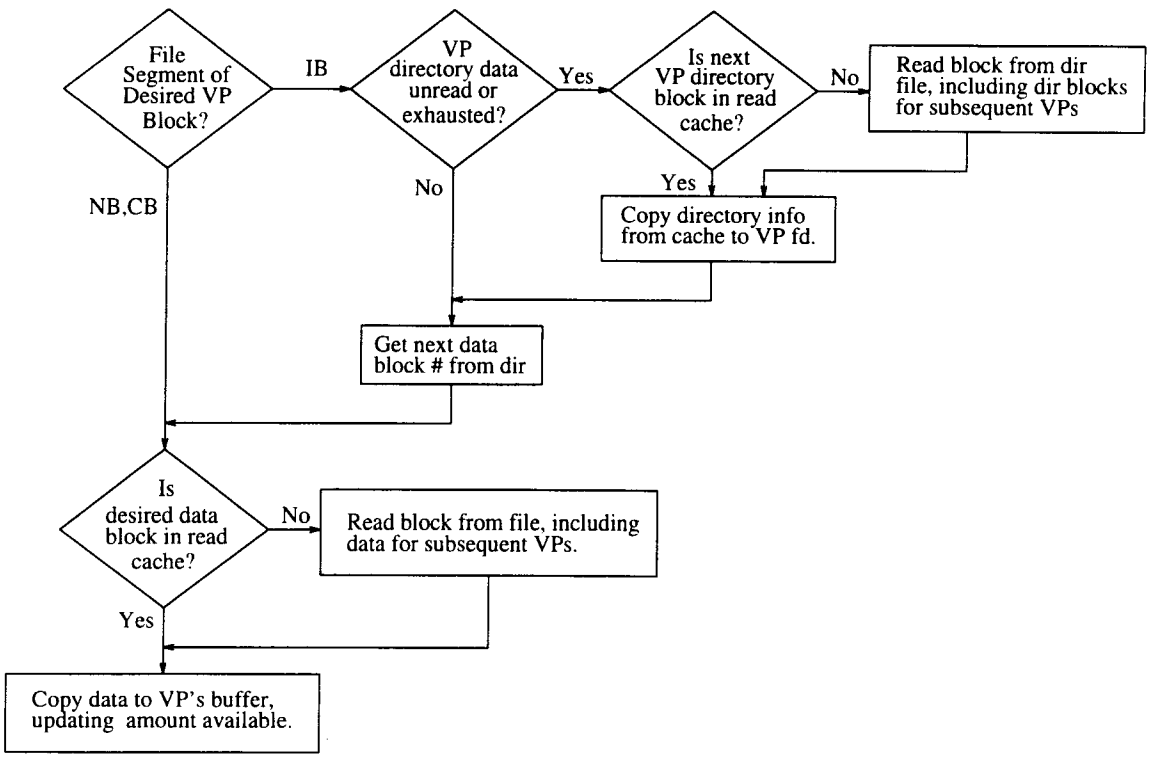


FIGURE 3.11. Simple decision flow for reading a single VP block in IB mode. This sequence may be repeated several times to satisfy a user-level request.

NB segment is being read, and if $b_{NB} < b_{CB}$, b_{NB} is the effective block size until the CB segment of the file is reached.

Transitions to IB occur in two steps. The first step allocates and initializes the read buffers. If the NB or CB segment of the file is being read, directory information is not needed. When the IB segment of the file is reached, the second step of the transition takes place. The `.first` file is read to get the first directory block pointer for each VP. Actual directory blocks are not read until individual VPs perform read operations.

3.6.4. I/O with Elemental Functions

As shown in [65], a single-stream file model for C* cannot support file output in elemental functions. Here we show that, to maintain a familiar programmer's interface, formatted input and output, critical when debugging at the VP level, must be performed using elemental functions. The functions `fprintf`, `vfprintf`, and `fscanf` receive a variable number of arguments whose type is specified in a format string. The programmer writes, for example:

```
fprintf(parFile,"Variable someVar is %d\n", someVar);
```

and wants the semantics to match `someVar`'s type (scalar or parallel). Unfortunately, the run-time system must rely exclusively on the type specifier in the format string to determine the type of `fprintf`'s arguments, and `%d` dictates a single integer, not a parallel variable. Two obvious solutions present themselves: allow a new parallel type specifier in the format string, or call `fprintf` once per VP so the values passed in are logically scalars. The former solution does not maintain the familiar C programming interface, while the latter, namely implementation of formatted output via elemental functions, does.

The above example shows a file operation implemented as an elemental function. A file operation may also be called from within an elemental function. The Stream* library includes its own versions of the scalar C file operations; these are called from within elemental functions. They check the `fd` value for the `FILE` variable used. If the `fd` represents a scalar file, the original scalar version of the routine (now renamed) is called. Otherwise, the Stream* handler performs the requested operation in IB mode, forcing a transition to IB on that compute node if necessary. Other compute nodes will be informed of the transition during the reduction in the next collective file operation.

3.6.5. Seeking

The Stream* `fseek` operation, in its most general form, allows each VP to seek to a different location in its stream. Whether reading or writing, typical data-parallel applications will have all VPs seeking to the same position in their streams. When this is the case, and the seek is to the NB or CB segment of the file, NB or CB mode may be used, even if IB was the previous mode. Here we simply point out some implementation concerns. When writing to a file, seeks may require flushing of buffers beforehand and read-modify-write sequences afterward. These operations can be supported in all three modes, although updates in IB mode may require that VP blocks be written individually, since each VP block's position is dictated by the earlier writing pattern. Our implementation does not yet support this type of operation, which does not appear frequently in data-parallel applications.

3.7. Redistribution Issues

So far, we have assumed the distribution of data on the file system matches the data distribution on the compute nodes. When this is not the case, data must be permuted during reading and/or writing. As we have noted, disk-directed I/O [52] or the two-phase access strategy [27] can be used to perform this permutation collectively for NB and CB modes in place of using many small, inefficient reads or writes. When such a permutation is performed, the layout of VP data in compute node memory is not the same as the layout on disk, and data do not move directly between compute node memory and the file system. However, assuming the permutation must be done, the data layout for NB writing (respectively, reading) requires no intermediate buffering of data in VP buffers before (after) the permutation. Therefore, NB's performance is comparable to that of an array-oriented system requiring a redistribution. CB performance can be enhanced by increasing b_{CB}, essentially the array element size, when redistributions are required; redistributions are faster in terms of bytes/second when data consist of larger elements [66].

C*'s definition [92] does not include a machine-independent construct for the programmer to specify data distributions. The UNH C* system currently utilizes only block distributions. It may be fruitful to explore adding constructs to support writing of files in a distribution other than the compute-node distribution. When reading, the run-time system requires no additional information from the programmer, since the distribution on disk is known from the `.meta` file.

3.8. Interfacing Stream* to External Programs

Stream* utilizes a unique file format; thus, the files are "internal" [21]. Ideally, Stream* could exchange files with external applications using little or no filter-

ing of files. With NB mode, this is the case. Files consisting of a single NB segment are laid out exactly as an array-oriented program would write them. These files can be used without conversion by external programs. External data files can be treated as files containing only an NB segment. If the first overloading of `fread` in Section 3.6.3 is used, an external file lacking a `.meta` file simply takes on the current shape and is assumed to consist of a single NB segment. A `.meta` file may have to be built using a simple utility program in some situations (e.g., if a file has a different distribution than the program that will read it). Because most data-parallel applications rely on regular array-oriented I/O, the use of NB as an interface to the outside world should work in most situations [22, 36, 56, 63, 64, 77].

Files with CB and IB segments require explicit conversions. A file containing a CB or IB segment can be converted to one containing a single NB segment using a high-level C* program. The program has each VP reading its existing stream and writing its data, or a fixed value upon reaching EOF on its input, in NB mode. Following three rules guarantees that an output file will consist of only an NB segment, which means it will be written at top speed and will be readable by external applications.

- Use the form of `fwrite` described in section 3.6.2.1 with all VPs active.
- Don't output fields of structs or individual parallel array elements.
- Make sure every VP in every call to `fwrite` outputs the same number of bytes.

Note that these rules are at the language level, so the programmer does not need Stream* implementation knowledge to get high performance.

3.9. Conclusions

We have shown that the programmer's I/O interface can be seamlessly integrated with C*'s virtual processor programming paradigm using data-parallel streams. Their implementation using machine-independent, automatically detected modes lets the most common file operations found in data-parallel programs run at the top speed supported by the file system. The high-performance modes, because of their array-oriented nature, can also take advantage of file redistribution optimizations developed for languages such as HPF. The general mode supports a wide variety of file operations while achieving bandwidth over half that of the high-performance operations by combining fine-grained virtual processor operations into large-grained file system operations.

4. ANALYSIS AND MODELING OF ARRAY REDISTRIBUTIONS

4.1. Introduction

On distributed memory parallel computers, data are distributed among processors, and the choice of data distribution can significantly impact performance. Programmers must take into account the interplay between parallel algorithms and data distributions when solving a problem, because many parallel algorithms perform well only when the data have a given distribution. For example, cyclic distributions are often applied to adaptive, irregular, and sparse matrix problems. Block distributions are often ideal for dense matrix problems. In many cases, the programmer cannot use a single data distribution throughout a program; data must be redistributed between tasks. The redistribution may be done for performance reasons, as when a distribution well-suited to one task in a program may force terrible performance for a subsequent task. Other times, the redistribution may be done out of necessity. Examples of such necessary redistributions include the use of a library function requiring a specific data distribution and the use of file data whose file distribution does not match that desired on compute nodes [27].

Redistribution can be expensive, though, and the programmer or a performance prediction tool would benefit from knowledge about the expected execution time. An obvious example is using a redistribution model to decide whether or not the cost of a redistribution coupled with the reduced cost of subsequent tasks is cheaper than the tasks without a redistribution. Another use of an accurate redistribution performance model is to determine the cheapest type of redistribution

performed, since some multi-dimensional and general redistributions can be done in one expensive phase or two or more efficient phases.

Unfortunately, existing performance models of data redistribution are limited in scope. An example is the model of Johnsson and Ho [46], which is geared to hypercubes with array sizes corresponding to machine size. Many approaches to redistribution have been presented [10, 39, 48, 59, 61, 91, 97] with no predictive performance models. Others model the *communication cost* [49] of redistributions. Communication cost, the actual time messages are sent between processors, is only one factor in the total cost of a redistribution. On a parallel computer with a high-bandwidth processor interconnect, total cost is dominated by the time spent building packets to be sent to other processors and the time spent unpacking data from received packets into their proper local array locations. Packet-building for a block-to-cyclic redistribution of a 50 MB array using 16 nodes of the Meiko CS-2 accounts for 76% of the redistribution time; unpacking for a cyclic-to-block redistribution takes 79% of the total redistribution time. Unlike these two simple redistributions, cyclic-to-cyclic redistributions require both packing and unpacking of data; these packet-handling phases take up to 86% of the redistribution time. Further, the cost of building packets is so high for general redistributions that such redistributions can sometimes be faster when performed as the composition of two simpler redistributions, even though the resulting communication costs are doubled [91]. Understanding the packet-handling phases is important not only because they dominate redistribution time, but because the time can vary significantly with only a slight change in variables. For instance, unpacking 1 MB of data from 64 processors for a Cyclic to Block redistribution takes 179 ms on the Alpha machine used in our experiments; the same job with 63 processors takes only 82 ms—less than half the time. This significant variability makes performance prediction impossible

without detailed understanding of the factors involved in building and unpacking packets for redistribution.

In this chapter, we develop and validate a performance model for the costs incurred when moving data between local arrays and communication packets. We show that by empirically running just a few key redistributions we can determine the parameters of a model that accurately describes a variety of redistributions. We point out machine parameters that play an important part in the cost of building packets. We also show the value of these parameters for three different machines so users of our model can discern a reasonable range of expected values to use in their own models.

In the following sections, we describe the redistribution problem and our assumptions. We focus on a single redistribution, the Cyclic(ky) to Cyclic(y) redistribution, to build a basic model. Subsequent sections show how to model the impact of TLB misses on the basic model for TLBs using random, LRU, and Pseudo-LRU replacement policies. TLB effects are also described for redistributions using large k values. Finally, the results are migrated to other redistributions. Sections 4.7 and 4.8 of the chapter describes the strided unpacking of data after the communication step. We show how the cache efficiency of the strided copy can be determined and modeled. TLB effects in the context of the strided copy are also modeled. In all cases, empirical results are given and compared to the model. We close with a discussion and conclusions.

4.2. The Modeled System

Our model describes the computation and data copying costs of a parallel array redistribution. It is is based on the algorithms of Thakur and Choudhary

[90, 91], because they are efficient and easy to implement. In these algorithms, the computation is performed during redistribution, and our model captures the cost of the computations as overhead for a copy operation. Although redistribution schemes exist which may be faster in some cases [10] [59], they require more complex up-front computations. We do not model these preliminary computations, but our model can effectively be applied to the run-time overhead associated with these models.

The data being redistributed are viewed as a global array A of N elements mapped to P processors. Each array element is e bytes. L denotes the size *in bytes* of the local array residing on a given processor. Data distributions are described using HPF nomenclature. A Block(m) distribution allocates the first m elements of A to processor 0, the next m elements to processor 1, and so on in order until all N elements have been allocated to processors. The term Block with no arguments is synonymous with Block($\lceil N/P \rceil$). A Cyclic(k) distribution allocates k array elements to each processor in round-robin fashion starting with processor 0. The term Cyclic with no arguments is synonymous with Cyclic(1). In Section 4.5 we show how our model is applied to a variety of redistributions; we first expand our model in the context of the Cyclic(ky) to Cyclic(y) redistribution, denoted KYY.

In modeling the KYY redistribution, we take into account the system features which impact the performance of the redistribution. These include number of processors, memory bandwidth in the context of packet-building, cache size and block size, and effective Translation Lookahead Buffer (TLB) size and replacement policy. We calculate parameters such as TLB miss penalty and cache miss penalty *in the context of the redistribution operation*, since these values vary greatly depending on the use of the overall system. For example, the TLB miss penalty depends on whether or not page table entries (PTEs) are cached, which depends on how the cache is used. Because our concern is *not* communication cost, but rather the

CPU	Clock Speed (MHz)	L2 Cache Size	Cache Block Size	TLB Size	TLB Replacement Policy	Page Size
Alpha	150	256K	32	32	LRU	8192
HyperSPARC	66	256K	32	64	Random	4096
PA-RISC	33	64K	32	120	Pseudo-LRU	4096

TABLE 4.1. CPUs used to build and validate the array redistribution model.

overhead of building data into packets and collecting received data, we can run our experiments on stand-alone machines using widely-available CPUs. This lets us see the impact of a variety of architecture and implementation features on redistribution performance. The CPUs used in the experiments are shown in Table 4.1 [26, 30]. The HyperSPARC experiments were run on a single node of a Meiko CS-2 multicomputer, while the Alpha and PA-RISC machine results are from stand-alone workstations.

We emphasize that we compare our model to the actual performance of these machines, not simulations thereof. The times calculated from these machines are the means of at least 10 runs, and the coefficient of variation (that is, the ratio of standard deviation to mean) for each value is less than 0.05. The accuracy of our model is expressed in terms of the *coefficient of determination*, denoted R^2, *maximum relative error*, and *90% error*, the 90th percentile of relative error values. The coefficient of determination is a measure of how closely the model predicts the actual redistribution time. A simple model predicting a run time equal to the mean of all runs, regardless of the run parameters, would have $R^2 = 0$. A model that perfectly predicts all redistribution times would have $R^2 = 1$. A negative value of R^2 occurs when the mean is a better predictor of results than the model. The

maximum relative error indicates the worst predictions made by the model. The 90% relative error value is less sensitive to outliers than the maximum relative error.

4.3. Basic Packet-Building Model

The first component of the model we examine is the cost of copying data from the source array to the packets bound for different processors for a KYY redistribution. A sketch of the packet-building algorithm run on each processor for this redistribution is:

1. Calculate destination processor p_d of first local element
2. For each block of size ky in the local array do
3. For $i = 0$ to $k - 1$ do
   Put elements (iy) through $(i + 1)y - 1$ of the current block of size ky into the packet for processor $(p_d + i) \bmod P$
   (Note that ey bytes are moved at a time)

In the implementation of the above algorithm, the mod function in the loop is replaced by a much faster increment and test. Figure 4.1 shows the distribution of the global array data for a Cyclic(6) to Cyclic(2) redistribution and how a processor builds packets for the associated redistribution. Note that, when $k < P$, the number of packets built is k rather than P. Hence when referring to P in the discussion to follow, we assume it represents the number of processors for which packets are being built, even if that number is less than the number of physical processors involved in the redistribution.

4.3.1. Cache Considerations

The performance of packing or unpacking data can vary greatly with cache utilization. The programmer cannot rely on predictably optimal performance with-

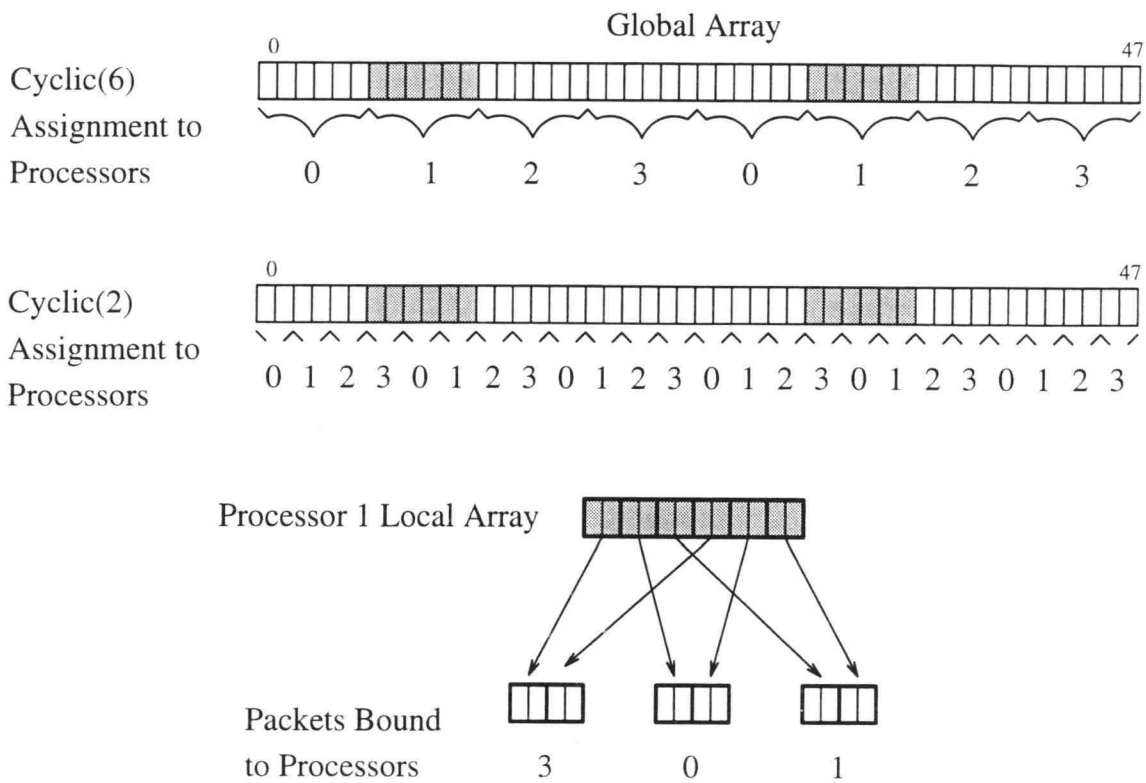


FIGURE 4.1. Cyclic(6) to Cyclic(2) (KYY) redistribution on a four-processor system. The two snapshots of the global array show to which processor each element is mapped for each distribution. When building packets for the redistribution, processor one moves sequentially through its local array, moving y elements, 2 in this example, at a time into the correct packets.

out taking into account cache effects on the building of packets. For example, a non-optimized KYY packetizing of 1 MB using the Alpha takes 194 ms; the cache-optimized version takes 100 ms—half the time. In this section we describe an easy, computationally inexpensive way to optimize cache usage for redistributions.

A wide variety of cache organizations are used in current processors, but our discussion centers on a direct-mapped, write-allocate, write-back cache with size in bytes denoted C. These features are commonly found in L2 caches. The main concern in this section is avoiding misses in the L2 cache, where a cache miss requires an expensive main memory access. This discussion can also be applied to L1 caches, whether associative or direct-mapped.

Many redistribution algorithms repeatedly move the same amount of data to all packets in a round-robin fashion. In this case, all packet pointers move in tandem as the algorithm progresses. A poor initial arrangement of packets relative to the cache will cause collisions throughout the packet building process. Therefore, our goal is to lay out the packets so the starting point of each packet does not map to the same cache block as the starting point of any other packet. Note that this is especially critical when the amount of data moved in a single copy, n, is a fraction of the cache block size b. For example, when 8 bytes at a time are moved to the packets on a system with a 32-byte cache block size, each block may be loaded into cache four times in the worst case, three times more than necessary.

Our goal is to lay out the packets so that, when data are moved to a packet, the last (i.e., current) block used by other packets is not removed from the cache. We assume a contiguous address space is available for the packets, and that the starting addresses of adjacent packets differ by a stride of σ (measured in cache blocks), where σ is at least as large as the largest packet.The following theorem

is an adaptation of a well-known technique from vector processing for eliminating memory bank conflicts during strided vector operations.

Theorem 4.1 *A stride σ meeting the following conditions ensures that the starting addresses of all packets map to different cache blocks.*

$$LCM(\sigma, C) \geq \sigma P,$$

where LCM is the least common multiple.

A simple algorithm using this result calculates the desired stride in bytes between packets:

```
1. Function CacheOptimalStride( BytesPerPacket, BlocksInCache,
                                CacheBlockSize, NumberOfPackets)
2. BlocksPerPacket = ⌈BytesPerPacket/CacheBlockSize⌉
3. While ( LCM(BlocksPerPacket,BlocksInCache) <
          BlocksPerPacket * NumberOfPackets)
4.                BlocksPerPacket = BlocksPerPacket + 1
5. Return(BlocksPerPacket * CacheBlockSize)
```

The algorithm above eliminates collisions only when elements do not span more than one cache block. To broaden its usefulness, we can logically increase CacheBlockSize to

$$2^{\lceil \log_2(\lceil n/b \rceil + 1) \rceil} b, \tag{4.1}$$

(thereby reducing BlocksInCache) to eliminate collisions for larger n values. We assume here that, as on most machines, C/b is a power of 2 and $C \bmod b = 0$. This approach can be used until the logical value of BlocksInCache becomes less than P. At this point, n is large enough that one extra cache miss per copy adds minimal time relative to the total copy cost. Although the LCM calculation requires expensive modulo arithmetic, the most expensive one on the benchmarks presented in this chapter takes less than one third of one percent of the time for the fastest

redistribution on each machine. This is a small price to pay to ensure efficient cache usage.

4.3.2. Modeling the Packet-Building Cost

The copying cost for building packets depends on several factors, which include the overhead (e.g., destination processor calculation, `memcpy` startup time, etc.) for each copy, `memcpy` implementation, memory bandwidth, and n (for the KYY redistribution ey), the number of bytes copied to each packet in a single copy operation. We represent the empirical copying cost in seconds per megabyte (MB) as $\phi(n, P)$.

When few bytes are moved, the processor calculation and `memcpy` overhead dominates, and the copy time per MB is high. As more bytes are moved in each operation, the overhead of the copy operations is minimized, and the copy time moves to its asymptotal value. Such behavior can be modeled using a function in this form:

$$\hat{\phi}(n) = \frac{a_S}{n^{b_S}} + c_S \tag{4.2}$$

In this function, the parameters a_S and c_S, respectively, represent the copy call overhead and actual time spent copying. The b_S value defines the shape of the $\hat{\phi}(n)$ curve, which depends on the relationship between a_S and c_S, the implementation of `memcpy`, and memory system design. For our model, these values are determined for a given machine by actually running three different KYY redistributions, namely when n = 8,40, and 1024 while $P = k = 8$. The first two points are used because they lie on the steep portion of the curve, where $\phi_{\text{KYY}}(n)$ changes significantly with a change in n. The last point provides the asymptotal cost when many bytes are copied at a time. The actual values of a_S, b_S, and c_S are calculated using Hooke's

CPU	a_S	b_S	c_S	R^2	Max relative error (%)	90th percentile relative error (%)
Alpha	0.579	1.44	0.067	0.961	2.48	0.98
HyperSPARC	1.58	1.32	0.066	0.989	9.15	1.23
PA-RISC	1.46	1.11	0.083	0.991	5.04	2.28

TABLE 4.2. Parameters for $\hat{\phi}(n) = a_S/(n^{b_S}) + c_S$ for three machines and the resulting R^2, MAX and 90th percentile error values relative to actual machine runs.

algorithm[1], a minimization routine effective for non-linear data. The value of b_S is greater than one because, as n is larger than a word, `memcpy` becomes more efficient. Further efficiencies can be gained with prefetching caches as n gets larger.

The function $\hat{\phi}(n)$ built from these points accurately approximates $\phi_{\text{KYY}}(n)$, as shown in Figure 4.2, where the actual HyperSPARC data and the approximation using $\hat{\phi}(n)$ are shown. Resulting values for a_S, b_S, and c_S along with relative error information for all three CPUs is shown in Table 4.2. The maximum error occurs on the steep portion of the curve when $n < 64$. This error can be reduced by including points from more experiments using small n values, but we want to limit the number of runs needed to create our model. Note that the low a_S value for the Alpha reflects a low overhead, including address calculation, for each copy. The high b_S value (the exponent in the denominator) for the Alpha reflects a $\phi_{\text{KYY}}(n)$ function that flattens more quickly than that of the HyperSPARC, while the lower b_S value of the PA-RISC machine implies that $\phi_{\text{KYY}}(n)$ approaches the asymptotal cost at a much higher value of n. Finally, the PA-RISC has a high c_S value, indicating a lower memory bandwidth than either of the other two machines.

---

[1]Available via netlib at URL `http://www.netlib.org/index.html`

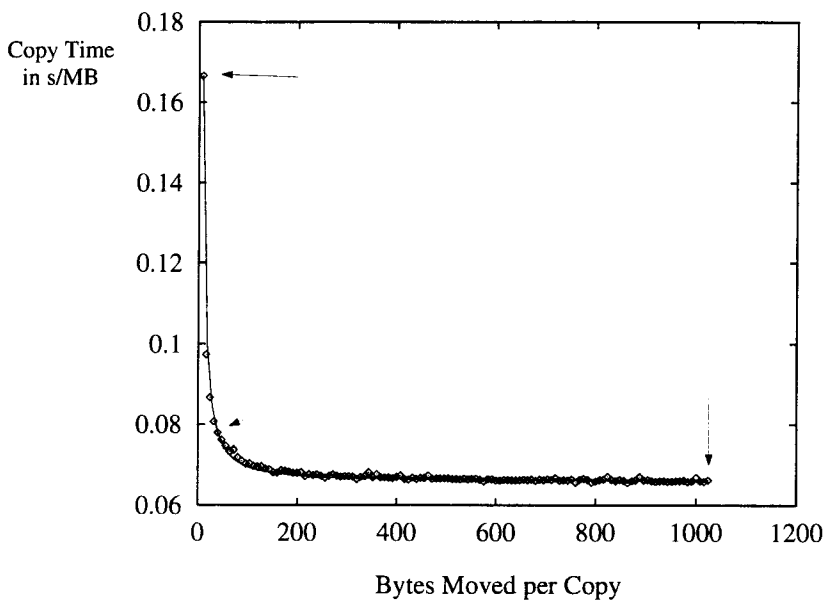


FIGURE 4.2. Packet-building copy time when $P = 8$ is shown for HyperSPARC ($\Diamond$ in the graph) as the number of bytes moved during each copy operation varies. Our model of the same data, $\hat{\phi}(n, 8)$, is generated using only data from the three runs pointed at by the arrows. The Max Error of the curve fit is 9.15%, with 90% Error of 1.23%.

4.4. Scaling the Basic Model to Larger Systems

As packets are built for more processors, the effective copy bandwidth remains heavily dependent on n, as described in the previous section. However, as more packets are built, a CPU implementation feature, namely Translation Lookaside Buffer (TLB) design, comes into play. The TLB holds Page Table Entries (PTEs) for recently-accessed pages in memory. A TLB miss typically requires at least one memory access. When multi-level page tables are employed, retrieving the desired page table entry may require several memory accesses. If packets are larger than the page size, and if the number of packets being built is greater than the TLB size, PTEs for some packets are removed from the TLB to make room for the other packets' PTEs. The removed PTEs must be read from memory, perhaps cache, when the corresponding packets are accessed again. Our studies have shown that 5 *auxiliary* TLB entries are needed to point to data aside from the packets themselves. These data include the source array, the stack, packet pointers, text and library code. Hence, although a TLB may contain a total of $T = 32$ entries, we say it has an *effective size*, denoted T_E, of 27.

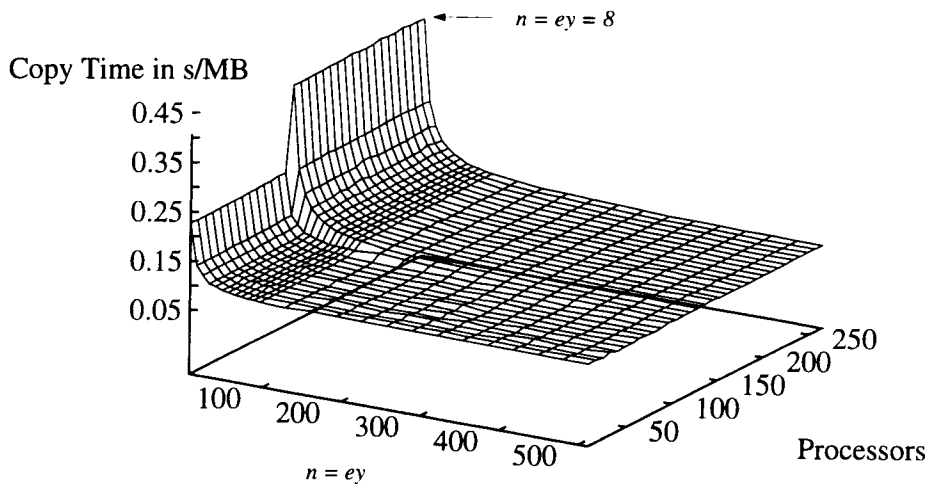
A TLB is typically implemented with either a Least Recently Used (LRU), Pseudo-LRU, or random replacement policy. Assume LRU or Pseudo-LRU is used, and that packets receive data in round-robin order (we address more complicated orderings in Section 4.4.2). When $P > T_E$, each packet will have its page table entry removed from the TLB before the packet is accessed again. The result is an abrupt increase in the packet-building cost when P moves from T_E to $T_E + 1$. The impact of TLB misses on performance is significant. Figure 4.3(a) shows that packet-building time for the PA-RISC nearly doubles as P exceeds 115, the effective size of its Pseudo-LRU TLB, when $n = 8$. The Alpha employs an LRU TLB with

$T_E = 27$. If a random removal policy is used, the copy time slowly increases as P increases above T_E until the probability that a packet's entry is in the TLB when needed is essentially zero. Figure 4.3(b) shows this trend for the HyperSPARC as P increases.

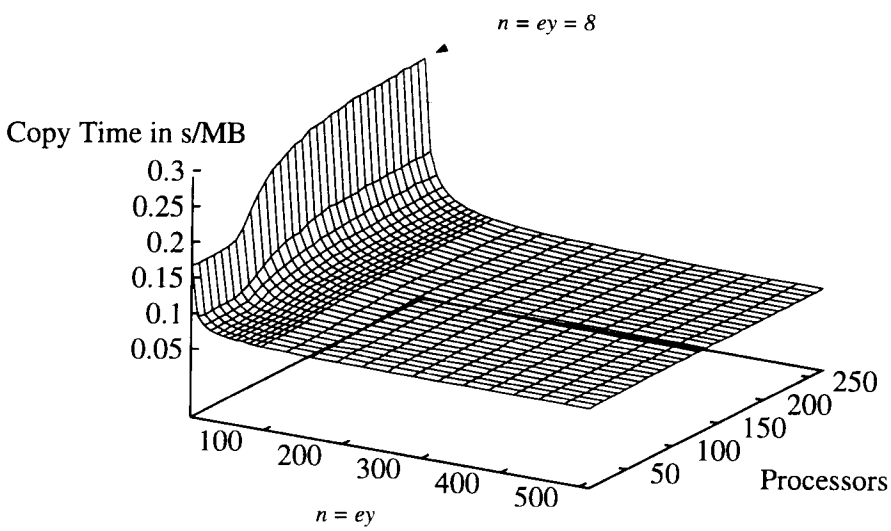
4.4.1. Calculating TLB Miss Costs

4.4.1.1. TLBs using LRU and Psuedo-LRU Replacement

We now calculate the modeled cost $\hat{T}(n, P)$ of the TLB misses. When a TLB miss occurs, PTEs must be accessed. PTEs can be cached like other data. As P increases and more PTEs are needed, more of these PTEs and packets compete for space in the cache, and more collisions occur. Therefore, we need to model the initial increase in cost as P steps above T_E as well as any increase in cost occurring as P continues to increase beyond this value. We run two benchmarks, both with $n = 8$. The first is with $P = T_E + 1$ (giving $\phi(8, T_E + 1)$), and the second is with $P = 256$ ($\phi(8, 256)$). The first of these establishes the initial jump in cost when TLB misses start occurring, and the second lets us determine any further incremental cost in TLB costs as P increases beyond $T_E + 1$. The PA-RISC TLB miss cost remains relatively constant as P increases. On the Alpha, which has an 8 KB, direct-mapped, L1 cache, page table entries needed during a TLB miss get crowded out of the L1 cache as P increases. We assume this increase in cost is linear. We determine both effects, the abrupt TLB miss cost when P increases above T_E, and the slight increase as P increases further as a difference based on three benchmark runs with $n = 8$, when $P = 8, T_E + 1$, and 256.



(a)



(b)

FIGURE 4.3. Impact of TLB misses on packet-building performance. (a) The PA-RISC uses an LRU replacement policy with $T_E = 115$. When P rises above this value, a dramatic increase in cost results, especially when n is small. (b) The HyperSPARC uses a random replacement policy with $T_E = 59$. When P rises above this value, the copy cost slowly increases as the probability of a TLB miss for each packet's page increases. Again, the impact is most noticeable when n is small.

$$\hat{T}(n,P) = \begin{cases} \frac{8}{n}\left(\phi(8,T_E+1)+(\phi(8,256)-\phi(8,T_E+1))\frac{P-(T_E+1)}{256-(T_E+1)}-\phi(8,8).\right), & P > T_E \\ 0, & \text{otherwise.} \end{cases}$$

The $8/n$ multiplier adjusts for the reduced number of TLB misses per MB copied as n increases. The remaining terms define the TLB cost model as a linear function generated from the two benchmark runs $\phi(8, T_E + 1)$ and $\phi(8, 256)$.

4.4.1.2. TLBs using Random Replacement

For a TLB using random replacement, the increase in copy time is more gradual as P increases. Note from Figure 4.3(b) that the increase starts even before P reaches T_E. This is due to the fact that compulsory TLB misses at the start of packet-building may force replacement of PTEs pointing to active packets' pages, even if stale PTEs reside in the TLB. We do not model the slight slope when $P < T_E$. Instead, as with the LRU model, we model the less subtle change that occurs when P exceeds T_E. We assume that, when $P = T_E$, the probability that a given packet's page table entry is in the TLB (its hit ratio) is 1.

The steady-state probability $h(P)$ of a TLB hit occurring is defined by a recurrence:

$$h(P) = \left(\frac{T-1}{T}\right)^{(1-h(P))(P-1)} . \tag{4.3}$$

The intuition for this formula is provided by a "packet-centric" view. A packet's TLB entry has a $(T-1)/T$ chance of staying in the TLB when a miss occurs. The exponent is the number of expected misses between subsequent times a packet is accessed: the number of other packets times the steady-state miss rate $(1 - h(P))$. Although this formula ignores the higher rate of pre-steady-state misses, it gives a remarkably accurate prediction of TLB miss costs.

This recurrence can be reduced to a function of $W(h(P))$, where $W(h(P))e^{W(h(P))} = h(P)$, which can be solved numerically [35]. We use Maple to repeatedly solve the recurrence to generate a table of $(P, \hat{h}(P))$ values to be referenced by the model. Finally, we use Maple to calculate $\hat{h}(256)$. Because we use $\phi(8, 256)$ to determine the cost of TLB misses, division by $(1 - \hat{h}(256))$ provides the proper scaling of probabilities to TLB miss cost. As with the LRU formula, we must also adjust the impact of the TLB cost based on n. The results presented here are found when $n = 8$, where the cost of TLB misses per bytes moved is great; the impact of TLB misses is inversely proportional to n.

$$\hat{T}(n, P) = \begin{cases} \dfrac{8\,(\phi(8,256) - \phi(8,8))}{n\left(1 - \hat{h}(256)\right)}(1 - \hat{h}(P)), & P > T_E \\ 0, & \text{otherwise.} \end{cases}$$

After adding this value to $\hat{\phi}(n)$, we have an accurate model of the packet-building cost in s/MB for the HyperSPARC. To evaluate the model, we compare it to actual runs with $n = 8$, where TLB misses dominate, and accuracy is critical. The comparison is shown in Figure 4.4. The model has a R^2 value of 0.996, the maxmum error of the fit is 3.0%, and the 90th percentile error is 2.5%.

4.4.2. Impact of k on TLB Misses

In the previous section we noted that when $k < P$, the number of processors for which packets are built is k. The results presented there used P to represent the *effective* number of processors, or $\min(k, P)$ when $k \leq P$. From the KYY algorithm on page 68, when $k > P$, the P packets are accessed in round-robin order until k of them have been accessed. The minimum number of times *all* P packets are accessed during these k accesses is $\lfloor k/P \rfloor$, and we call this the number of *full iterations*

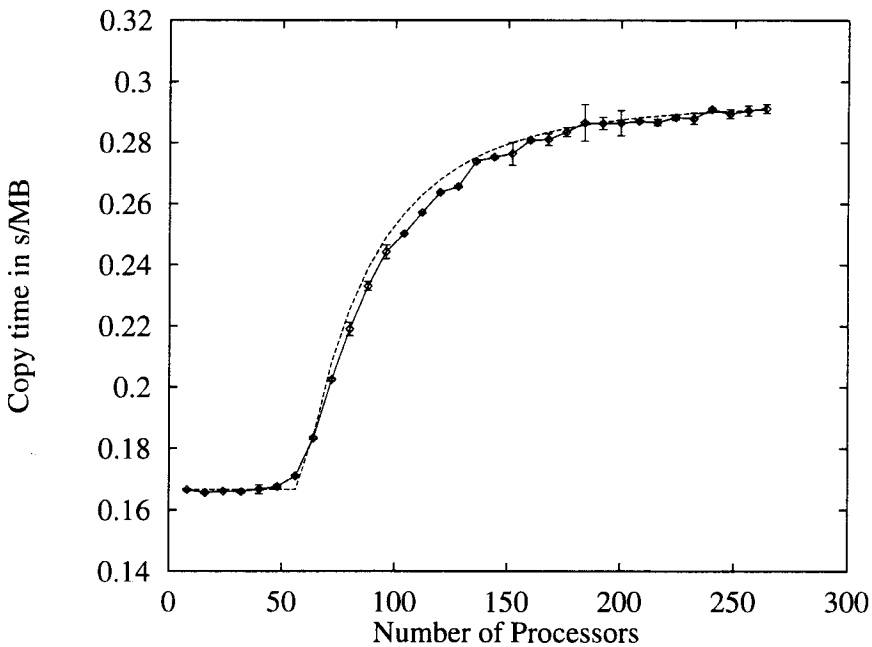


FIGURE 4.4. Comparison of our model (dashed lines) for a random TLB with actual results from a HyperSPARC (solid lines). The model curve is built from the steady-state probability plus benchmark runs when $P = 8$ and $P = 256$. The HyperSPARC curve shows the copy cost calculated from actual runs and includes 95% confidence intervals.

through the P packets. The remaining accesses, $k \bmod P$, of them, occur in a single *partial iteration.*

When $k > P$ and $P > T_E$, the partial iteration through the packets can change the impact of TLB misses described in the earlier model. The impact on TLBs with random replacement is minimal, so we do not vary the model presented in the previous section. The results are shown in Table 4.3.

4.4.2.1. Large k with True LRU Replacement

We now assume the TLB implements true LRU replacement. The partial iteration accesses $k \bmod P$ packets. If $(k \bmod P) \leq T_E$, PTEs for these packets remain in the TLB and result in TLB hits during the first full iteration. When the last $P - T_E$ packets are accessed during the first full iteration, the first packets' TLB entries are removed and must be reloaded for the next iteration, whether it is full or partial. Because the TLB hits occur only during the first full iteration, the biggest impact occurs when $\lfloor k/P \rfloor$ is 1 and $k \bmod P$ is large, because nearly half of the memory accesses can result in TLB hits. If many full iterations take place, the number of TLB hits is small relative to the number of misses. We model this behavior as follows:

$$\hat{\mathcal{T}}_k(n, P) = \begin{cases} \frac{k - (k \bmod P)}{k} \hat{\mathcal{T}}(n, P), & 0 < k \bmod P \leq T_E \\ \hat{\mathcal{T}}(n, P), & \text{otherwise.} \end{cases}$$

Table 4.3 shows this model's approximation of the Alpha's performance when $k = 64 = 2T$ and P ranges from 32 to 64 with $n = 8$. These experimental values are used because they show a range in which the TLB effects have the greatest impact, when the number of full iterations is one.

4.4.2.2. Large k with Pseudo-LRU Replacement

Unfortunately, packet-building performance with CPUs using common pseudo-LRU implementations depends on which TLB entries are used. A typical implementation [87] marks a TLB entry as VALID when it is accessed. When a TLB miss occurs, the page table entry from that memory access is added to the TLB; the entry it replaces is the lowest-numbered entry marked INVALID. If no entries are marked INVALID, all unlocked entries are reset to INVALID and the lowest-numbered entry marked INVALID (at this point, entry 0 unless it has been locked by the operating system) is replaced. We model the PA-RISC TLB performance in this manner [43].

Figure 4.5 shows how the PA-RISC TLB's state changes during a KYY redistribution with $k = 240$ and $P = 128$. Assuming that the 0th packet is put into TLB entry 0 at the start of packet-building[2], the first 115 packets cause misses and take up all of the available TLB entries. When a new packet is accessed, space must be made in the TLB for its PTE. All entries are currently VALID, so they are subsequently marked INVALID—the result is that the new PTE goes into TLB entry 0. PTEs for subsequent packets follow, and at the conclusion of adding data to 128 different packets, the TLB looks as shown in Figure 4.5(a). The partial loop builds only 112 packets starting at packet 0, whose PTE is not in the TLB. Hence, 112 misses occur, and the TLB state after this partial loop is shown in Figure 4.5(b). We now begin a full loop adding data to all 128 packets. Entries for packets 0 through 111 are in the TLB, so they are marked VALID (c). After packets 112-114 are accessed, all entries are VALID, so since there is nowhere to place packet 115, all TLB

---

[2]This may vary depending on the current state of a program starting a redistribution. However, the programmer has no control over the TLB.

TLB VALID Bit	1	0
Packet #	115 - 127	13 - 114

(a)

TLB VALID Bit	1	0	0
Packet #	102-111		0 - 101

(b)

TLB VALID Bit	1	0	1
Packet #	102-111		0 - 101

(c)

TLB VALID Bit	1	1
Packet #	102-114	0 - 101

(d)

TLB VALID Bit	0	0
Packet #	102-114	0 - 101

(e)

TLB VALID Bit	1	0
Packet #	115-127	0 - 101

(f)

FIGURE 4.5. Sequence of TLB states for a pseudo-LRU TLB as packets are built when $P = 128$ and $k = 240$.

entries are marked INVALID (e), leading to PTEs for packets 115-127 being added to the TLB starting at entry 0. From this point on, PTEs for packets 102-114 trade with PTEs for packets 115-127 in the first 13 positions of the TLB, never forcing removal of entries 0-101.

Theorem 4.2 *Assuming the first packet's PTE goes into TLB position 0, and that subsequent packets' PTEs follow in sequence, with*

1. $P > T_E$

2. $k \bmod P <= T_E$

3. $k <= 2P$

the PTEs for packets 0 through $2T_E - P - 1$ *are not removed from the TLB after the first partial iteration (i.e., after k packets have been accessed) during the packet-building phase of a KYY redistribution if and only if* $k > 2T_E$.

Proof. The conditions in Theorem 4.2 guarantee that there is one full iteration and one partial iteration per k packet accesses. After the first full iteration, TLB entries 0 through $P - T_E - 1$ are VALID with data for packets $2T_E - P$ through $P - 1$ (in the manner of Figure 4.5(a)). In the subsequent partial iteration, packet 0's PTE goes into TLB entry $P - T_E$, and subsequent packets' PTEs follow in sequence. We consider two cases:

i) $k \leq 2T_E$ (only if case). In this case, the number of packets accessed in the partial iteration, $k - P$, does not require more TLB entries than those marked INVALID during the wraparound in the middle of the previous (full) iteration. Hence, the next iteration (another full iteration) will fill the remaining $2T_E - k$ INVALID TLB entries. At this point, all entries are VALID, so all are marked INVALID and TLB entries are filled starting at 0. In this full iteration, $P - 2T_E + k$ packets remain to fill TLB entries from 0, thus forcing removal of packet 0 at position $P - T_E$.

ii) $k > 2T_E$ (if case). The number of packets accessed in the partial iteration, $k - P$, is greater than the number of INVALID TLB entries. Once these are filled, all TLB entries are marked INVALID, and $k - 2T_E$ packets' PTEs are

inserted starting at TLB entry 0. Note that packet 0 at TLB entry $P - T_E$ is not removed here because $k \leq T_E$. During subsequent full iterations, the following occurs in order:

(a) PTEs for packets 0 through $2T_E - P - 1$ at TLB entries $P - T_E - 1$ through $T_E - 1$ are used and marked VALID.

(b) PTEs for packets $2T_E - P$ through $k - P - 1$ are reused at TLB entries 0 through $k - 2T_E - 1$ and marked VALID.

(c) INVALID PTEs from $k - 2T_E$ through $P - T_E - 1$ in the TLB are loaded with PTEs for packets $k - P$ through $T_E - 1$ and marked VALID, at which point all PTEs are VALID.

(d) The next packet accessed requires a new PTE, so all PTEs are marked INVALID. Entries for packets T_E through $P - 1$ fill TLB entries 0 through $P - T_E - 1$. The TLB entry containing packet 0 is not removed. Subsequent alternation between partial and full iterations result in identical behavior—the partial iteration results in $k - 2T_E$ TLB misses starting at 0, and the full iteration uses these entries before generating misses. The misses do not overwrite packet 0's TLB entry, because it is marked VALID at the start of an iteration and the Pseudo-LRU algorithm overwrites a newly INVALID TLB entry 0 before writing over a newly INVALID entry elsewhere in the TLB. ■

Theorem 4.2 applies to those redistributions with small k values. Its basic ideas can be expanded to generate an algorithmic model for $k >= 2P$, in which many of the subtractions must be replaced by the mod operator. We present here only the model for TLB miss costs for a Pseudo-LRU TLB with $k < 2P$, noting that the steady-state fraction of TLB misses per k accesses is $2(P - T_E)$.

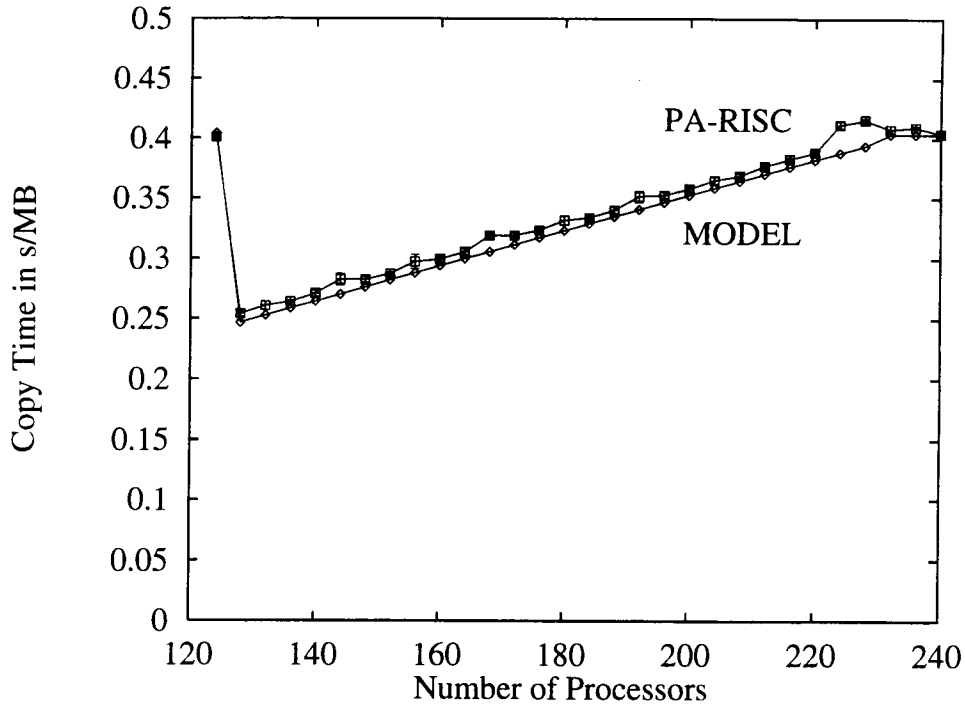


FIGURE 4.6. Impact of k on packet-building time as P changes. Our model is compared to actual results for the PA-RISC, which has an effective TLB size (T_E) of 115. k is 240, and $n = 8$. The maximum relative error of the model is 5.9%, while the 90th percentile relative error is 4.2%. 95% confidence intervals for actual data are shown.

$$\hat{T}_k(n,P) = \begin{cases} \hat{T}(n,P)(2(P-T_E)/k), & T_E + P \bmod T_E >= P \\ \hat{T}(n,P), & \text{otherwise.} \end{cases}$$

Figure 4.6 shows this model's approximation of the PA-RISC performance when $k = 240 = 2T$ and P ranges from 124 to 240 with $n = ky = 8$. These values are used because they show a range in which $\lfloor k/P \rfloor$ is 1 and $k \bmod P$ varies, where the TLB effects have the greatest impact. Table 4.3 summarizes the comparisons of our model to actual runs on all three machines with three different TLB replacement policies.

CPU	k	First P	Last P	Step	R^2	Max Error	90% Error
Alpha	64	36	64	1	85.9	4.7	1.8
HyperSPARC	128	56	128	4	91.8	5.9	5.8
PA-RISC	240	124	240	4	96.8	5.7	4.2

TABLE 4.3. Parameters and results for experiments determining impact when $P > T_E$ and $k \bmod P \neq 0$. For each CPU the range and stepsize of P values is given along with k.

4.4.3. Impact of Page Size

Assume packets are laid out end-to-end in memory. As the number of packets increases, or as L decreases, the size of individual packets decreases. The separation between the packets may be less than the page size Ψ. In this case, one TLB entry may point to more than one packet, resulting in fewer TLB misses during the packet-building phase. Our model does not incorporate this refinement, because we assume moderate-sized machines (hence few packets) and large arrays are used.

Another unmodeled impact of small page size appears when systems have LRU or Pseudo-LRU TLBs and small page sizes. The PA-RISC, with a 4096 byte page size, is an example of such a system. As n increases, the proportion of `memcpy` operations that move data to two different pages also increases. When a copy moves data to two pages, the copy uses up two TLB entries. Assume all packets get the same amount of data during each iteration (i.e, $k \leq P$). At the start of the iteration in which two pages are accessed for each packet, the LRU nature of the TLB implies that $T_E - P$ entries are stale. The first $T_E - P$ packets will hit their current TLB entries and replace stale entries. The next packet will hit its current TLB entry and erase the entry for the next packet. The remaining packets will encounter two TLB misses each. During the next iteration, when each packet again accesses a single

page, all packets will encounter a TLB miss. The total number of hits during these two iterations is $T_E - P + 1$. Clearly, the number of hits is minimized when P nears T_E. On the PA-RISC, the deviation from our model when $P = 112$ (recall that $T_E = 115$) is greater than 10% when $112 \leq n \leq 188$. Figure 4.3(a) shows that the increase in cost due to TLB misses moves from $T_E + 1$ to a smaller P value as n increases.

4.5. Other Redistributions

The previous discussion focused on the KYY redistribution. We have shown that, using only four or five sample runs and TLB parameters, we can generate an accurate model for the KYY redistribution under a variety of conditions. In this section, we show that a single additional run is needed to transfer our KYY model to each of several other redistributions. The redistributions we include in our model here are Cyclic(x) to Cyclic(kx), denoted XKX, Block to Cyclic, denoted BCY, and the more general Cyclic(x) to Cyclic(y), denoted GEN. The additional run used to transfer the model is performed with $P < T_E$ to avoid TLB effects. The number of bytes moved in a single copy, n, is 8. At this point, the differences in time taken for the repeated calculations are most apparent, and the resulting model is more accurate. For a given redistribution $\mathcal{R}$, the difference in time between this sample run for $\mathcal{R}$ and the one for KYY is denoted $\Delta^{\mathcal{R}}_{\text{KYY}}$.

We assume, with the exception of the GEN redistribution, that the TLB costs that arise as P increases are independent of the computation performed in the loop. Hence we can use the same TLB costs for XKX and BCY that we used for KYY, with one of two XKX options using the TLB model for $k > P$ described in Section 4.4.2. We simply need to generate the approximation $\hat{\phi}_{\mathcal{R}}$. We do this by

noting that the difference in costs between two redistributions lies in the overhead, not in copy costs. The overhead per byte copied diminishes as n increases, so we adjust for the overhead difference as follows:

$$\hat{\phi}_{\mathcal{R}}(n, P) = \hat{\phi}(n) + \frac{8\Delta^{\mathcal{R}}_{\text{KYY}}}{n}$$

In the paragraphs to follow, we briefly describe each of the redistributions and their performance. The results of our model relative to actual machine runs are shown in Table 4.4. The comparisons between our model and actual runs are divided into two parts. The first comparison is for $8 \leq n \leq 128$, the steep part of the curve where differences between the model and empirical runs are most pronounced. The second is for the entire range of our experiments, $8 \leq n \leq 1024$, which includes the more easily-modeled steady-state behavior. We separate out the first comparison to give a 90th percentile error value not watered down by many of the easily-modeled points.

The XKX redistribution is performed using two different algorithms depending on the relationship between k and P. The first algorithm, denoted XKX-1, consists of nested loops like the KYY redistribution. XKX-1 must use the TLB model described in Section 4.4.2, since its loop structure is identical to that of KYY. Each iteration requires an addition and a mod operation, performed using a conditional, to select the destination processor. The second algorithm, XKX-2, consists of a single loop in which the expensive mod operation is replaced by two conditional subtractions and two additions per iteration. For the XKX redistributions, $n = ex$.

The BCY redistribution is performed using a single loop containing an increment and test to select the destination processor. It should run slightly faster than the KYY algorithm for the same value of n.

The GEN, or Cyclic(x) to Cyclic(y) algorithm requires several division operations in the form presented in [90]. The formula for the destination processor for the ith data element on a processor p is given by

$$(((i \bmod x) + (P(i/x) + p)x)/y) \bmod P$$

The inner mod and division operations can be replaced using more efficient additions, subtractions and tests, with two divisions remaining. The best known general redistribution algorithms also require two divisions [10]. These operations make GEN much more expensive than the previously discussed redistributions. In addition to being expensive operations, they must be performed once per element, since $n = e$ with this redistribution. Another difference between GEN and the other redistributions is that we do not account for TLB misses. That is, $\hat{\mathcal{T}}(n, P) = 0$. We can do this because the TLB misses are overlapped with the expensive division and mod operations, and hence they do not add to the cost of the redistribution. The PA-RISC and HyperSPARC have consistent cost in s/MB regardless of P. Figure 4.7 points out an interesting characteristic of the Alpha system while showing that TLB misses do not play a role in the GEN redistribution. The Alpha's integer division routine (implemented in software) checks the divisor for a power of two. If a power of two is found, an inexpensive shift is performed in place of the division. This optimization has a significant impact on the redistribution cost when P is a power of two. Our model takes this optimization into account by performing an extra run when P is one away from a power of two, calculating the number of division operations performed, and generating the cost per division.

Finally, the Cyclic to Block redistribution differs from those above in that it moves only large, contiguous chunks of data at a time. These can be sent from in-place in the local array; packets are not built. Unpacking is performed after the

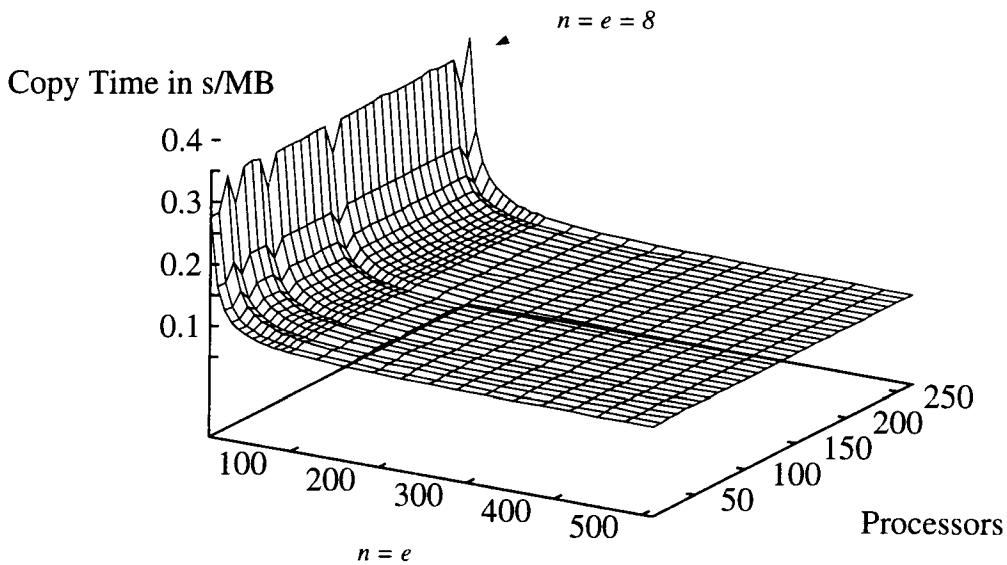


FIGURE 4.7. The GEN (Cyclic(x) to Cyclic(y)) redistribution performed on the Alpha. Two features of the graph should be noted. First, when P is a power of 2, the division operation is replaced by a more efficient shift, and a significant performance gain results. Second, no TLB miss effects appear; the curve remains flat, the power of 2 values of P aside, as P increases.

Redist	Architecture	$8 \leq n \leq 128$			$8 \leq n \leq 1024$		
		R^2	Max Error (%)	90th %-ile Error (%)	R^2	Max Error (%)	90th %-ile Error (%)
KYY	Alpha	0.989	6.6	3.1	0.981	6.6	1.7
	HyperSPARC	0.994	12.2	4.2	0.994	12.2	2.8
	PA-RISC	0.997	11.7	4.0	0.995	12.3	3.2
BCY	Alpha	0.980	9.3	3.5	0.969	9.3	1.7
	HyperSPARC	0.993	11.8	5.1	0.992	11.8	3.1
	PA-RISC	0.998	11.4	3.9	0.995	12.5	3.2
XKX-1	Alpha	0.977	9.8	4.8	0.973	9.8	1.8
	HyperSPARC	0.994	14.4	4.4	0.994	14.4	2.7
	PA-RISC	0.997	13.6	3.9	0.995	13.6	3.0
XKX-2	Alpha	0.979	12.4	5.0	0.970	12.4	2.5
	HyperSPARC	0.993	11.0	5.1	0.992	11.0	3.3
	PA-RISC	0.995	13.1	5.7	0.993	13.1	4.0
GEN	Alpha	0.997	8.4	4.5	0.981	9.8	6.9
	HyperSPARC	0.994	13.0	4.7	0.992	13.0	5.1
	PA-RISC	1.000	4.8	2.5	0.998	7.7	6.4

TABLE 4.4. The accuracy of our model is shown for all redistributions for two configurations. In all cases, both P varies from 8 to 264 by 8. In the first configuration, n varies from 8 to 128 by 8. This region is where the data changes most drastically with n. For the second configuration, n varies from 8 to 1024 by 8.

exchange of messages. We describe the cost of unpacking data in the next part of this chapter.

4.6. Packet-Building Summary

We have identified and accurately modeled the factors affecting the cost of packet-building for redistributions. The steps taken to build the model include

1. Determine $\hat{\phi}(n)$, the model for copy time as the number of bytes per copy changes. This can be done once per redistribution or, as we did here, doing

it once and extrapolating to other redistributions using a single run for each new redistribution.

2. Determine TLB miss cost $\hat{\mathcal{T}}(n, P)$ when $k \leq P$, based on either LRU/Pseudo-LRU or random TLB replacement schemes.

3. Determine TLB miss cost for all k values, $\hat{\mathcal{T}}_k(n, P)$, again depending on TLB replacement scheme.

With these in hand, we have a complete model of the packet-building costs for a redistribution:

$$\hat{\pi}_{\mathcal{R}}(n, P) = \hat{\phi}(n) + \hat{\mathcal{T}}_k(n, P). \tag{4.4}$$

In the next sections we build, in a significantly different manner, a model for the unpacking of data.

4.7. Introduction to Unpacking of Received Messages

Once all packets have been built, the compute nodes perform an all-to-all personalized communication to deliver packets to their destination nodes. It is beyond the scope of this chapter to model the details of this communication, which varies from machine to machine. We focus in subsequent sections on the time needed to unpack received packets into their proper layout in memory.

The algorithms in [90] [91] contain two methods for unpacking data. The first is denoted *synchronous*. With the synchronous approach, each processor receives all incoming messages before unpacking the data. The local array is then built in the following manner:

1. For i = 0 To Size(`LocalArray`)
2. Calculate source processor of `LocalArray`[i]
3. Copy data from source processor's packet to `LocalArray`[i]

Note that the size of elements copied depends on the redistribution performed. This operation is the dual of the packet-building operation described earlier; here data are copied *from* packets to the local array. Otherwise, this routine uses memory, cache and the TLB in the same fashion as the packet building described in Sections 4.3 through 4.6. Therefore, the model developed earlier can be used to model synchronous unpacking as well.

A second method for the unpacking operation uses an *asynchronous* approach, in which packets are unpacked as soon as they are received:

1. For i = 1 TO number of packets to receive
2. Receive packet from some processor
3. Calculate Location in `LocalArray` of first element in packet
4. Place subsequent elements in packet in `LocalArray` with stride S

Note that only one address calculation is required per packet, so expensive division and mod operations are not repeated during the strided copy.

Although special instances of some redistributions rule out use of the asynchronous approach, it can be used in many cases, and it is typically faster than the synchronous approach [90] [91]. When available, it is the method of choice for receiving and unpacking data. Our analysis of these steps focuses on the asynchronous approach. This analysis here can also be applied to *packet-building* schemes relying on strided copies of data.

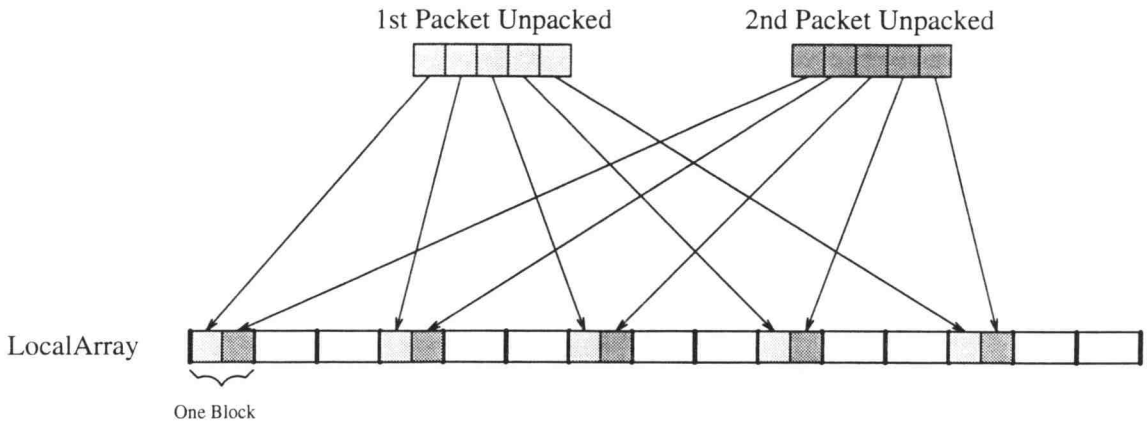


FIGURE 4.8. Example of efficient cache usage during unpacking of data. Memory blocks are brought into cache during the unpacking of the first packet. Elements from the next packet map to array locations sharing the same blocks, already in cache.

4.8. Model Elements

When performing the packet-building operation, the programmer controls the memory locations of the packets. Hence, cache collisions can be minimized; the current block for each packet remains in the cache from one use to the next. With the unpacking operation, cache efficiency is dependent on both the data and the order in which packets are processed. Cache efficiency impacts performance the most when small elements are copied. In the worst case, a block of LocalArray will be loaded into the cache j times, if n is b/j. In the best case, each block is loaded into cache only once. Figure 4.8 shows how properly ordering the packets for processing can ensure that cache blocks are reused.

Unfortunately, while the programmer may be able to control the order in which packets are processed, efficient cache use cannot be guaranteed. The stride of the data, over which the programmer has no control, may cause cache collisions. Figure 4.9 shows how the array blocks of later elements in a packet may collide with

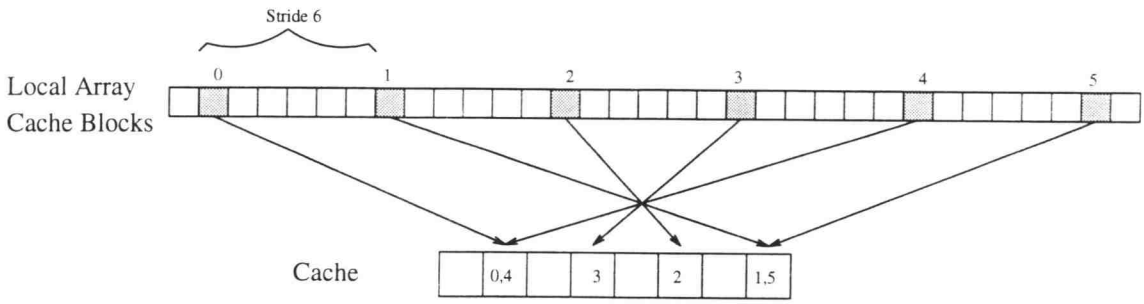


FIGURE 4.9. Example of redistribution-dependent cache collisions caused during unpacking of a single packet. The stride of the packet data maps later elements' array blocks to the same cache location as earlier elements' blocks. As subsequent packets are processed, the CPU is forced to reload the cache blocks used by the early elements, 0 and 1, since their blocks were forced from the cache when elements 4 and 5 were copied.

array blocks of earlier elements from the same packet. When this happens, array blocks to which the first elements of neighboring packets map are not in cache, and every array block must be reloaded into cache each time it is referenced.

Because the roles of TLB and cache misses in unpacking data are different than in packing data, we must build the model $\hat{v}$, for the total unpacking time, in a different way. Here we briefly outline the steps taken in the following sections to build this model.

1. Determine the cost θ of a TLB miss in the context of unpacking.

2. Determine the cost κ of a cache miss in the context of unpacking.

3. Determine the cache-efficient copying rate $\hat{u}_E(n)$ in s/MB for unpacking data.

Given these three values, we use a combination of analytical models and algorithmic techniques to determine the total cost of unpacking data for a given redistribution.

4.8.1. Isolating Cache and TLB Miss Costs

In this section we build the foundations for our strided unpack model. The model, while resulting in graphs similar in shape to the (n, P) graphs for packetizing, must be built in a different fashion. We assume P is the number of compute nodes from which packets are received, and hence may be smaller than the total number of compute nodes. The stride with which elements received from another node are placed in LocalArray is implicitly nP. Since we expect to be redistributing large arrays, our benchmarks build local arrays of size at least ΨT_E to include TLB effects. All of our experiments use a local array size of 1 MB.

4.8.1.1. Modeling TLB Effects

The TLB effect which occurs during packet-building and described in the first part of this chapter results from the simultaneous building of P packets. In the asynchronous receive, only one packet is processed at a time. However, striding through the local array once requires accessing

$$\begin{array}{ll} \frac{L}{\Psi}, & nP \leq \Psi \\ \frac{L}{nP}\left\lceil\frac{n+1}{\Psi}\right\rceil\frac{n \bmod \Psi}{\Psi}, & \text{otherwise} \end{array}$$

different pages. The multiplier on the second of these terms reflects the frequency with which multiple pages are accessed during a single copy. This multiplier is at least 1, and it increases with n. If the number of TLB misses when unpacking data from a single packet is greater than T_E, then repeated TLB misses will occur. For CPUs with LRU or pseudo-LRU TLB replacement policies, each new local array page access results in a TLB miss. For CPUs with a random TLB replacement policy, the miss ratio increases with the number of pages in the local array. We

assume the local array is large enough that the miss ratio is essentially 1. For example, with the page size of 4096 bytes and a 1 MB file, 256 pages are accessed; in this case the miss ratio, from our earlier model, is (1 - $\hat{h}(256)$) = 0.981.

When nP is less than the page size Ψ, every page in the local array is accessed[3] for each packet. Hence, the TLB cost in this case is $(LP\theta)/\Psi$, where θ is the cost for a *single TLB miss* in the context of the unpacking operation. As nP increases beyond Ψ, a page may occasionally be skipped. In these cases, we model the TLB cost as $(LP\theta)/(nP)$. Finally, if nP increases so that $L/(nP) < T_E$, the TLB cost is L/Ψ, *assuming packets are received so that neighboring elements are copied from subsequent packets.* If this condition is not met, we use the previous formula. Note that it is possible to receive packets so that some packets are neighbors and others are not. It is beyond the scope of this chapter to model this behavior. We can summarize the time for the total TLB miss cost for the unpacking operation as follows:

$$\Upsilon = \begin{cases} \frac{LP\theta}{\Psi}, & nP <= \Psi \\ \frac{L\theta}{n}\left\lceil\frac{n+1}{\Psi}\right\rceil\frac{n \bmod \Psi}{\Psi}, & \Psi < nP < \frac{L}{T_E} \\ \frac{L}{\Psi}, & \text{otherwise} \end{cases} \tag{4.5}$$

For our model, we must empirically determine the cost of a single TLB miss, θ, as well as the cost of a cache miss, κ, in the context of the unpacking operation. We can determine the TLB miss cost without interference from cache misses by running benchmarks when $n = 32$, the cache block size b of all the machines used in this study. With $n = 32$, each block is loaded into the cache exactly

[3]We ignore the case where the starting page or ending page, both of which may be only a fraction of a page size, are not accessed once per packet.

once, assuming alignment of the local array to a block boundary[4], and efficiency is optimal. Therefore, we can benchmark $v(32, p_1)$ and $v(32, p_2)$, the actual run times for unpacking when $n = 32$ with both p_1 and p_2 processors, calculate the number of page references for each, and determine the cost of each TLB miss. We choose $p_1 = 8$ as the foundation for our model, but p_2 must be selected carefully.

Although any p_2 such that $p_2 \neq 8$ and $p_2 < (L)/(\Psi n)$ will let us calculate the cost of each TLB miss, careful selection of p_2 allows us to calculate the cost of each cache miss as well as the TLB miss cost. By benchmarking $v(8, 8)$ with a sufficiently large L, we are guaranteed to have a cache miss on every access of the local array when using a power-of-two-sized cache. We want p_2 such that $v(p_2, 8)$ has no (L2) cache collisions. As we discuss later, such a value is dependent on the array size. For a 1 MB local array, $v(8, 124)$ results in no cache collisions on any of our three machines and has the same TLB cost as $v(32, 124)$. Although several choices for P result in no collisions, we want a high value just below 128. The long distance between data points ensures greater accuracy of our model, and keeping the value less than 128 ensures a straight-line model of TLB effects when $\Psi = 4096$ and $n = 32$ (parameters for both the PA-RISC and hyperSPARC machines). That is, we needn't use more than one formula in Equation 4.5. We can extrapolate the cache-efficient time

$$\tilde{v}_E(8, 8) = v(8, 124) - v(32, 124) + v(32, 8) \tag{4.6}$$

and use it to calculate the cost of a cache miss in the context of the strided unpacking:

$$\kappa = \frac{v(8, 8) - \tilde{v}_E(8, 8)}{((b - 8)/b)(1048576/8)}. \tag{4.7}$$

---

[4]We force this alignment.

CPU	a_G	b_G	c_G	κ	θ
Alpha	0.072	0.711	0.0662	0.96 μs	0.061 μs
HyperSPARC	3.04	1.674	0.0661	0.76 μs	0.50 μs
PA-RISC	2.51	1.379	0.0837	0.61 μs	1.3 μs

TABLE 4.5. Parameters for the cache-efficient curve fit and the cost of an L2 cache miss and a TLB miss for each CPU.

The numerator is the time difference between a cache-inefficient unpack and a cache-efficient one. The denominator gives the number of excess misses caused by the cache-inefficiency. As described in the next section, one cache miss per block is compulsory.

Next the curve-fitting technique used for packet-building is also used to generate a baseline cache-efficient s/MB curve for unpacking:

$$\hat{u}_E(n) = \frac{a_G}{n^{b_G}} + c_G$$

We want this rate to be independent of TLB effects, so we adjust the three points $\tilde{v}_E(8,8), v(32,8)$, and $v(1024,8)$ by subtracting $(8)(1048576)\theta/\Psi$ before using Hooke's algorithm to generate a baseline cache-efficient s/MB curve. Table 4.5 shows the curve fit parameters and miss times for the Alpha, HyperSPARC, and PA-RISC processors. These differ significantly from the values in Table 4.2 due to the absence of even compulsory TLB misses in their formulation.

4.8.1.2. Modeling Cache-Inefficient Unpacking

Cache-inefficient unpacking can occur due to self-interference, when a packet's later elements collide with its earlier elements in the local array, and due to

poor ordering of packets, when local array blocks brought into cache by one packet are not utilized by a neighboring packet before being bumped from the cache.

Theorem 4.3 *When cache-inefficient gathering occurs, the number of cache block misses exceeds the optimal number in a sequence of $LCM(n,b)/b$ cache-aligned array blocks by*

$$\begin{array}{ll} LCM(n,b)/n - 1, & n \bmod b \neq 0 \\ 0, & n \bmod b = 0 \end{array}$$

where LCM is the least common multiple function.

Proof. The second case is trivial; each block is accessed the minimum number of times, once. For the first case see Figure 4.10. The LCM condition ensures that only the leftmost write is aligned on the left with a block. Each subsequent write ($\mathrm{LCM}(n,b)/n - 1$ of them) is not aligned with a block; at least one other write accesses its first block further to the left. Therefore, assuming the leftmost write to each block is compulsory, the writes accessing the same block further to the right are in excess of the optimal number. ■

From Theorem 4.3, the total number of excess cache misses during cache-inefficient unpacking in which $n \bmod b \neq 0$ is

$$\frac{L}{\mathrm{LCM}(n,b)}\left(\frac{\mathrm{LCM}(n,b)}{n} - 1\right). \tag{4.8}$$

Figure 4.11(a), actual results from the PA-RISC, along with our model in Figure 4.11(b), illustrates the way cache misses drastically change the underlying smooth copy model. The other two architectures show similar characteristics, but their lower TLB costs result in a smaller cost increase as P increases. Analysis of our cache-inefficient model is shown in Table 4.6. We induced cache inefficiency in the actual runs by processing packets from processors in an order that guaranteed that

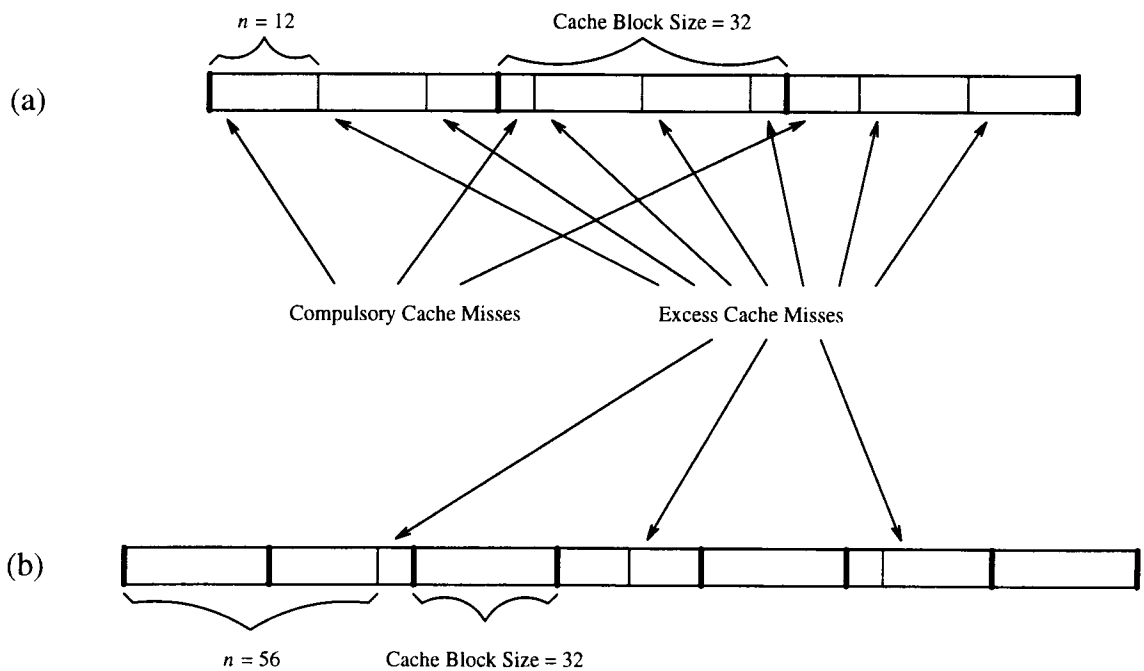


FIGURE 4.10. Determining the number of excess cache block accesses. (a) $n = 12, b = 32$, and the number of excess misses is LCM(12,32) $/12 - 1 = 7$. (b) $n = 56, b = 32$, and the number of excess misses is LCM(56,32) $/56 - 1 = 3$.

CPU	R^2	Max Error (%)	90% Error (%)
Alpha	0.98	14.0	8.1
HyperSPARC	0.99	10.2	5.0
PA-RISC	0.99	9.8	6.1

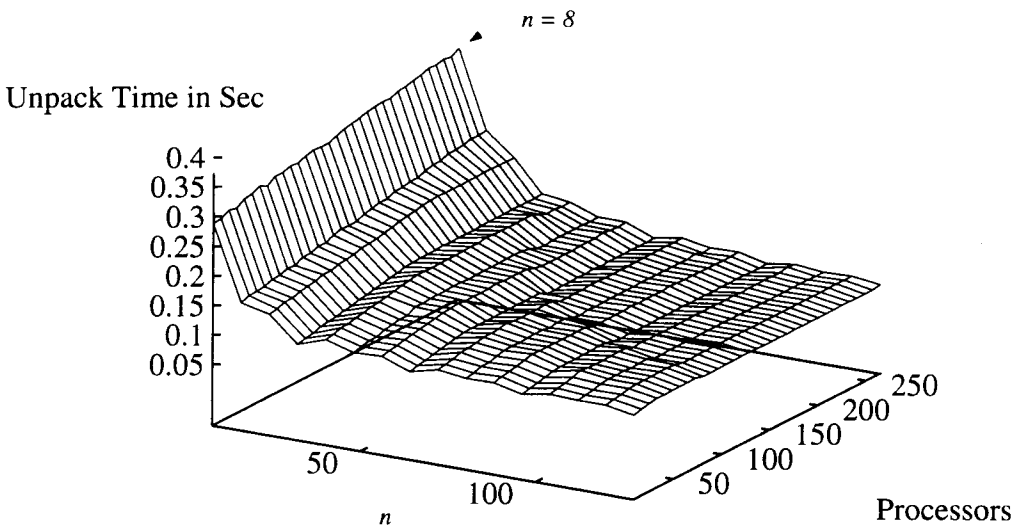
TABLE 4.6. Results of our model of cache-inefficient unpacking compared to actual runs on three machines.

blocks loaded into cache were removed before reuse. This approach also ensures that TLB PTEs are not reused, and increases in TLB costs were also modeled.

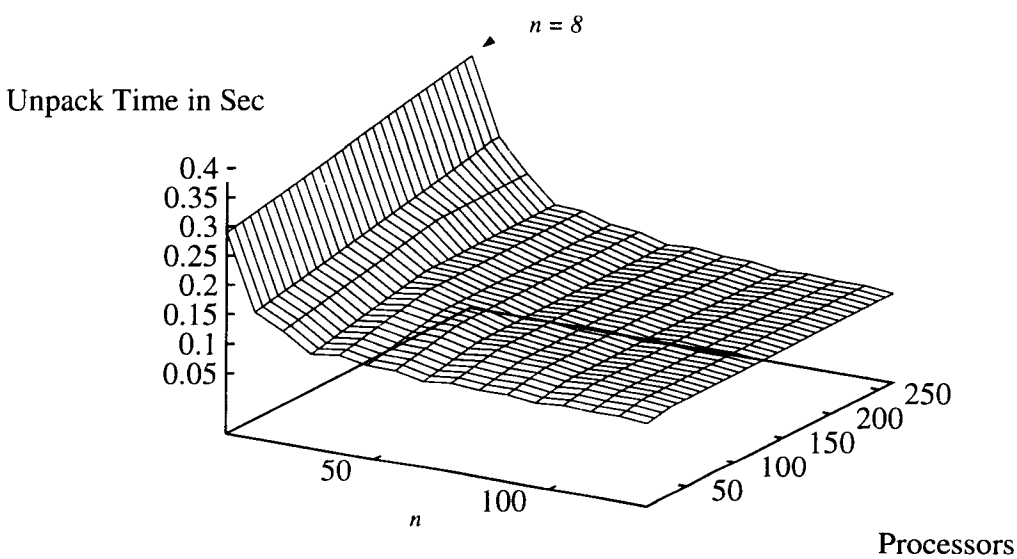
4.8.1.3. Modeling Cache-Efficient Unpacking

As illustrated in Figure 4.9, the stride dictated by the data can often force poor cache use, so that cache misses occur on every local array access. However, sometimes the data are more cooperative, and no cache collisions occur. This is more likely with a large cache size C. Another possibility is that some fraction of local array accesses result in cache misses. In this section we present the results of our model with cache-efficient unpacking and discuss the methods used to determine the cost when a fraction of accesses cause collisions.

Figure 4.12 shows the widely varying unpacking times on all three machines when $n = 8$ and $L = 1$ MB as P varies from 4 to 264 in increments of 4. The cache-efficient and cache-inefficient models derived earlier are shown bounding the times. The figure supports the intuitive notion that a larger cache minimizes cache interference. The PA-RISC's smaller cache forces many more cache-inefficient unpacking sequences than the HyperSPARC or the Alpha. Note that, although it has the same size L2 cache as the Alpha, the HyperSPARC has more values between the



(a)



(b)

FIGURE 4.11. Cache-inefficient unpacking of 1 MB using the PA-RISC. (a) shows the actual machine runs, while (b) shows our model. The stair-step effect is caused by changes in cache efficiency as n changes. The steep increase in cost as P increases is due to repeated TLB misses. The abrupt reduction in slope along a fixed n value as P increases is due to a TLB miss cost decrease. The TLB miss cost decreases as the stride increases beyond the page size of 4092, and this occurs at a lower value of P as n increases. All values have a coefficient of variation less than 0.05.

CPU	R^2	Max Error (%)	90% Error (%)
Alpha	-0.44	5.5	3.1
HyperSPARC	0.70	3.7	2.5
PA-RISC	0.87	3.7	2.7

TABLE 4.7. Results of our model of cache-efficient unpacking compared to actual runs on three machines when $n = 8$. Only those P values which guarantee no L2 cache collisions are used in this analysis. The R^2 value for the Alpha represents the fact that our model is no better model than the mean. This is expected, because the Alpha cache-efficient curve is flat. Similarly, the other R^2 values are relatively low, an expected phenomenon when modeling a nearly flat straight line.

bounding lines; this is due to cache block prefetching done by the HyperSPARC, which increases the number of cache collisions in some cases. Table 4.7 quantifies the accuracy of the cache-efficient model for the cache-efficient unpacking when $n = 8$, where the model's accuracy is most severely tested. The R^2 value for the Alpha represents the fact that our model is no better model than the mean. This is expected, because the Alpha cache-efficient curve is flat. Similarly, the other R^2 values are relatively low, an expected phenomenon when modeling a nearly horizontal straight line with slope less than 0.0003.

The points residing between the cache-efficient and cache-inefficient curves experience varying numbers of cache misses. We do not know of a closed form expression to describe the self-interference caused by the strided accesses to the local array. We instead present a brief discussion and example to outline an algorithmic approach to modeling this behavior.

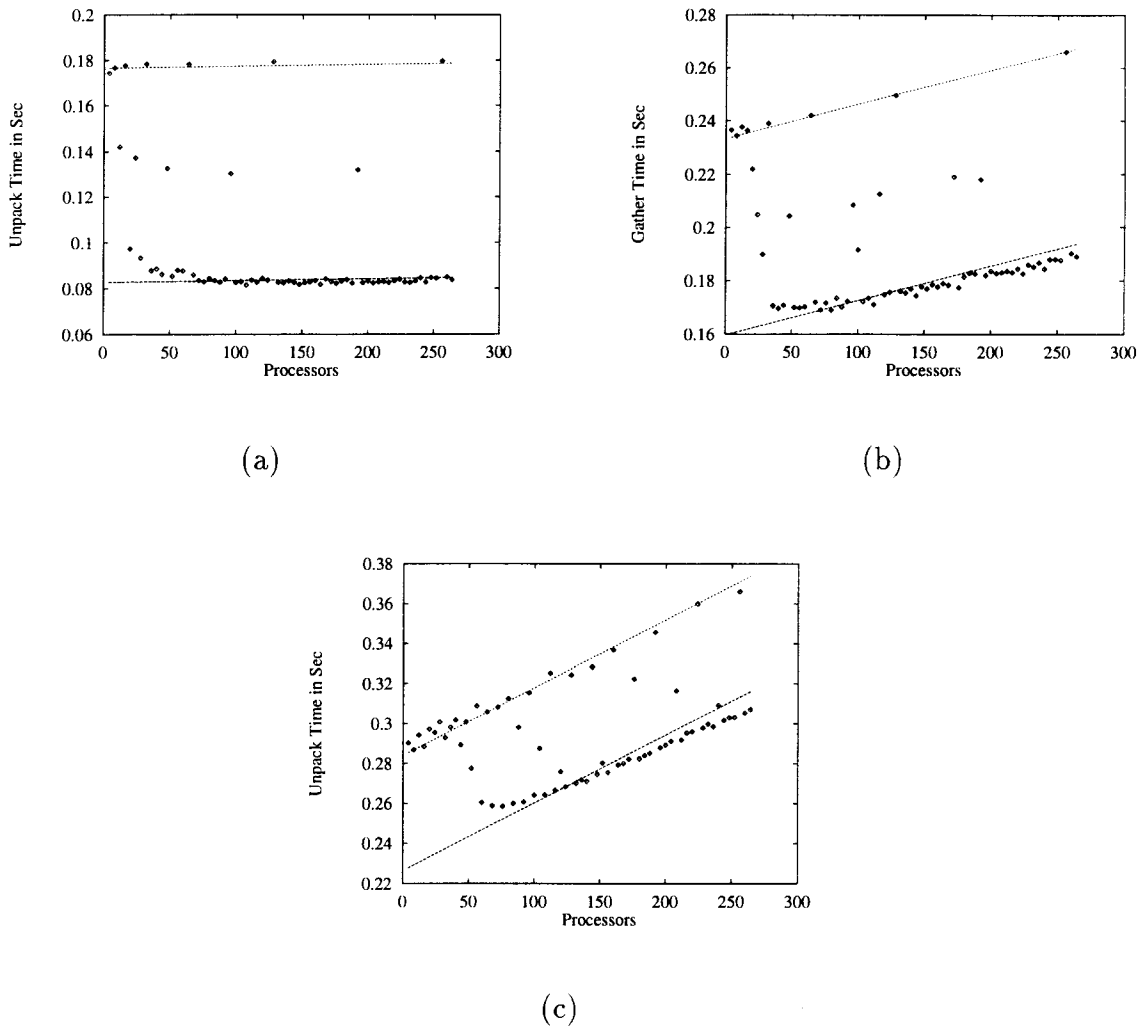


FIGURE 4.12. Unpacking time when $n = 8$ and L =1 MB as P varies from 4 to 264 in increments of 4. Bounding each plot are the cache-efficient and cache-inefficient model derived earlier in the chapter. (a) The Alpha, which has a minimal TLB miss cost and, hence, a relatively flat curve. (b) The HyperSPARC. (c) The PA-RISC, with the highest TLB miss cost and greatest slope.

Definition 4.1 A *potential collision* occurs when placing elements into the local array with stride S and at least one byte of a copied element is within $(b-1) \bmod C$ bytes of an element placed earlier in the same strided copy. ■

Definition 4.2 The *target element* in a potential collision is the one placed in the local array earlier. ■

Definition 4.3 The *colliding element* in a potential collision is the one placed in the local array later. ■

Theorem 4.4 *When placing elements into the local array with stride $S \gg b$ and $S > n$, the target element of the first potential collision is η_0, the first element placed during the strided copy.*

Proof. Any potential collision, with colliding element η_j and target element η_i, occurs when $-b \bmod C < (j-i)S \bmod C < n + b \bmod C$. This is independent of j and i, so the earliest potential collision has $i = 0$. ■

Theorem 4.4 tells us that we can work from the first element to find the first potential collision. This collision occurs when JS, $J > 0$, is near a multiple of C. We look for potential collisions using the following algorithm, which is limited to the special case where $n = 8$ and $b = 32$. The algorithm assumes the first element is aligned with its cache block:

```
For i = 1 TO ⌊L/C⌋
    leftapproach = ⌊(iC)/S⌋ S
    leftdistance = iC−leftapproach
    rightapproach = S− leftdistance
    If leftdistance < 32
        Potential Collision on left-hand-side
    If rightapproach < 32
        Potential Collision on right-hand-side
```

Note that if `leftdistance`= 0, the collision is a direct hit, and every potential collision is, in fact, a collision. We follow a more interesting example using the Alpha, which has a cache of 256 KB. For this example, $P = 26$ and $n = 8$, so $S = 208$. As shown in Figure 4.13(a), when $i = 3$, $3781S$ mod 262144 results in a potential collision, with `rightapproach`= 16. Figure 4.13(b) depicts the four possible ways these elements can be targeted to the same or neighboring cache blocks. Note that collisions occur in only half the configurations, when the first element falls in one of the first two of locations in the cache block. Figure 4.13(c) shows how four segments, each of size C, of the local array map to the cache. The elements copied into SEG2 and SEG3 map to unused cache blocks; no collisions occur until data are copied into SEG4, whose elements map to (potentially) the same cache blocks as those in SEG1. Since the collisions come 3/4 of the way through the local array, only the first 1/4 of the array, SEG1, experiences collisions with elements from the last 1/4, SEG4. Processing of the next packet, assuming it starts at the position after the packet shown, will have to reload blocks for SEG1. When SEG4 is reached, those blocks must be reloaded. Blocks in SEG2 and SEG3, making up 1/2 of the array, remain in cache throughout, so 1/2 of the time 1/2 of the accesses result in cache misses. Therefore, 1/4 of the copies, 32768 in all, result in excess cache misses. On the Alpha, the cache miss cost is 0.96 μs, so we add 0.031 to the modeled cache-efficient value of 0.0828 to give 0.1138, which is within 4% of the empirical value of 0.1186.

Although this example is specific and relies on favorable alignment of data within cache blocks, it presents a framework in which a more general model could be built. The details of such a model, which must take into account multiple collisions, cache prefetching effects and all possible n and S values, are beyond the scope of this chapter.

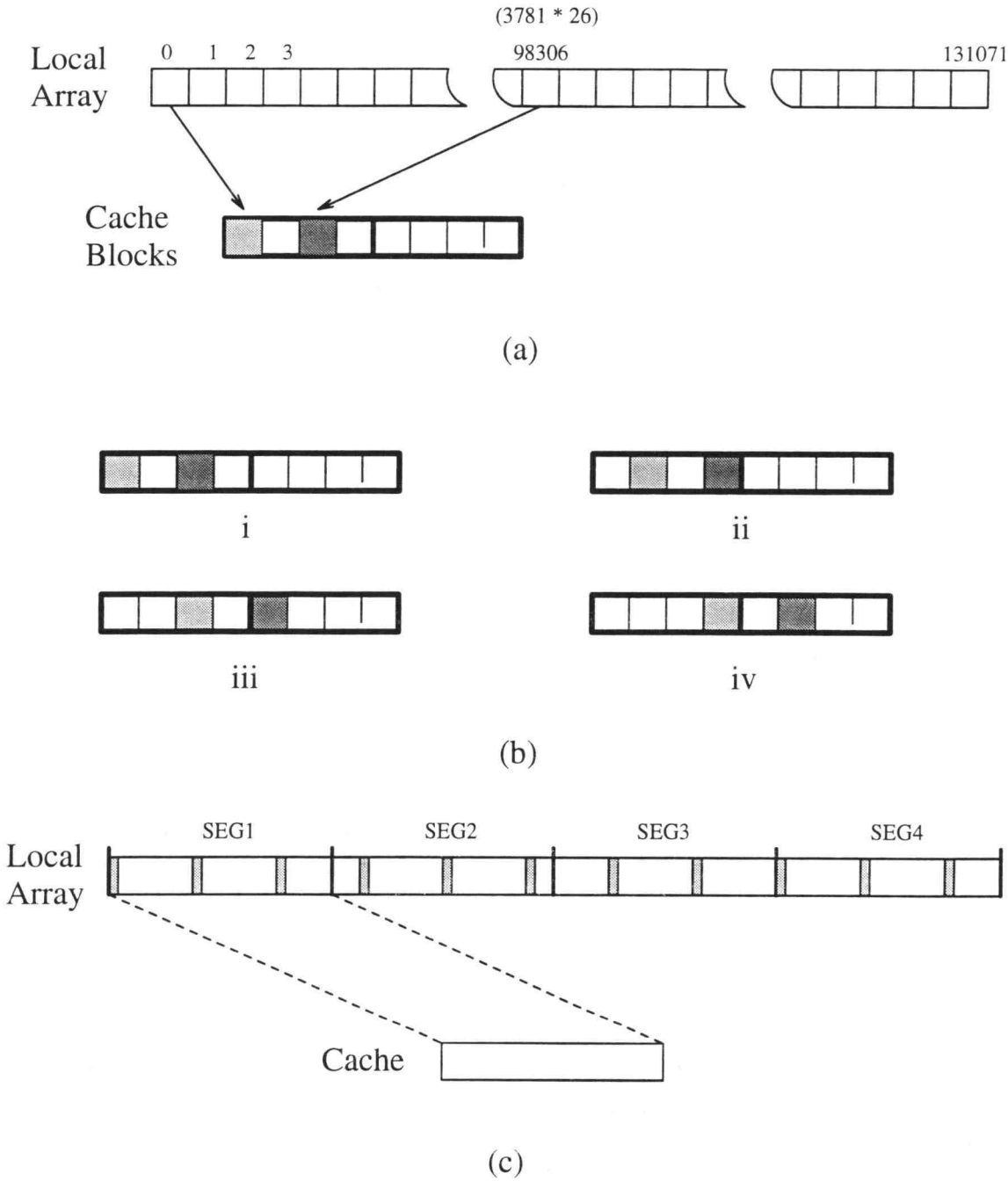


FIGURE 4.13. (a) Collision resulting when the 3,781st element is copied into the local array. This element maps to the same cache block that the first element copied used. (b) Four possible ways the potential collision can map to the cache, depending on the position within the cache block of the first element copied. Only (i) and (ii) result in collisions. (c) As the local array is filled with stride S, four logical sweeps of the cache take place, since the array is four times the size of the cache. Collisions are avoided during the first three sweeps because of the stride. The last sweep, when SEG4 is filled, results in potential collisions with blocks used by SEG1.

4.9. Discussion

The results presented in this chapter provide many insights into the costs of a redistribution. The dominant factor for both packing and unpacking data is the granularity of data copied— the more data copied at a time, the lower the cost per byte. TLB costs come into play for the packing step only when the number of packets approaches the number of TLB entries, and the impact is significant only for fine-grained copies. The careful programmer can avoid cache collisions when packing data. Hence, the model for the packing step is relatively simple, and we can use a fixed rate in s/MB for our model.

The unpacking step is more complicated. TLB effects come into play for non-trivial-sized arrays regardless of the number of packets, and they are dependent on the array size and stride of received elements. Cache effects are difficult to model, and they too are dependent on the array size. Unpacking is modeled, therefore, using a fixed copy rate in s/MB plus TLB and cache costs. Both cache and TLB effects have the greatest impact when the copy granularity is small.

Our results show that, even for a GEN redistribution, the copy overhead per byte quickly drops to near the optimal machine value as the granularity increases. Therefore, performing a general Cyclic(x) to Cyclic(y) redistribution as the composition of a Cyclic(x) to Cyclic(LCM(x, y)) (XKX) redistribution followed by a Cyclic(LCM(x, y)) to Cyclic(y) (KYY) redistribution is cost-effective only when n is small. Figure 4.14, in which the cost of a general redistribution performed in one expensive phase is compared to the cost of the redistribution performed as the composition of two simpler redistributions, demonstrates this fact. The results are modeled on the Meiko CS-2, which achieves 20 MB/s message bandwidth during the complete exchange of data, and whose HyperSPARC processor's packet-building

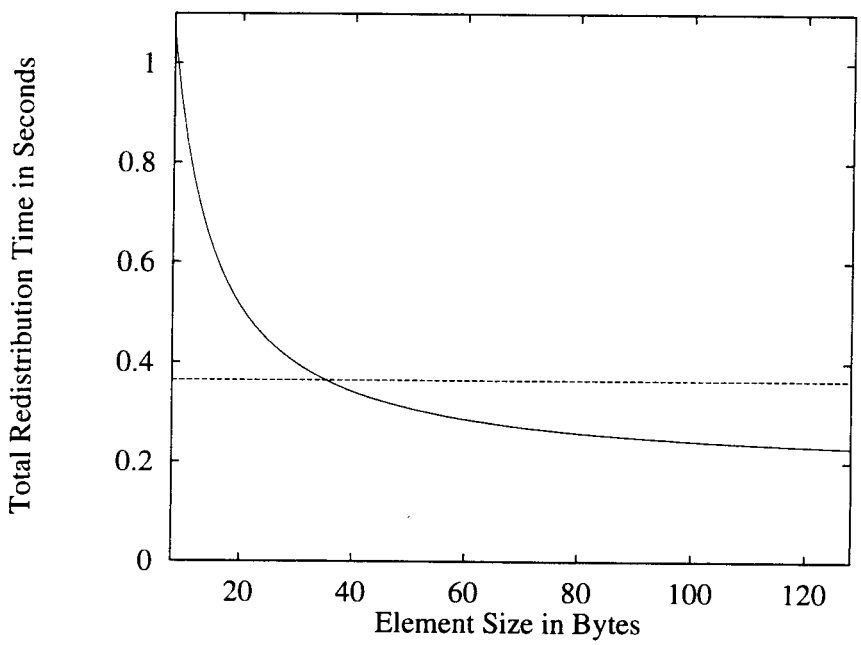


FIGURE 4.14. The cost of a general redistribution performed in one expensive phase is compared to the cost of the redistribution performed as the composition of two simpler redistributions. The constant line represents the composition of two simpler redistributions whose packing and unpacking steps are achieved at the maximum memory bandwidth and, hence, take constant time regardless of element size. The curve which decreases as the element size increases is the cost for a general, one-phase redistribution. The data are modeled using Meiko CS-2 characteristerics.

characteristics have been examined in detail in this chapter. Each processor has a local array of 1 MB, and we assume the number of processors is small enough that TLB effects do not come into play. In the figure, the cost of the one-phase redistribution decreases as the element size increases. Recall that, for the simpler redistributions composing the two-phase redistribution, several elements are copied to and from packets together. The exact number varies from redistribution to redistribution. In Figure 4.14, we assume elements are moved in large units, so the HyperSPARC steady-state copy rate of 0.066 MB/s is achieved. Since data must be packed twice and unpacked twice for the two-phase redistribution, the packet-building cost is 0.264 seconds. The message-passing costs for two complete exchanges add 0.1 seconds to the time for the two-phase redistribution. Despite the optimistic assumptions for the two-phase redistribution, it outperforms the general, one-phase redistribution only when the element size is less than about 36 bytes.

The results from our strided unpacking model, namely that the redistribution performed—not the programmer—dictates cache efficiency, has implications for the packet-building phase of the redistribution. That is, redistribution schemes that propose to build packets by striding through a local array cannot count on the most efficient use of the cache. Therefore, while such schemes may require less computation [78] than the more straightforward algorithms employed here, the advantage may be lost due to poor use of cache.

4.10. Conclusions and Future Work

This work has shown that the cost of a redistribution is dependent on many CPU implementation features as well as the redistribution performed—its data granularity, synchronicity, and computation overhead. Because they can greatly impact

the redistribution time, these CPU features, including TLB size, TLB replacement algorithm, cache size, and cache block size must be taken into account when redistribution schemes are developed. Furthermore, if the programmer cannot avoid inefficient use of cache when using a given scheme, benchmarks must be performed in both cache-inefficient and cache-efficient configurations, and the frequency with which both occur should be analyzed.

5. ENHANCING DISK-DIRECTED I/O FOR FINE-GRAINED REDISTRIBUTION OF FILE DATA

5.1. Introduction

In contrast to early parallel computers, many current multicomputers provide users with reasonable disk bandwidth by striping data across several I/O nodes, each of which may operate a high-speed disk array. Unfortunately, high disk bandwidth is necessary, but not sufficient, to support common parallel file operations. The operating system must translate available disk bandwidth into user bandwidth, but in many sytems the user sees only a fraction of the peak bandwidth available. One reason for this situation is that the distribution of parallel data among the processors is often different than the layout of data in the file. For example, if P processors use a Cyclic data distribution and array data are laid out in the file in Block fashion, each processor must perform a fine-grained file operation to read or write every Pth element in the file. These fine-grained operations significantly reduce the effective file system bandwidth.

More recent approaches to redistributing file data rely on the notion of *collective I/O operations*, which require that all compute nodes synchronize before the file system is contacted. The file operation to take place, typically a transfer of a logically contiguous file segment, is then viewed from a global point of view rather than from the local view of the individual processors accessing their own portions of the file segment. A file-system-independent approach to collective I/O is the Two-Phase Access Strategy [8, 27], in which file data are accessed using the disk distribution. Ideally, the disk distribution is independent of the configuration of the

compute nodes. Therefore, the typical disk distribution is Block rather than Cyclic. When the Two-Phase Access Strategy is used, the compute nodes redistribute the data from the compute node distribution (e.g., Cyclic) to the disk distribution before a write, or they redistribute data from the disk distribution to the compute node distribution after a read. These redistributions ensure that for both reads and writes, each compute node accesses a large, logically contiguous portion of the file.

Kotz has proposed Disk-Directed I/O (DDIO) [52], a unique approach to parallel file systems relying on collective operations. The central idea behind DDIO is that the I/O nodes, given the extra information provided by a collective interface, can direct the distribution of data from I/O nodes to compute nodes when reading, or vice versa when writing. When DDIO is used, each I/O node first sorts the disk blocks to be accessed during the course of the entire collective operation[1]. Sorting is done by location on disk, not logical location in the file. The sorting step significantly reduces the total disk access time. Because the blocks are not necessarily accessed in logical file order, the I/O nodes must direct the movement of data to and from the compute nodes. DDIO relies on a double-buffering scheme, with each buffer consisting of a disk block. While one block is read from or written to disk, each I/O node computes the destination or source processor and address for each record in the just-read (when reading) or next-to-be-written (when writing) block. The processors and addresses vary depending on the distribution of data among the compute nodes. Data are striped across the I/O nodes in canonical row-major (i.e., with a one-dimensional Block distribution or two-dimensional Block-None distribution) order. Low-latency `memput` and `memget` operations are used by the I/O nodes to direct movement of data between compute nodes and I/O nodes.

---

[1]Throughout this paper, as in [52], we assume a block (alternatively, stripe) size of 8 KB.

As shown in [52], Disk-Directed I/O provides high user bandwidth due to the overlapping of disk operations and communication. We say the double-buffering scheme achieves *optimal overlap* when the message-passing required for one block of data takes less time than the overlapping disk operation for the second block of data. In other words, the redistribution of data adds no time to the required disk operations, except the transfer of the first (when writing) or last (when reading) block, which is not masked by a disk operation. When large records are read or written, or when many small logically contiguous records are distributed to the same compute node, few **memput** or **memget** operations are needed to move the 8 KB of data in a block, and DDIO achieves optimal overlap.

In other cases, Disk-Directed I/O performance is not optimal, achieving only a fraction of the bandwidth provided by the disk system (but far outperforming non-collective approaches) when 1) the record size is small; 2) the distribution of data on the compute nodes requires that no two records which are logically contiguous in the file reside on the same compute node; and 3) disk bandwidth is high, due either to the use of multiple disks, or to a layout of data on disk (e.g., contiguous) supporting fast access. A compute-node distribution meeting conditions (1) and (2) requires a fine-grained redistribution. Distributions forcing condition (2) include the Cyclic one-dimensional distribution and the Block-Cyclic and Cyclic-Cyclic two-dimensional distributions. These Cyclic distributions efficiently support adaptive and irregular data-parallel computations, and traces of production parallel file operations show that such distributions are used frequently and make up a substantial percentage of file accesses [56, 77].

In this chapter we analyze DDIO's less than optimal performance for fine-grained file distributions. We examine alternative approaches to the message-passing-oriented DDIO. One set of alternatives is based on building packets of data

Parameter	Meaning
P	Number of compute nodes
I	Number of I/O nodes
B	I/O node block size in bytes
$\mathcal{M}$	Number of disk blocks combined in messages for MB-DDIO
D_1	Time to seek, rotate, and transfer first block
D_B	Time to read a single disk block *after* head positioned
λ	Message startup latency
β	Time per byte to send data over the network
β_x	Time per byte on loaded network (for complete exchange)
N	Number of array elements to read/write
e	Size of array elements in bytes
$\hat{v}_{mb}(e_v, n_v)$	Time per byte to unpack MB-DDIO data on compute nodes. The number of bytes moved at a time is e_v ; n_v, the amount of data unpacked after a message-passing step by a compute node, is used in the next chapter to determine if the cost of repeated TLB misses is added
$\hat{\pi}_{\text{BC}}(e, P)$	Time per byte to build P packets e bytes at a time including cache and TLB effects (BC $\equiv$ Block-to-Cyclic)
$\hat{\pi}^{IO}_{BC}(e)$	Time per byte for an I/O node to build packets e bytes at a time

TABLE 5.1. Parameters used in our model.

to reduce message-passing overhead. Another alternative examined is Two-Phase Disk-Directed I/O, which combines DDIO with the Two-Phase Access Strategy [8, 27]. We provide models for the effective bandwidths of DDIO, DDIO using packets, and Two-Phase DDIO. We validate the models on a real parallel computer and use them to compare different approaches with different machine configurations. Table 5.1 shows parameters of our models, which are described in detail in Section 5.4.1.

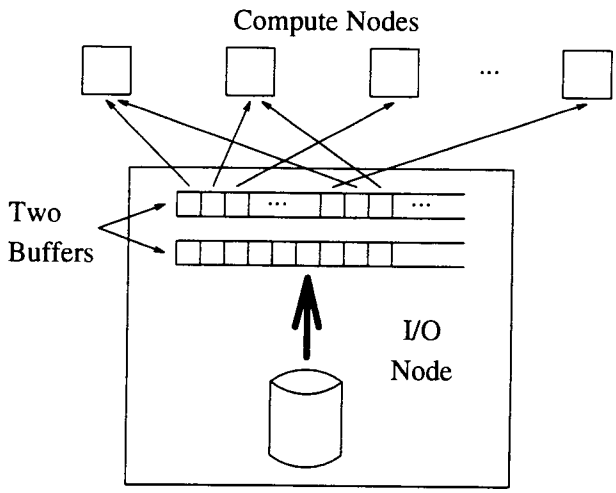


FIGURE 5.1. Disk-Directed read of data distributed in Cyclic fashion on the compute nodes. The I/O node calculates the destination processor and address for each record in the block just read from disk. Each record is transmitted to the destination compute node which writes the data at the location specified by the I/O node. The transmission of data overlaps the reading of data into the second buffer. The roles of the buffers are reversed while the next disk block is read.

5.2. Disk-Directed I/O with the Cyclic Distribution

In this section we analyze DDIO's performance when redistributing data to or from a Cyclic compute-node distribution with a record size of 8 bytes. The Two-Dimensional Block-Cyclic and Cyclic-Cyclic distributions have more expensive destination-address computations than the Cyclic distribution, so their performance using DDIO is slightly worse than Cyclic's [52]. As shown in Figure 5.1, records from the file for a Cyclic compute node distribution are assigned to compute nodes in round-robin fashion. Using DDIO for a read, each element is transmitted *individually* to its destination. For a write, each I/O node requests from the desired compute node the single record needed at that time. We first examine data laid out on disk in a random layout, in which DDIO achieves optimal overlap [52].

5.2.1. Random Disk Layout

In the original DDIO paper [52], results with both random and contiguous layouts of data on disk are shown. With a random layout of blocks[2] sorted by starting sector number, the validated simulation of the HP 97560 disk drive [57] used in [52] requires a mean access time of 18.8 ms with a standard deviation of 4.5 ms. With these parameters, one can ensure optimal overlap 99% of the time when the time needed to perform all **memput** or **memget** operations for a block is less than 8.3 ms. Using DDIO, this requires that address calculation and message-passing overhead for 8-byte records be less than 8.1 μs *per record.* Although the simulated machine used in [52] meets this requirement [53], many existing parallel machines do not have such low message-passing latency.

One obvious way to avoid the high cost of 1024 messages per block is to build packets of data (i.e., *packetize* data) containing all records bound to or from each compute node in a given block. This approach is essentially a scatter/gather, which Kotz mentions as a way to achieve higher bandwidth on fine-grained redistributions [52]. For a read, one packet for each compute node is built on each of the I/O nodes, which send one message to each processor per block. For a write, each I/O node requests from each compute node only the data the I/O node needs in the next block it writes; the I/O nodes unpack each compute node's data in a strided fashion to form the disk block to be written. We call this approach the *packet-based variation* of DDIO and denote it PB-DDIO.

The cost for both building and unpacking packets was studied in great detail in Chapter 4. It was shown there that, as the number of packets built increases

---

[2]We assume the 16 individual 512-byte sectors making up a block are contiguous on the disk.

beyond the number of Translation Lookaside Buffer (TLB) entries, assuming packets at least a page in size, all TLB entries are exhausted, and each packet access requires an expensive memory access to reload the TLB with the correct page table entry. In this scenario, the TLB miss costs become significant. When unpacking, the TLB miss costs increase as the local array size increases. Since the data in packets on I/O nodes totals 8 KB, and a few TLB entries can point to all packets, no TLB effects occur at the I/O nodes for either packing or unpacking data. We have found that the average time for the HyperSPARC processor of the Meiko CS-2 to build packets for 8 KB of data for a Cyclic compute node distribution is 1.20 ms. When this time is used in conjunction with the 99% optimal overlap time from the previous paragraph (8.3 ms), the overhead for each message can be up to $7.1/P$ ms in order to maintain the desired overlap with disk operations. On a 128-processor machine the per-processor overhead is 55 μs; for 16 processors it is 444 μs. These numbers allow machines with higher message-passing latency than that modeled in [52] to achieve optimal overlap using PB-DDIO when data have a random layout on disk. However, while PB-DDIO is a big improvement over DDIO for fine-grained redistributions, this approach has its limitations, as we see when we examine higher-performance disk systems.

5.2.2. Contiguous Disk Layout and High-Bandwidth Disk Systems

Typical general-purpose computers store many small, temporary files on disk. Eventually the disk becomes fragmented, and blocks of a file may be scattered randomly across the disk. We anticipate that a parallel file system relying on DDIO will avoid much of this fragmentation and support contiguous layout of data, not necessarily of an entire file, but at least of the portion accessed during a single file

operation. Note that administrative files, programs, and other small files whose modification often leads to fragmentation are typically accessed from a separate file server and not the parallel file system. Their layout does not impact the parallel file system. DDIO's collective nature plays a big part in ensuring contiguous layout; rather than allocating a single disk block at a time, an I/O node using DDIO allocates in advance disk blocks needed for an entire collective operation. Therefore, it is essential that the file system using DDIO be optimized for transfers of contiguous data.

When transferring data to or from contiguous blocks, optimal overlap of message-passing operations with disk operations for fine-grained redistributions is not possible [52]. The HP 97560, modeled in the original DDIO paper [52], spins at 4002 RPM and has 72 512-byte sectors per track; a read or write of 16 sectors (one 8K block) takes 3.3 ms on this disk. The original DDIO scheme, required to send 1024 messages (double that for writes) per disk block for a Cyclic distribution of 8-byte records, takes approximately 6.5 ms in the message-passing phase for a block. The result is that the message-passing, not disk speed, is the bottleneck, and effective bandwidth is approximately half what the disks themselves provide. The imbalance is worse when we examine more recent disk technology. Today, disk speeds of 7200 RPM are common, while the density of data in sectors per block is rising to twice that of the HP 97560 and beyond [82]. The time needed to transfer a block to or from a state-of-the-art disk, assuming a contiguous data layout, is decreasing to close to 0.8 ms—giving a bandwidth of nearly 10 MB/s from a single disk. Several disks used together, perhaps in a RAID configuration, can provide this bandwidth and greater, even without the optimistic assumption of contiguous data layout [11].

Recall that on the Meiko CS-2, a representative time for packetizing a block of 8-byte elements is 1.20 ms; the corresponding time for unpacking received data for a write is 1.18 ms. These times are for memory operations only and do not include message passing, so the time needed to build or unpack packets of 8-byte records for one block actually exceeds the high-performance disk system transfer time of 0.8 ms *before we consider the added cost of sending messages.* Hence putting data from a block into packets and sending only one message per block to each compute node, namely PB-DDIO, does not provide a good balance between disk operations and message-passing for fine-grained redistributions when a high-bandwidth disk system is used. The time for packing and unpacking 4-byte records is considerably worse. Therefore, building packets is only a partial solution that does not guarantee balanced performance when a high-bandwidth disk system or single disk with contiguous file layout is used.

5.3. Alternatives to Simple Packet Building

In this section we propose two alternatives to PB-DDIO. The first scheme, like PB-DDIO, relies on the I/O nodes to pack or unpack data bound to or from compute nodes. This new scheme reduces the number of messages sent, and hence the impact of message latency, by putting data from $\mathcal{M}$ disk blocks into packets transmitted between compute nodes and I/O nodes. This approach, denoted MB-DDIO for "Multiple Block" DDIO, provides higher effective bandwidth than PB-DDIO, but it has two drawbacks. The first drawback is that MB-DDIO requires more memory on the I/O nodes than PB-DDIO or DDIO. This is not a major concern, because memory requirements go from about 32 KB for PB-DDIO to, perhaps, 256 KB for MB-DDIO, both modest amounts. A more significant disadvantage of

MB-DDIO is that the compute nodes must process data sent to or received from I/O nodes. Because of the fixed stripe size for the file, data from consecutive blocks on the same I/O node do not reside in contiguous compute node memory—compute nodes must unpack received data and pack data transmitted to I/O nodes. The ideal number of disk blocks to process for each message exchange varies from machine to machine. For our experiments, we combine 16 8 KB blocks to give a total of 128 KB transferred during each message-passing phase of the redistribution. The 128 KB value was chosen to avoid cache and TLB effects that might arise with a larger transfer size.

Another approach combines the efficient Block file transfers of DDIO with the Two-Phase Access Strategy [8] [27], giving an entirely different approach to optimizing fine-grained file redistributions. The Two-Phase Access Strategy used without DDIO suffers from the fact that the I/O nodes must react to uncoordinated, potentially ill-timed, individual block requests from compute nodes. When we add the optimizations of DDIO to the Two-Phase Access Strategy, I/O nodes can minimize disk access times by taking into account the entire collective operation being performed. In the hybrid strategy, denoted 2P-DDIO, which combines DDIO with the Two-Phase Access Strategy, data are transferred between disk and the compute nodes in the disk distribution (Block for one-dimensional arrays and Block-None for two-dimensional) and redistributed to the desired distribution by the compute nodes. When reading, the compute nodes perform a redistribution among themselves after receiving the data from I/O nodes. When writing, the data are redistributed among compute nodes before the I/O nodes begin the file write. With 2P-DDIO, file access is fast, and compute nodes perform no unpacking of data received from I/O nodes.

While the file operations of 2P-DDIO are fast, the redistribution phase is not performed in conjunction with the file operations. Hence, unlike with PB-DDIO and MB-DDIO, the redistribution in 2P-DDIO is not overlapped with disk operations[3]. 2P-DDIO requires more memory, on compute nodes rather than I/O nodes, than DDIO or PB-DDIO. It also rules out fully asynchronous I/O, since the compute nodes must participate in the redistribution prior to a write or after a read. On the positive side, 2P-DDIO requires much less work of the I/O nodes and maintains a clear division of work between I/O nodes and compute nodes. It utilizes compute nodes, which typically outnumber I/O nodes, for packet-building. Finally, its performance improves as disk speeds improve, a certainty for the foreseeable future. Kotz [52] concludes that the Two-Phase Access Strategy should be slower than DDIO, but that conclusion is based on the assumption that the Two-Phase Access Strategy is used with a traditional parallel file system, with its inherent prefetching mistakes and non-optimized disk accesses. When the Two-Phase Access Strategy is combined with DDIO's optimizations, the result can speed fine-grained file redistributions.

5.4. Building and Validating Models

5.4.1. Analytic Models for Four Disk-Directed File Redistribution Schemes

In this section we validate analytic models for the performance of DDIO, PB-DDIO, MB-DDIO, and 2P-DDIO. We model only reading here. The conclu-

---

[3]We do not take into account overlapping of packet-building for redistribution on compute nodes with disk operations. However, an aggressive implementation of read could perform packet-building by compute nodes upon receipt of the first blocks of data, thereby achieving some overlap and higher effective bandwidth than that shown in Section 5.5.

sions drawn from the experiments with these models can be qualitatively applied to writing as well, but simple analytical models cannot accurately describe for a variety of machines the overlapping of data requests and replies needed for writing. Our goal here is to establish the merits of MB-DDIO and 2P-DDIO; the simple, easily validated models for reading allow us to do that. Consistent with a high-performance file system or contiguous data layout with an element size of 8 bytes, the models for DDIO, PB-DDIO, and MB-DDIO assume the processing requirements for a block of data are greater than D_B, the time needed to access a block of disk data assuming a contiguous layout. To simplify the models presented here, we define $\mathcal{B}_{IO}$, the maximum number of file blocks at any I/O node, to be

$$\mathcal{B}_{IO} = \left\lceil \frac{\lceil \frac{Ne}{B} \rceil}{I} \right\rceil,$$

where N is the number of elements in the global array, e is the array element size, B is the file system block size, and I is the number of I/O nodes.

We model DDIO's performance, assuming message-passing dominates disk read time, when reading from a file with a contiguous layout as

$$T_{DDIO} = D_1 + \mathcal{B}_{IO}(\left\lceil \frac{B}{e} \right\rceil \lambda + B\beta_x), \tag{5.1}$$

where the first term represents the time needed to access the first disk block, and the remaining time is the number of blocks per I/O node times the message-passing costs per block. λ represents the message latency, while β is the inverse of network bandwidth, namely the time per byte for sending messages. Note that each message transmitted is smaller than B, but since the messages are serialized by the sender[4], the total amount of data sent for a block is B.

---

[4]We assume single-port communication.

For PB-DDIO, we assume packet building for a block dominates the actual disk block access time (recall that a node of the Meiko CS-2 takes 1.20 ms to packetize 8 KB of data, which is read from a state-of-the-art disk in just 0.8 ms). Hence, reading using PB-DDIO is modeled as

$$T_{PB} = D_1 + \mathcal{B}_{IO}(\hat{\pi}^{IO}_{BC}(e)B + P\lambda + B\beta). \tag{5.2}$$

After the first block is read, packets are built and transmitted for every block.

The MB-DDIO model is similar to that for PB-DDIO, with the exception that the number of messages sent is reduced, and the extra cost of unpacking at the compute nodes is included. During a read operation, $\mathcal{M}$ disk blocks are read and packetized before messages are sent from the I/O nodes, so the number of times messages are sent is $\lceil \mathcal{B}_{IO}/\mathcal{M} \rceil$. The unpacking, denoted by $\hat{v}_{mb}(\lfloor B/P \rfloor, \mathcal{M}I\lceil B/P \rceil)$, incurs relatively little overhead except when P gets large and the amount of data bound for a compute node in a block is small. Moreover, only the unpacking of the last blocks received does not overlap I/O node disk and packing operations; the number of blocks received in the last group is

$$\mathcal{L}_{mb} = (\mathcal{B}_{IO} + \mathcal{M} - 1)\%\mathcal{M} + 1, \tag{5.3}$$

in which the modulo (%) function adjusts for the case when the I/O nodes do not pack $\mathcal{M}B$ bytes for the last message. Therefore, for reading, our model of MB-DDIO time is

$$T_{mb} = D_1 + \mathcal{B}_{IO}B(\hat{\pi}^{IO}_{BC}(e) + \beta) + \left\lceil \frac{\mathcal{B}_{IO}}{\mathcal{M}} \right\rceil P\lambda + \hat{v}_{mb}(\lfloor B/P \rfloor, \mathcal{M}I\lceil B/P \rceil)I\mathcal{L}_{mb}\left\lceil \frac{B}{P} \right\rceil. \tag{5.4}$$

We model the time a 2P-DDIO read takes as

$$T_{2P} = D_1 + D_B(\mathcal{B}_{IO} - 1) + \lambda + B\beta + \hat{\pi}_{\mathrm{BC}}(e, P)\left\lceil \frac{Ne}{P} \right\rceil + P\lambda + \beta_x \left\lceil \frac{Ne}{P} \right\rceil. \tag{5.5}$$

The first two terms represent disk read time, and the next two represent moving the last block of data read (which does not overlap any disk operations) from the I/O nodes to the compute nodes. The remaining costs are for building packets and exchanging data among compute nodes. Note that the $\hat{\pi}_{BC}(e, P)$ term, because it reflects the redistribution of an entire array, is impacted by Translation Lookaside Buffer (TLB) effects as P increases.

5.4.2. Validating the Models

In this section we empirically validate the models for DDIO, PB-DDIO, MB-DDIO, and 2P-DDIO presented in the previous section using experiments on a Meiko CS-2 multicomputer. The CS-2 at Oregon State University consists of 16 nodes connected by a 40 MB/s bidirectional, fat-tree, indirect network. Each node has a communication coprocessor facilitating low latency user-accessible communication. Although each node of the CS-2 has a local disk, we do not have control of the disks at the level required for DDIO. This does not affect our models, though, because our assumption is that message-passing time combined with packet building and unpacking time exceeds the disk block access time. Hence, these times form the critical path in the execution of DDIO, PB-DDIO, and MB-DDIO.

The data for our DDIO, PB-DDIO, and MB-DDIO validation runs is shown in Table 5.2. Each block of the table includes three values. The top value is the actual run on the CS-2 in milliseconds, the middle value is the modeled value in ms, and the bottom value, in boldface, is the relative error of the model. The actual runs, with the exception of the DDIO results on 50 MB files, consist of the average of 10 runs. The maximum coefficient of variation (i.e., the ratio of standard deviation to mean) among the values is 0.022. Because of their excessive duration,

DDIO times for 50 MB files are the average of two runs. The DDIO read model slightly underestimates the actual read time. This is not surprising, since our model does not account for the excess loading on each node and the network caused by the overwhelming number of fine-grained messages. However, because DDIO is not competitive with the other schemes presented here, the slightly optimistic nature of our DDIO model has no qualitative impact on our results.

The results in Table 5.2 show that our models are accurate for small values of P. As shown in Chapter 4, increasing P has no impact on packet building or unpacking times when TLB misses do not occur—the case in I/O node packetizing. Therefore, $\hat{\pi}_{BC}^{IO}(e)$ does not vary with P. Neither does the message-passing model for reading (i.e., transmitting a block of data from I/O nodes to compute nodes in time $P\lambda + B\beta$) lose accuracy as P increases. Hence, our models scale to larger values of P.

For 2P-DDIO, we simply validate the data redistribution portion of the model. The remainder of the model, with the exception of the transmission of data not overlapped with disk operations, consists of disk parameters, which are modeled independently of the redistribution. The results of our redistribution model are shown in Table 5.3. As P increases, both the reading and writing models take into account TLB effects, so the packet building and unpacking components maintain their accuracy. On recent scalable parallel computers, the bisection bandwidth increases with P, so the message-passing performance predicted by our model (only 50% of peak bandwidth) for the complete exchange should be maintainable for larger systems.

P	Array Size	I	DDIO Read	PB-DDIO Read	MB-DDIO Read	
12	10MB	4	6221	517	482	Observed (ms)
			5720	517	458	Predicted (ms)
			8.1	**0.0**	**5.0**	Error (%)
		2	12230	1022	935	
			11427	1032	912	
			6.6	**1.0**	**2.5**	
	50MB	4	31195	2593	2367	
			28547	2580	2277	
			8.5	**0.5**	**3.8**	
		2	61588	5125	4657	
			57082	5159	4548	
			7.3	**0.7**	**2.3**	
8	10MB	4	6242	498	482	
			5720	495	459	
			8.4	**0.6**	**4.8**	
		2	12239	975	932	
			11427	989	910	
			6.6	**1.4**	**2.4**	
	50MB	4	31371	2492	2361	
			28547	2471	2271	
			9.0	**0.8**	**3.8**	
		2	58873	4896	4641	
			57082	4941	4535	
			3.0	**0.9**	**2.3**	

TABLE 5.2. Comparison of our analytic model with actual run times for file redistribution schemes on a Meiko CS-2. The compute node distribution is Cyclic, and array elements are 8 bytes. Each block of the table includes three values. The top value is the actual run on the CS-2 in milliseconds, the middle value is the modeled value in ms, and the bottom value, in boldface, is the relative error of the model. The actual runs, with the exception of the DDIO results on 50 MB files, consist of the average of 10 runs. The maximum coefficient of variation (i.e., the ratio of standard deviation to mean) among the values is 0.022. Because of their excessive duration, DDIO times for 50 MB files are the average of two runs.

Data Redistribution		10 MB			50 MB			
	$P \rightarrow$	16	12	8	16	12	8	
BC		139	189	268	690	947	1334	Observed (ms)
		138	184	277	691	922	1383	Predicted (ms)
		0.5	**2.4**	**3.2**	**0.2**	**2.6**	**3.7**	Error (%)

TABLE 5.3. Comparison of our analytic model with actual run times for Block to Cyclic (BC—used for reads) redistributions on a Meiko CS-2. The compute node distribution is Cyclic, and array elements are 8 bytes. Each block of the table includes three values. The top value is the average time for 10 runs on the CS-2 in milliseconds, the middle value is the modeled value in ms, and the bottom value, in boldface, is the relative error of the model. The largest coefficient of variation among the actual runs is 0.006.

5.5. Performance Comparison of the Disk-Directed I/O Redistribution Schemes

5.5.1. 8-Byte Record Size

For our comparison of 2P-DDIO, MB-DDIO, PB-DDIO and DDIO, we use two different machines for our models. Their parameters are shown in Table 5.4. The first, dubbed "DDIO machine" is similar to the one simulated in [52]. This machine has low message-passing latency and a high-bandwidth network in which we assume half its peak bandwidth is utilized when performing an all-to-all communication. The second machine is the Meiko CS-2. Because packing and unpacking are not discussed in [52], our model of the DDIO machine uses the Meiko's values for $\hat{\pi}^{IO}_{BC}(e)$, $\hat{\pi}_{\text{BC}}(e, P)$, and $\hat{v}_{mb}(\lfloor B/P \rfloor, \mathcal{M}I\lceil B/P \rceil)$. For reference, the $\hat{\pi}_{\text{BC}}(e, P)$ value's behavior is dictated by the Meiko CS-2's HyperSPARC processor, whose TLB has 64 entries and uses random replacement.

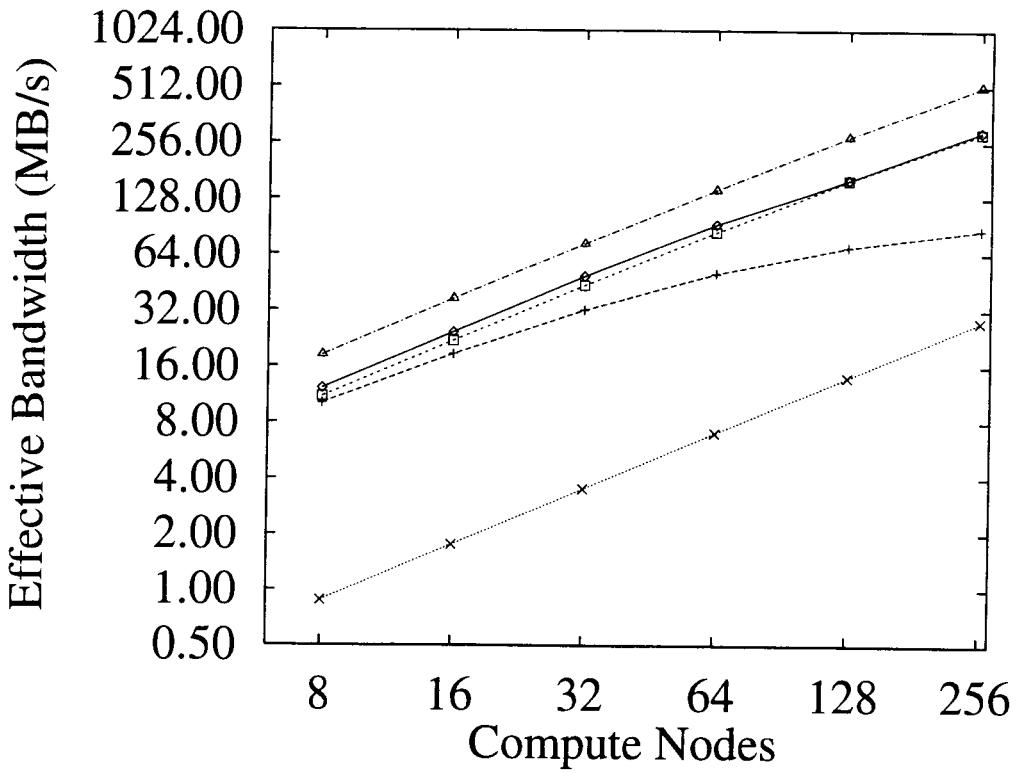
DDIO Machine	
λ	2 μs
β	0.001667 μs/byte
β_x	0.003333 μs/byte

Meiko CS-2	
λ	17 μs
β	0.025 μs/byte
β_x	0.052 μs/byte

Shared Parameters	
$\hat{\pi}_{\text{BC}}(e, P), P < 64$	0.159029 μs/byte
$\hat{\pi}_{\text{BC}}(8, 64)$	0.1769 μs/byte
$\hat{\pi}_{\text{BC}}(8, 128)$	0.2593 μs/byte
$\hat{\pi}_{\text{BC}}(8, 256)$	0.2803 μs/byte
$\hat{v}_{mb}(32, \mathcal{M}I\lceil B/P \rceil)$	0.1357 μs/byte
$\hat{v}_{mb}(64, \mathcal{M}I\lceil B/P \rceil)$	0.0982 μs/byte
$\hat{v}_{mb}(128, \mathcal{M}I\lceil B/P \rceil)$	0.0816 μs/byte
$\hat{v}_{mb}(256, \mathcal{M}I\lceil B/P \rceil)$	0.0744 μs/byte
$\hat{v}_{mb}(512, \mathcal{M}I\lceil B/P \rceil)$	0.0712 μs/byte
$\hat{v}_{mb}(1024, \mathcal{M}I\lceil B/P \rceil)$	0.0698 μs/byte
D_1	13 ms
D_B	0.844 ms
B	8192 bytes
e	8 bytes
N	6553600

TABLE 5.4. Machine parameters used to generate the data in this section.

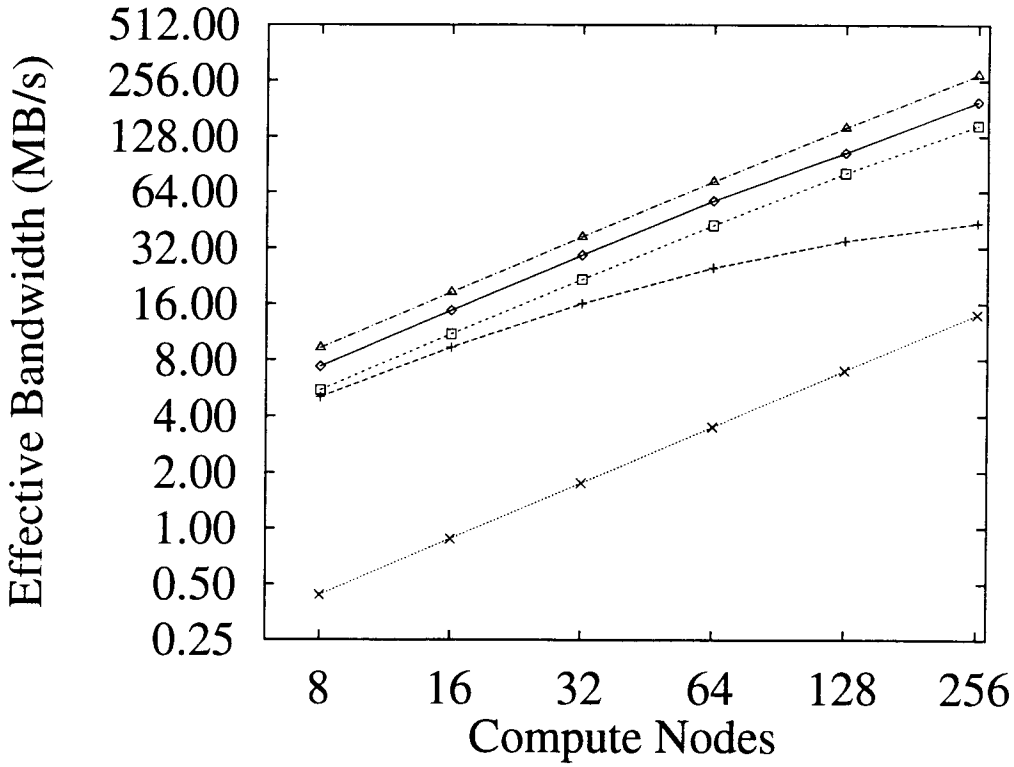
At the I/O nodes, we assume the disk spins at 7200 RPM, has 172 sectors (21.5 8K blocks) per track, and 10 tracks per cylinder. The average block read time, including skews, of a contiguous file is 0.833 ms, resulting in a bandwidth of 9.26 MB/s. These numbers represent a state-of-the-art disk drive as of early 1996 [82].



$P/I = 4$

		P					
Key	Method	8	16	32	64	128	256
△	OPT	18.4	36.7	72.7	142.7	275.1	513.5
◊	2P-DDIO	12.2	24.3	48.4	92.9	160.9	295.5
□	MB-DDIO	11.0	21.9	43.2	84.2	160.1	288.7
+	PB-DDIO	10.1	18.6	32.0	50.0	69.7	86.7
×	DDIO	0.9	1.8	3.5	7.0	14.0	27.8

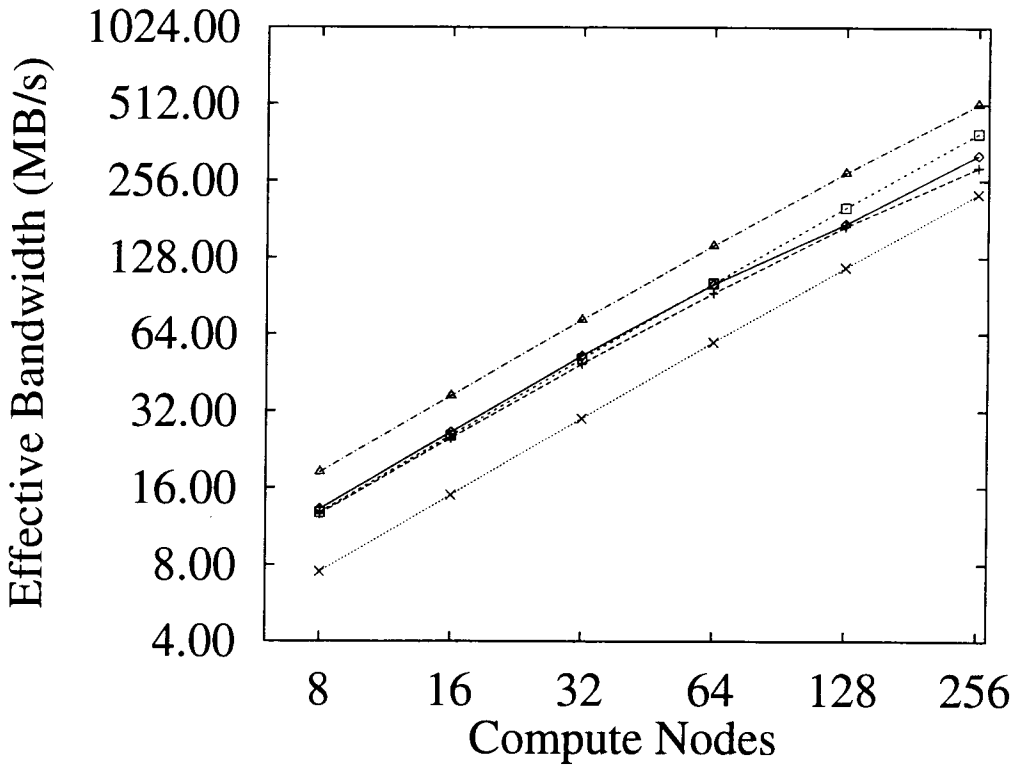
FIGURE 5.2. Modeled bandwidth using Meiko parameters reading a 50 MB file to a Cyclic distribution of 8-byte records with a P/I ratio of 4.



$P/I = 8$

		P					
Key	Method	8	16	32	64	128	256
△	OPT	9.2	18.4	36.7	72.7	142.7	275.1
◊	2P-DDIO	7.4	14.7	29.3	57.1	104.3	197.2
□	MB-DDIO	5.5	11.0	21.7	42.3	80.6	147.0
+	PB-DDIO	5.1	9.3	16.0	25.0	34.9	43.4
×	DDIO	0.4	0.9	1.8	3.5	7.0	14.0

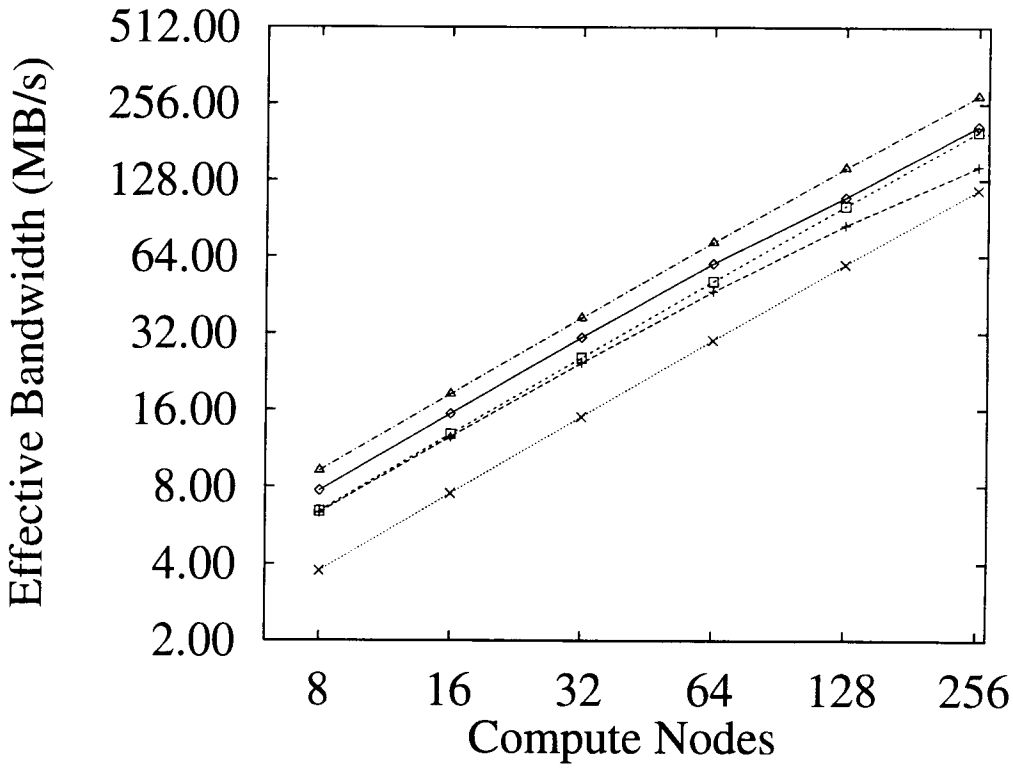
FIGURE 5.3. Modeled bandwidth using Meiko parameters reading a 50 MB file to a Cyclic distribution of 8-byte records with a P/I ratio of 8.



$P/I = 4$

		P					
Key	Method	8	16	32	64	128	256
△	OPT	18.4	36.7	72.7	142.7	275.1	513.5
◊	2P-DDIO	13.2	26.4	52.5	100.5	173.1	322.2
□	MB-DDIO	12.8	25.6	51.0	101.3	200.7	393.1
+	PB-DDIO	12.7	25.0	48.8	92.8	169.3	287.8
×	DDIO	7.5	15.0	29.9	59.3	116.8	226.7

FIGURE 5.4. Modeled bandwidth using the DDIO machine parameters (low latency, high bandwidth) reading a 50 MB file to a Cyclic distribution of 8-byte records with a P/I ratio of 4.



$P/I = 8$

		P					
Key	Method	8	16	32	64	128	256
△	OPT	9.2	18.4	36.7	72.7	142.7	275.1
◊	2P-DDIO	7.7	15.4	30.7	59.9	109.3	208.7
□	MB-DDIO	6.4	12.8	25.6	50.9	100.9	198.5
+	PB-DDIO	6.3	12.5	24.4	46.4	84.8	144.3
×	DDIO	3.8	7.5	15.0	29.9	59.3	116.8

FIGURE 5.5. Modeled bandwidth using the DDIO machine parameters (low latency, high bandwidth) reading a 50 MB file to a Cyclic distribution of 8-byte records with a P/I ratio of 8.

The central comparison we make is between PB-DDIO, MB-DDIO, and 2P-DDIO, although we also show the modeled results for DDIO. We use the formulas above to compare these schemes when reading a 50 MB file of 8-byte records with a Block distribution on disk and a compute-node distribution of Cyclic. We assume the file is divided into separate contiguous chunks of 512 KB each, hence the average transfer time for each block using 2P-DDIO is slightly higher (with a value of 0.844 ms/block) than the maximum speed of 0.833 ms/block. Although this access time represents a single disk with contiguous data layout, it could just as well be a multi-disk RAID attached to each I/O node and delivering data at an effective 9.26 MB/s regardless of data layout. This figure does not directly impact MB-DDIO, PB-DDIO, or DDIO as long as the time taken for each of these to process a block is greater than the disk block access time. The bandwidth figure directly impacts 2P-DDIO's performance, since 2P-DDIO, as modeled, does not overlap unpacking of data with disk operations.

Several parameters affect the relative performance of these schemes. An important factor is the number of I/O nodes relative to the number of compute nodes. Reducing the number of I/O nodes negatively impacts all of the schemes due to less parallelism in the I/O system. Because, in the modeled environment, packetizing is more expensive than disk access, reducing the number of I/O nodes adversely impacts MB-DDIO and PB-DDIO more than 2P-DDIO. On parallel computers in use today, the ratio of compute nodes to I/O nodes is typically in the range of 3 to 16. We display our results using P/I ratios of 4 and 8. File size is 50 MB, which is large enough to ensure that I/O nodes on large systems have more than a trivial amount of data. Each of Figures 5.2 through 5.5 shows five logarithmic bandwidth graphs sharing a specific P/I ratio. Each graph includes a reference curve representing the optimal disk bandwidth, denoted OPT, for which optimal overlap is achieved.

Shown with each graph is the same data in tabular form. We now examine the data in detail.

Modeling the Meiko CS-2 reading a 50 MB file on a system with a P/I ratio of 4, as in Figure 5.2, 2P-DDIO outperforms MB-DDIO by 10% and PB-DDIO by 20% for small machine configurations. As P increases, 2P-DDIO becomes less efficient due to TLB misses when building many packets. PB-DDIO performance drops off even more noticeably as P increases, because P messages must be sent for each file block. The combining of messages for MB-DDIO allows its per-I/O-node performance to remain constant as P and I increase. When P/I increases to 8, as in Figure 5.3, 2P-DDIO achieves bandwidth 34% higher than MB-DDIO due to the much higher computing power available for packetizing on the compute nodes relative to the I/O nodes. 2P-DDIO bandwidth exceeds that of PB-DDIO by 45% for small P values, and the difference is much greater as P increases.

Message latency plays a big role in the relative performance of these schemes. Figures 5.4 and 5.5 are based on the DDIO machine, which has much lower latency than the Meiko. Here, the difference in performance between 2P-DDIO, PB-DDIO, and MB-DDIO is negligible when P/I is 4, with MB-DDIO outperforming 2P-DDIO as P increases due to the impact of TLB effects on 2P-DDIO. PB-DDIO performance does not tail off as significantly as for the Meiko, since message latency is so low on the DDIO machine. When P/I increases to 8, 2P-DDIO achieves a 20% speedup, decreasing slightly as P increases, over MB-DDIO and PB-DDIO.

5.5.2. Impact of Increasing Record Size

So far, we have examined different approaches for fine-grained redistribution of data with a specific record size of 8 bytes. Since 8 bytes is the size of a typical

e	$\hat{\pi}^{IO}_{BC}(e)$	$\hat{\pi}_{BC}(e,P)$
8	0.1469	0.1590
16	0.0747	0.0905
24	0.0631	0.0812
32	0.0574	0.0749
40	0.0541	0.0732
48	0.0516	0.0706
56	0.0505	0.0699
64	0.0485	0.0677

TABLE 5.5. Packet-building costs in μs/byte on I/O nodes and compute nodes as e increases. For the fixed value of $P = 32$ in this section, $\hat{v}_{mb}(\lfloor B/P \rfloor, \mathcal{M}I\lceil B/P \rceil) = 0.07438$.

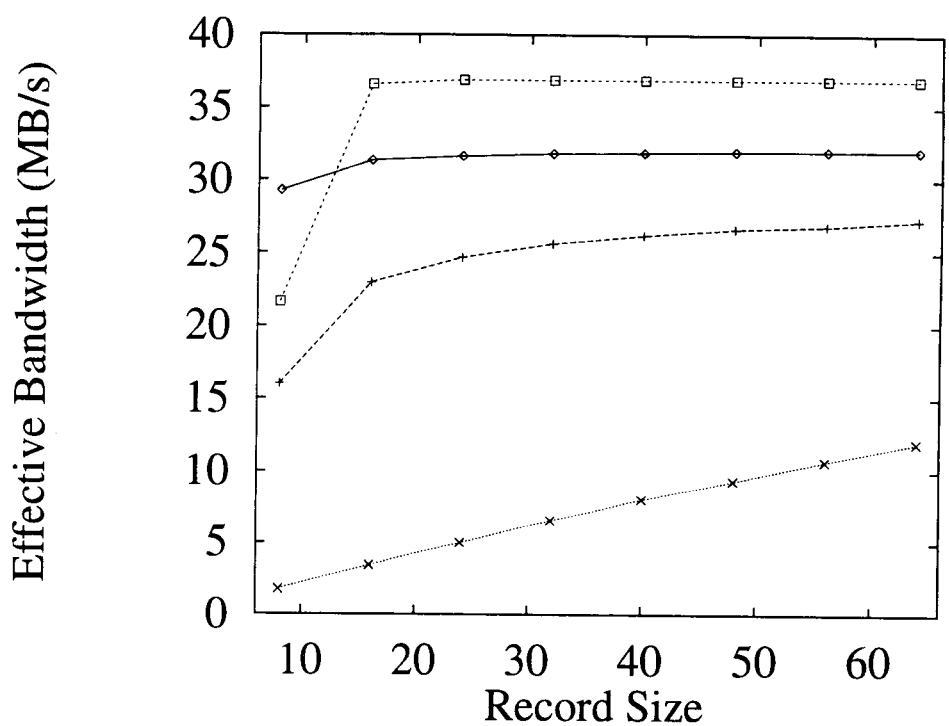


FIGURE 5.6. Bandwidth of 2P-DDIO (◊), MB-DDIO (□), PB-DDIO (+), and DDIO (×) while reading a file to a Cyclic distribution. The Meiko CS-2, with $P = 32$ and $I = 4$, is the modeled machine. Each I/O node's disk system provides data at 9.26 MB/s. The array record or element size, e, varies from 8 to 64.

double-precision floating point value, scientific applications frequently deal with 8-byte records. However, array elements may be larger, or a distribution like Cyclic(2), in which array elements are distributed to compute nodes two at a time, may be used. In this section we examine the impact on 2P-DDIO, PB-DDIO, and MB-DDIO of increasing the record size. To reduce the amount of data presented, we limit our experiments here to a system with 32 compute nodes and 4 I/O nodes. The resulting P/I ratio, still optimistic relative to the I/O balance found on many parallel machines, favors 2P-DDIO when $e = 8$. In the experiments, a 50 MB file is read to a Cyclic distribution. Unless otherwise stated, the disk bandwidth is 9.26 MB/s, the bandwidth delivered by a state-of-the-art disk using contiguous layout. The models presented in Section 5.4.1 are used here, but they must be used in conjunction with the disk parameters used in Section 5.5. That is, when packetizing costs for PB-DDIO and MB-DDIO are less than disk block access times, packetizing must wait on disk accesses.

As shown in Table 5.5, packing costs decrease substantially as e increases. As noted above, this cost decrease leads to a condition in which packetizing time for a block of data moves below the disk block access time. Further increases of e allow both packetizing *and* message-passing to overlap disk operations. When this occurs, 2P-DDIO cannot compete with PB-DDIO and MB-DDIO. Factors other than e also come into play, as we see in Figures 5.6 through 5.8.

Figure 5.6 shows the modeled times for the Meiko CS-2. MB-DDIO outperforms 2P-DDIO when $e \geq 16$. The plateau on the MB-DDIO curve implies that it reaches its maximum performance when $e \geq 16$. This is close to optimal overlap, because the disk controller continues to read ahead [57] while MB-DDIO's intermittent message-passing takes place. The quick processing of prefetched blocks allows the packetizing to catch up to the disk by the time the next messages are required.

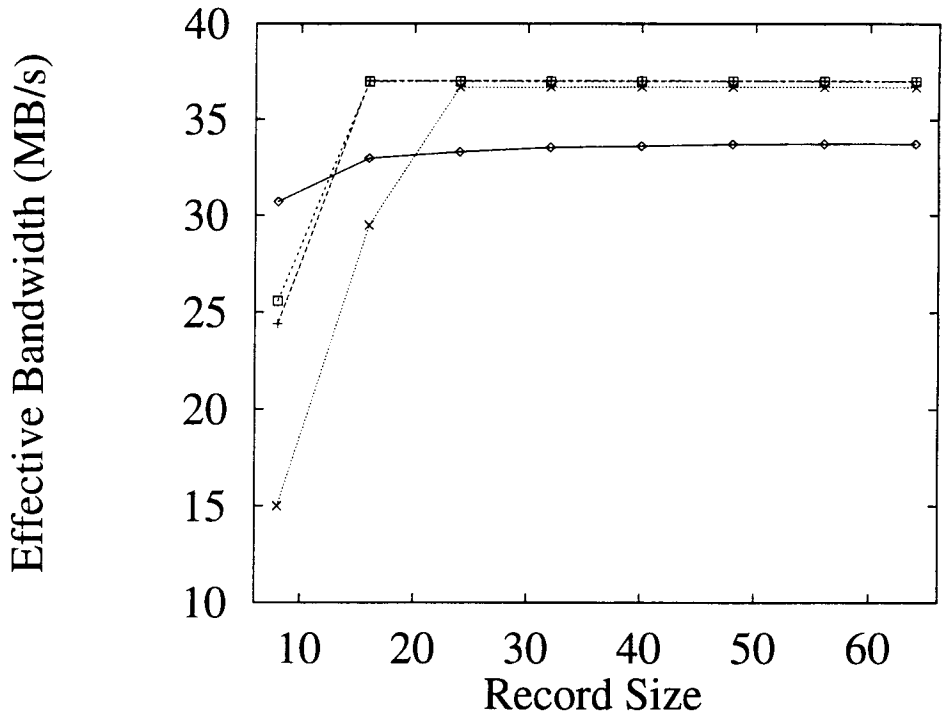


FIGURE 5.7. Bandwidth of 2P-DDIO (◊), MB-DDIO (□), PB-DDIO (+), and DDIO (×) while reading a file to a Cyclic distribution. The low-latency DDIO machine, with $P = 32$ and $I = 4$, is the modeled machine. Each I/O node's disk system provides data at 9.26 MB/s. The array record or element size, e, varies from 8 to 64.

PB-DDIO, with its high message-passing costs, cannot hide both packetizing and message-passing behind disk operations.

In Figure 5.7 the low-latency, high-bandwidth DDIO machine is modeled. We see that, because of their low message-passing costs, both PB-DDIO and MB-DDIO move to their optimal values when $e \geq 16$. 2P-DDIO benefits only slightly from the lower latency. Note that, with $e = 24$ and a block size of 8192 bytes, 342 messages are sent when DDIO is used. With this low-latency machine, these messages can be overlapped with disk operations, and DDIO achieves optimal overlap.

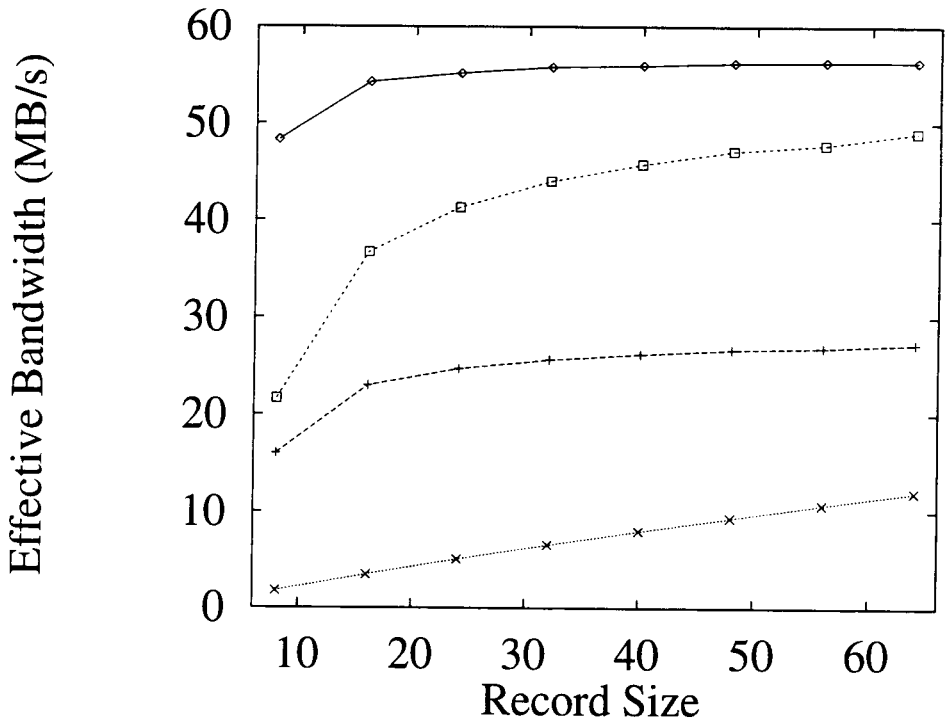


FIGURE 5.8. Bandwidth of 2P-DDIO (◊), MB-DDIO (□), PB-DDIO (+), and DDIO (×) while reading a file to a Cyclic distribution. The Meiko CS-2, with $P = 32$ and $I = 4$, is the modeled machine. Each I/O node's disk system provides data at 18.52 MB/s. The array record or element size, e, varies from 8 to 64.

	10 MB		50 MB	
I \ $P \rightarrow$	12	8	12	8
4	8.3	10.8	8.6	11.3
2	7.0	7.8	7.1	7.9

TABLE 5.6. Sustained disk bandwidth in MB/s, per I/O node, needed for 2P-DDIO to match MB-DDIO performance. These data verify that, for 2P-DDIO to be competitive, required disk bandwidth diminishes as P/I increases.

To determine the impact of disk bandwidth, we compare these file redistribution schemes using the Meiko CS-2 machine parameters, but assuming disk bandwidth twice that of our previous models—18.52 MB/s. The results are shown in Figure 5.8. Although sustaining this bandwidth may be difficult today, disk speeds are increasing quickly, and arrays of disks have the potential to deliver much higher data rates. With this high-performance disk system, the results are not surprising. The I/O nodes packetize data for PB-DDIO and MB-DDIO at the same rate as before, but the time taken for disk accesses shrinks, resulting in less overlap of packetizing with disk operations. The higher disk performance provides an overall bandwidth improvement for 2P-DDIO. Analyzing bandwidth needs differently, we can use the actual validation runs on the Meiko CS-2 (shown in Tables 5.2 and 5.3) to calculate the minimum sustained disk bandwidth needed at each I/O node for 2P-DDIO to match MB-DDIO performance. The results are shown in Table 5.6. As required bandwidth decreases, 2P-DDIO becomes the fastest choice for more configurations. Higher required disk bandwidths favor MB-DDIO. Note the dependence of the required bandwidth on P/I.

Finally, Figure 5.9 shows the results when the packetizing speed (in MB/s) increases by 50%. As packetizing cost decreases, PB-DDIO and MB-DDIO overlap their packetizing operations on I/O nodes with disk operations, moving these schemes toward optimal overlap. The packetizing speed increase improves 2P-DDIO's performance as well, but the large number of compute nodes among which data are divided relative to the number of I/O nodes makes the impact less significant than at the I/O nodes. Note that this modification to machine parameters, namely increasing the packet-building speed, closes the gap between 2P-DDIO and MB-DDIO much faster than decreasing communication cost (see Figure 5.5). How-

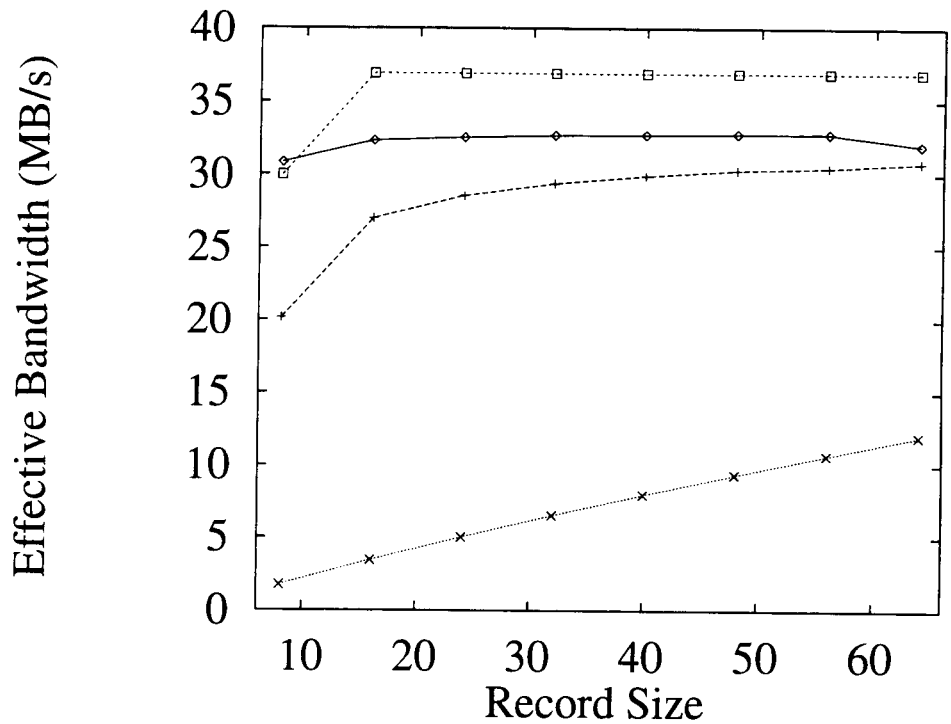


FIGURE 5.9. Bandwidth of 2P-DDIO (◊), MB-DDIO (□), PB-DDIO (+), and DDIO (×) while reading a file to a Cyclic distribution. A modified Meiko CS-2 with packetizing performance 50% faster than the current CS-2 processors is modeled. $P = 32$ and $I = 4$. Each I/O node's disk system provides data at 9.26 MB/s. The array record or element size, e, varies from 8 to 64.

Parameter	Impact
• P/I	The more processors that can be used for packetizing, the most expensive part of a redistribution, the better. Hence, 2P-DDIO has better performance relative to PB-DDIO and MB-DDIO as P/I increases.
• Disk System Bandwidth	As disk bandwidth increases, PB-DDIO and MB-DDIO packetizing operations take the same time. The file access component of 2P-DDIO shrinks, resulting in better time for 2P-DDIO relative to PB-DDIO and MB-DDIO.
• Message-Passing Latency	Low latency helps MB-DDIO, and, to a greater extent, DDIO and PB-DDIO; latency differences have minimal impact on 2P-DDIO.
• Packetizing Speed	Higher speed for building and unpacking packets helps PB-DDIO and MB-DDIO more than 2P-DDIO, since they must perform packetizing with fewer processors than 2P-DDIO uses. A high enough packetizing speed gives PB-DDIO and MB-DDIO optimal overlap with disk operations, in which case 2P-DDIO cannot compete. DDIO is not impacted by changes in packetizing speed.
• Array Element or Record Size	As the record size increases, the packetizing speed increases, so the impact is the same as increasing the packetizing speed described above.
• P	As P increases, TLB miss costs increase for 2P-DDIO; message-passing costs increase for PB-DDIO, somewhat less for MB-DDIO. DDIO is not impacted.

TABLE 5.7. Summary of factors affecting schemes for fine-grained redistribution of file data.

ever, all of these machine characteristics have a significant impact on which file redistribution method is fastest.

5.6. Conclusions

We have presented and validated models for Disk-Directed I/O (DDIO) and Packet-Based DDIO. To overcome the shortcomings of these schemes for fine-grained redistribution of file data, we have proposed a packet-based scheme which further reduces the number of messages sent during the redistribution, and Two-Phase DDIO, an approach combining the strengths of both DDIO and the Two-Phase Access Strategy. In all the configurations presented here, 2P-DDIO provided the highest bandwidth when reading 8-byte records to compute nodes desiring a Cyclic data distribution. However, 2P-DDIO, in many modeled configurations, loses its edge to MB-DDIO as the record size increases. Our model for 2P-DDIO does not include aggressive overlap of compute-node packing of data with I/O node disk reads, so higher performance than that shown here can realistically be achieved using 2P-DDIO.

Many factors influence the performance of fine-grained file redistributions. These factors, along with their impacts on the redistribution schemes examined in this chapter, are summarized in Table 5.7. File system designers can use the results here in conjunction with their machine parameters to determine which of these schemes should be incorporated into their file system utilizing Disk-Directed I/O. Of course, considerations such as system complexity and ease of integration into a complete prefetching and write-buffering strategy may also dictate which schemes are implemented. Ideally, the programmer ignores implementation details while the system chooses the best method for the requested redistribution. Outside of the file system, parallel programmers who can optimize their programs more aggressively than a generic file system can might choose to use the Two-Phase Access Strategy for fine-grained redistributions, if the underlying disk-directed system eschews im-

plementation of 2P-DDIO. Because of the nature of actions taken on the I/O nodes, PB-DDIO and MB-DDIO must be built into the file system and cannot be easily replaced by a user optimization.

6. COLLECTIVE PREFETCH WITH REDISTRIBUTION FOR DISK-DIRECTED I/O

6.1. Introduction

In the last chapter we explored various ways to speed up fine-grained redistributions of file data on top of a Disk-Directed I/O (DDIO) foundation. Another tool the file system can use to speed up file accesses is prefetching. Prefetching is performed by existing file systems like the Paragon's PFS [45], in which each I/O node performs a one-block read-ahead operation after a requested read. Because read requests come asynchronously and unordered from compute nodes, these prefetched blocks are often removed from the prefetch buffer before they are accessed, and the cost of the prefetch is wasted. The prefetches are not technically mistakes; they are simply ill-timed. However, the cost of such a prefetch is the same as that of a mistaken prefetch.

As Kotz says of DDIO [52], "Prefetching ... make[s] no mistakes." This statement is true only because DDIO does not perform prefetching. So, while DDIO does not suffer from prefetching mistakes, neither does it improve performance through successful prefetching. This is a significant weakness, because a file system that works as closely with compute nodes as does DDIO should be able to optimize structured reads with prefetching.

The mechanism through which DDIO gains file access knowledge, namely collective operations, can also be used to guide prefetching. We propose the notion of *collective prefetching*, in which the file system is not misled by seemingly random fine-grained block-at-a-time file requests, but instead takes the global view of

Disk-Directed I/O. With this view, the pattern of previous collective operations or run-time hints [75] [85] can guide the file system to aggressively prefetch for subsequent operations. Large-scale, aggressive prefetching should be effective for most scientific applications, which tend to access data in a regular, sequential fashion, redistributions notwithstanding [22, 36, 56, 63, 64, 77]. Because most file operations for data-parallel applications, when viewed collectively, access a contiguous segment of a file, collective prefetching schemes can utilize a global "one segment read-ahead" policy rather than a local one block read-ahead. Such a prefetching policy, combined with applications using regular file accesses, can greatly increase the effective file system bandwidth. However, prefetching alone does not address the common situation in which the compute node distribution differs from the I/O node distribution. When MB-DDIO is used for the redistribution of file data, for example, simply prefetching disk data may provide *no speedup* for a read requiring a fine-grained redistribution. When a read request asks for the prefetched data, every prefetched block must be put into packets, an expensive operation. As shown in the previous chapter, the cost of packet-building is higher than disk access costs for high-bandwidth disk systems. Hence, pre-reading disk data is not enough—the file system must prepare the data for its ultimate distribution as a part of the prefetch operation.

In this chapter we analyze several approaches to prefetching array data with redistribution. We model two costs for each approach. The first cost is the time spent reorganizing data during the prefetch phase of the read. This cost determines how many blocks are prefetched in a given amount of time. The second cost is the time taken to get the data properly organized on the compute nodes. This cost depends on what, if any, reorganization of prefetched data is performed by the I/O nodes. For the prefetching schemes presented here, these two costs are inversely

related; a scheme with a high prefetching cost has a low post-request reorganization cost. These costs are modeled for both fine-grained and large-grained redistributions in conjunction with MB-DDIO. Our results show (1) by overlapping I/O node packet-building with disk reads, significant gains in bandwidth can be achieved over collective prefetching alone; and (2) rarely is it worthwhile to reorganize data on I/O nodes to reduce the unpacking costs on compute nodes.

This chapter begins by developing models for calculating the amount of data prefetched and the time to fulfill a request using several prefetching schemes. These schemes are then compared using stochastic workloads modeling somewhat balanced computation and I/O requirements. We then compare the schemes under optimal conditions to determine the maximum benefit derived from each prefetching scheme. Table 6.1 lists the parameters used in our models.

6.2. Prefetching and Redistribution Approaches

6.2.1. Preliminaries

We assume here the redistribution algorithm in use is the MB-DDIO scheme presented in the previous chapter. Recall that processors using MB-DDIO combine data from several disk blocks in their messages, thereby reducing the impact of message latency relative to PB-DDIO. The MB-DDIO model in the previous chapter was for fine-grained file redistributions. For our discussion here, we define a fine-grained redistribution as one for which packet-building for a disk block takes longer than reading or writing a block. As in the last chapter, we assume a contiguous file layout and fast block access times. While the focus of the previous chapter is fine-grained file redistributions, the analysis here includes large-grained redistributions. We loosely define large-grained redistributions as those in which data from $\mathcal{M}$ disk

Parameter	Meaning
P	Number of compute nodes
I	Number of I/O nodes
P_S	Maximum number of compute nodes to which an I/O node sends data
B	I/O node block size in bytes
$\mathcal{B}_{pre}$	Number of disk blocks prefetched
$\mathcal{B}_{IO}$	Maximum number of blocks on one I/O node
$\mathcal{L}_{mb}(\mathcal{B})$	Number of blocks sent in the last message of MB-DDIO when each I/O node hold $\mathcal{B}$ blocks
$\mathcal{M}$	Number of disk blocks combined in messages for MB-DDIO
D_1	Time to seek, rotate, and transfer first block
D_B	Time to read a single disk block *after* head positioned
λ	Message startup latency
β	Time per byte to send data over the network
β_x	Time per byte on loaded network (for complete exchange)
N	Number of array elements to read/write
e	Size of array elements in bytes
e_v	Number of bytes in a disk block bound for contiguous memory on one compute node
n_v	Total number of bytes unpacked by one node
$\hat{v}_{mb}(e_v, n_v)$	Time per byte to unpack packets where n_v bytes are unpacked e_v bytes at a time
$\hat{v}_{mb}^{IO}(e_v, n_v)$	Time per byte to unpack packets on I/O nodes where n_v bytes are unpacked e_v bytes at a time
$\hat{\pi}_{\mathrm{BC}}(e, P)$	Time per byte to build P packets e bytes at a time including cache and TLB effects (BC $\equiv$ Block-to-Cyclic)
$\hat{\pi}_{BC}^{IO}(e)$	Time per byte for an I/O node to build packets e bytes at a time

TABLE 6.1. Parameters used in our model.

blocks can be put into packets and transmitted in less time than $\mathcal{M}$ disk block reads. For our large-grained redistribution experiments, an element or record size e of 256 is used.

We briefly review the MB-DDIO model here. First, the number of blocks sent in the last message to the compute nodes from I/O nodes is

$$\mathcal{L}_{mb}(\mathcal{B}) = (\mathcal{B} + \mathcal{M} - 1) \bmod \mathcal{M} + 1.$$

With the fine-grained assumptions made in the previous chapter, we could assume that each disk block processed by an I/O node contained data bound for each compute node. This assumption does not hold as e increases. For a given read operation, the maximum number of compute nodes to which any I/O node sends data, P_S, depends on $\wp$, the number of nodes with which data are exchanged, and l, the number of blocks processed:

$$P_S(\wp, l) = \min(\wp, \left\lceil \frac{lB}{e} \right\rceil).$$

Similarly, the grain size of data unpacked at each compute node, e_v, depends on how much data each block holds for a given compute node. For a fine-grained redistribution, the fewest bytes one compute node receives from a block is $\lfloor B/P \rfloor$. For a large-grained redistribution, not counting some splitting of records across blocks, a block contains e bytes per compute node. Hence,

$$e_v = \max\left(\left\lfloor \frac{B}{P} \right\rfloor, e\right).$$

Given these parameters, we recall the fine-grained MB-DDIO read model:

$$\begin{aligned} T^{fg}_{mb}(\mathcal{B}) = \; & D_1 + B\mathcal{B}(\hat{\pi}^{IO}_{BC}(e) + \beta) + \left\lceil \frac{\mathcal{B}}{\mathcal{M}} \right\rceil P_S(P, \mathcal{M})\lambda + \\ & I\mathcal{L}_{mb}(\mathcal{B}) \left\lceil \frac{B}{P} \right\rceil \hat{v}_{mb}(e_v, I\mathcal{M}\lceil B/P \rceil). \end{aligned} \tag{6.1}$$

The large-grained model is dominated by disk read time and includes the time taken to buffer and send the last blocks of data to the compute nodes. Besides disk time,

the major factor in this model is the time taken for the compute nodes to unpack their data, with the most data per compute node being $I\mathcal{L}_{mb}(\mathcal{B})\lceil B/P \rceil$ bytes.

$$\begin{aligned} T^{lg}_{mb}(\mathcal{B}) = \; & D_1 + (\mathcal{B}-1)D_B + \hat{\pi}^{IO}_{BC}(e)B + \lambda P_S(P, \mathcal{L}_{mb}(\mathcal{B})) + B\mathcal{L}_{mb}(\mathcal{B})\beta + \\ & I\mathcal{L}_{mb}(\mathcal{B}) \left\lceil \frac{B}{P} \right\rceil \hat{v}_{mb}(e_v, I\mathcal{M}\lceil B/P \rceil). \end{aligned} \tag{6.2}$$

In the context of our experiments involving prefetching with redistribution, we assume each compute node prefetches the same number of blocks, $\mathcal{B}_{pre}$. When a request for prefetched data arrives, packet-building for the current block is completed, and all prefetched data packets are transmitted to the proper compute nodes. The models shown for the completion time of the read, $T_{fulfill}$, do not include the "cleanup" time for prefetched blocks currently being processed. However, the experiments presented in Sections 6.3 and 6.4 do include this delay before the MB-DDIO process is started on remaining (non-prefetched) data.

After the prefetched data are sent to the compute nodes, the MB-DDIO file redistribution scheme is performed on the remaining data. The completion time depends on the MB-DDIO completion time as well as the unpacking done by the compute nodes. If few blocks are prefetched, the compute nodes can unpack the prefetched blocks while the I/O nodes are building packets for the unprefetched data. In this case, unpacking on compute nodes overlaps packet-building on I/O nodes, and the completion time is simply the completion time of the MB-DDIO scheme. If many blocks are prefetched, the unpacking costs on the compute nodes may dominate. Hence, we model the time taken once MB-DDIO starts as the later of the MB-DDIO completion time and the time the compute nodes take to complete unpacking of all data they receive. The model of this completion time for both fine-grained and large-grained redistributions is

$$T_v = \max\left(I\left\lceil \frac{B}{P} \right\rceil (\mathcal{B}_{pre}\hat{v}_{mb}(e_v, I\mathcal{B}_{pre}\lceil B/P \rceil) + (\mathcal{B}_{IO} - \mathcal{B}_{pre})\hat{v}_{mb}(e_v, I\mathcal{M}\lceil B/P \rceil)), \right.$$
$$\left. T_{mb}(\mathcal{B}_{rem}) \right) \tag{6.3}$$

where T_{mb} is either T_{mb}^{fg} or T_{mb}^{lg}, as appropriate. Note that the unpacking cost is potentially different for the two types of unpacking done on compute nodes. The amount of prefetched data could be large enough to cause TLB miss effects to occur during unpacking. The data unpacked as part of the MB-DDIO redistribution is free of TLB effects due to the limited amount of data sent at a time.

We now use the building blocks described above to model three different prefetch schemes, two of which combine redistribution with prefetching.

6.2.2. Disk Prefetch Only

Prefetching disk data without processing it for redistribution, denoted here as DISK-PRE, has a prefetch cost based only on disk bandwidth, which is

$$T_{pre} = D_1 + (\mathcal{B}_{pre} - 1)D_B \tag{6.4}$$

The time seen by the application after the data are requested depends on the granularity of the redistribution. A fine-grained redistribution requires expensive packet-building that renders the prefetch useless, since the packet-building time for a disk block is larger than the block read time. For a large-grained redistribution, the prefetched data can be packetized by the I/O nodes and unpacked by the compute nodes at approximately the full memory copy bandwidth. The packing for prefetched data assumes a large prefetch buffer, in which TLB effects come into play, so we model $\hat{\pi}_{\mathrm{BC}}(e, P)$ as described in Chapter 4. The unpacking done by the compute nodes and the resulting value of T_v were described in Section 6.2.1. The

amount of time taken after data are requested but before MB-DDIO commences consists of packet-building and message-passing for prefetched data.

$$T_{fulfill} = B\mathcal{B}_{pre}(\hat{\pi}_{\text{BC}}(e, P) + \beta) + \lambda P_S(P, \mathcal{B}_{pre}) + T_v \tag{6.5}$$

Note that this model and the underlying approach are applicable only to large-grained redistributions. It suffers from the fact that I/O node packet-building is not overlapped with disk operations. However, this approach saves the I/O nodes from CPU-intensive packet-building until data are actually requested and may be useful if prefetching is speculative or not well-informed.

6.2.3. Prefetching with Packet Building

While disk prefetching alone may provide a mild boost in effective bandwidth for large-grained redistributions, I/O nodes should be able to further speed the redistribution by overlapping packet-building with disk reads during prefetching. This approach, which we denote PB-PRE, can potentially help both fine-grained and large-grained redistributions. In building our models for this scheme, we note that building packets for a block of data for large-grained redistributions is faster than reading the associated disk block, so the redistribution cost includes the disk read time plus the time to packetize the last block of data:

$$T_{pre} = D_1 + (\mathcal{B}_{pre} - 1)D_B + B\hat{\pi}_{\text{BC}}(e, P). \tag{6.6}$$

Since the prefetch buffer may be large, we assume cache and TLB miss costs in $\hat{\pi}_{\text{BC}}(e, P)$ are consistent with those described in Chapter 4; these costs are more expensive than those for canonical MB-DDIO, which accesses the same memory locations repeatedly. For fine-grained redistributions, the packetizing costs dominate, and the prefetching cost is

$$T_{pre} = D_1 + \mathcal{B}_{pre} B \hat{\pi}_{\mathrm{BC}}(e, P) \tag{6.7}$$

For either coarse- or fine-grained redistributions, the time to complete the redistribution after the data are requested is

$$T_{fulfill} = B\mathcal{B}_{pre}\beta + P_S(P, \mathcal{B}_{pre})\lambda + T_v, \tag{6.8}$$

where the MB-DDIO routine starts after the prefetched blocks are sent to the compute nodes.

6.2.4. Eliminating Unpacking Costs

While the above scheme overlaps packet-building with reading of prefetched data, the striped nature of the data requires that the packet from an I/O node to a compute node contains data not residing in contiguous memory locations on the compute node. Hence, compute nodes must unpack the data received. It may be desirable to eliminate this unpacking step at the compute nodes. The way we eliminate unpacking is by collecting all data bound for a compute node on a single I/O node acting as the compute node's representative. Each I/O node holds data bound for as many as $\lceil P/I \rceil$ compute nodes.

To collect data, the I/O nodes overlap prefetching of disk data with packet-building and exchange data among themselves every $\mathcal{M}$ blocks, ensuring that each I/O node has all prefetched data bound for compute nodes it represents. We denote this scheme PBX-PRE. The per-block prefetching cost of PBX-PRE is higher than that for PB-PRE due to communication among I/O nodes; hence the number of blocks prefetched in a given time is less than the number when simply building packets. The gain of PBX-PRE may be found at the end of the redistribution, when the compute nodes need not process the prefetched data sent to them. Because

the I/O nodes must pack and unpack data to collect it into contiguous chunks for represented compute nodes, we assume even a large-grained redistribution for a block takes longer than a disk block read, so both fine-grained and large-grained redistributions share the same model for the time taken to prefetch $\mathcal{B}_{pre}$ blocks with this scheme:

$$T_{pre} = D_1 + B\mathcal{B}_{pre}(\hat{\pi}^{IO}_{BC}(e) + \beta_x) + \left\lceil \frac{\mathcal{B}_{pre}}{\mathcal{M}} \right\rceil P_S(I, \mathcal{M})\lambda + B\mathcal{B}_{pre}\hat{v}_{mb}\left(e_v, I\mathcal{M}\left\lceil \frac{B}{P} \right\rceil\right) \tag{6.9}$$

Because I/O nodes collect data into ordered packets for each compute node as part of prefetching, compute nodes need not unpack received prefetch data. Hence, the cost to the requesting program for this scheme is

$$T_{fulfill} = \left\lceil \frac{P}{I} \right\rceil \lambda + B\beta\mathcal{B}_{pre} + T_{mb}(\mathcal{B}_{IO} - \mathcal{B}_{pre}), \tag{6.10}$$

where T_{mb} is either fine-grained or large-grained, as necessary.

6.3. Performance Comparisons

6.3.1. Machine and Workload Models

In this section we compare the performance of the prefetching schemes described in Section 6.2. Our models rely heavily on the validated parameters of the Meiko CS-2 shown in Table 6.2. The baseline configuration uses 32 compute nodes and 4 I/O nodes. We modify some of these values in later experiments to determine the robustness of our results. The experiments determine the effective program-level bandwidth, that is, the rate seen by the requesting program, measured from the request time until the time the request is satisfied. Each I/O node has a prefetch buffer for performing large-scale, collective prefetching. This buffer limits the amount of

data prefetched, regardless of the time available for prefetching. In our experiments, we vary the prefetch buffer size on each I/O node from 512 KB to 8 MB.

In our comparison of prefetching strategies, the compute-node workload chosen plays an important role. As done in [54] and [55], we model the workload using an exponential distribution in which average computation time is equal to the average file access time, hence the program is neither I/O bound nor compute bound. With less computation, minimal prefetching may be performed, while more computation allows more prefetching until the I/O node prefetch buffers become full. Further increases in computation time, while supporting maximal prefetch, reduce the relative impact of prefetching on the entire computation-I/O sequence. We assume a contiguous file layout with approximate disk bandwidth of 10 MB/s, giving a mean for the exponential distribution of $Ne/(10485760I)$ seconds. Each result presented in our experiments is the average resulting from 1000 runs using computation time (and therefore prefetch time) generated from this exponential distribution.

6.3.2. Analysis of Results

Our first experiments compare the effective bandwidth for the DISK-PRE, PB-PRE, and PBX-PRE schemes for both fine-grained and large-grained file redistributions with 50 MB arrays. These results are also compared to the bandwidth when no prefetching (NO-PRE) is performed. Figure 6.1 shows the results with a fine-grained ($e = 8$) redistribution. Recall that, with a fine-grained redistribution, DISK-PRE is no better than NO-PRE. The PB-PRE scheme provides higher bandwidth than PBX-PRE, showing that the extra work of exchanging prefetched data to simplify unpacking for compute nodes results in too few blocks being prefetched. The PB-PRE scheme manages 27% and 52% speedups over NO-PRE using I/O

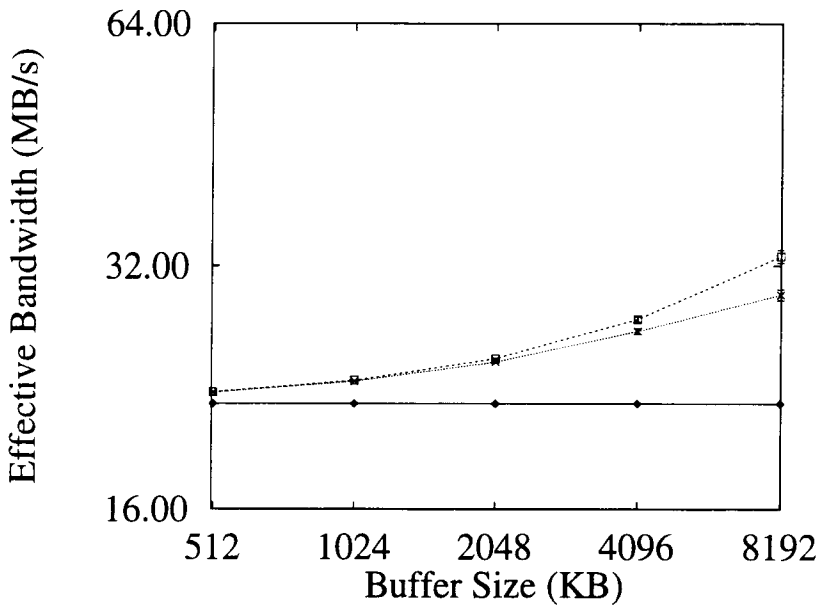
λ	17 μs
β	0.025 μs/byte
β_x	0.052 μs/byte
$\hat{\pi}_{BC}(8, P), P < 64$	0.159029 μs/byte
$\hat{\pi}_{BC}(256, P), P < 64$	0.063832 μs/byte
$\hat{\pi}_{BC}(8, 256)$	0.2803 μs/byte
$\hat{\pi}_{BC}(256, 256)$	0.06762 μs/byte
$\hat{\pi}^{IO}_{BC}(8)$	0.1474 μs/byte
$\hat{\pi}^{IO}_{BC}(256)$	0.04402 μs/byte
$\hat{v}_{mb}(e_v, n_v)$	$4.094/(e_v^{1.864}) + 0.0687\mu$s/byte
$\hat{v}^{IO}_{mb}(e_v, n_v)$	$4.094/(e_v^{1.864}) + 0.0503\mu$s/byte
D_1	13 ms
D_B	0.844 ms
B	8192 bytes

TABLE 6.2. Machine parameters used to generate the data in this section.

node buffer sizes of 4 MB and 8 MB, respectively. The results for a large-grained ($e = 256$) redistribution of a 50 MB file are shown in Figure 6.2. The relative bandwidth gains as the I/O node buffer size increases are similar to those for the fine-grained redistribution.

With a smaller file, the results are similar. Figures 6.3 and 6.4 show the performance of fine-grained and large-grained redistributions respectively reading a 10 MB file. In both cases the PBX-PRE falls short of the simpler PB-PRE prefetching scheme. The differences in bandwidth seen in Figure 6.4 are magnified by the short read time for the 10 MB file. Obviously, if most or all of a file is prefetched, as is the case in some runs here, the resulting bandwidth is very high.

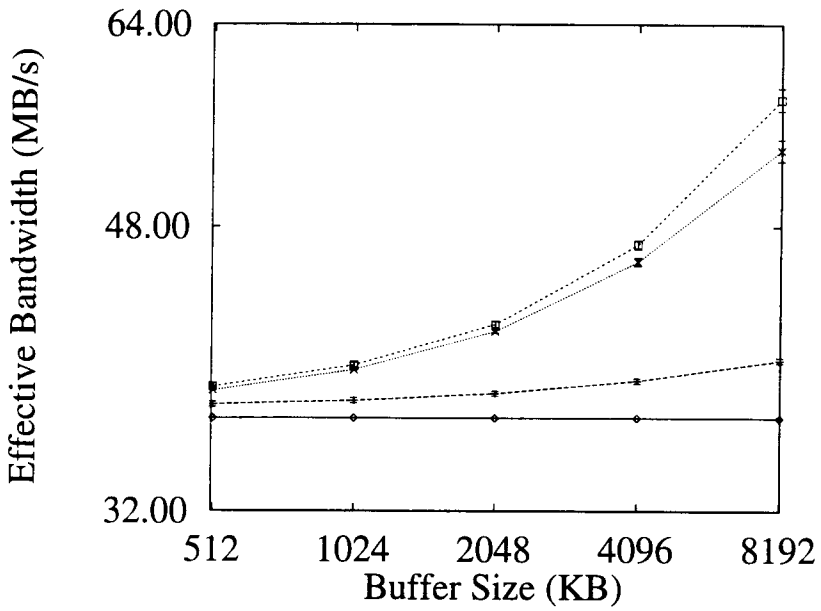
From the results presented so far, we can conclude that PB-PRE with prefetching can significantly increase the effective file system bandwidth, while PBX-PRE incurs too much communication overhead during prefetching to compete with PB-PRE. In the next section we vary our experimental parameters across several



50MB file, $P = 32, I = 4, e = 8$

		Prefetch Buffer Size				
Key	Method	512KB	1MB	2MB	4MB	8MB
◊	NO-PRE	21.6	21.6	21.6	21.6	21.6
+	DISK-PRE	21.6	21.6	21.6	21.6	21.6
□	PB-PRE	22.3	23.1	24.6	27.5	32.9
×	PBX-PRE	22.3	23.0	24.3	26.6	29.5

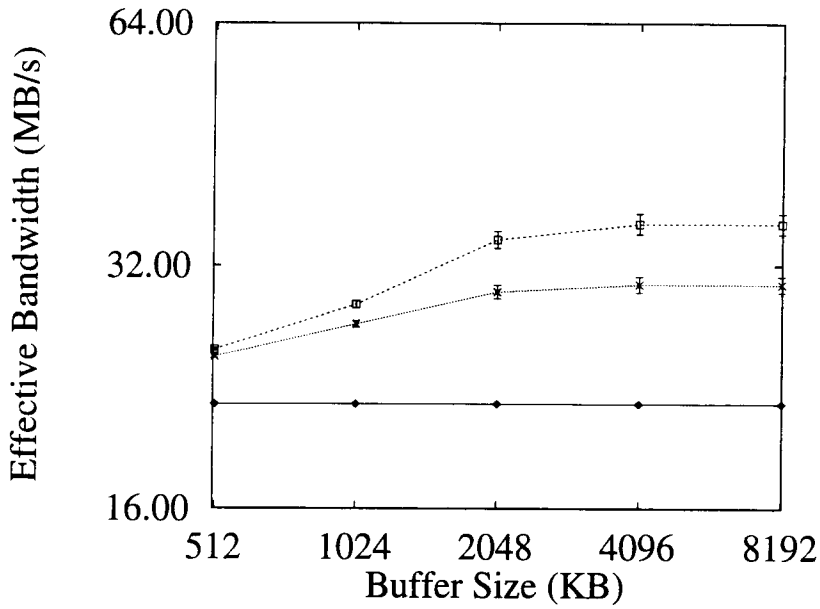
FIGURE 6.1. Effective bandwidths for varying prefetch buffer sizes using Meiko CS-2 parameters and random computation time for a fine-grained redistribution of a 50 MB file.



50MB file, $P = 32, I = 4, e = 256$

		Prefetch Buffer Size				
Key	Method	512KB	1MB	2MB	4MB	8MB
◊	NO-PRE	36.6	36.6	36.6	36.6	36.6
+	DISK-PRE	37.3	37.5	37.9	38.5	39.7
□	PB-PRE	38.2	39.4	41.8	46.8	57.5
×	PBX-PRE	38.0	39.1	41.3	45.7	53.5

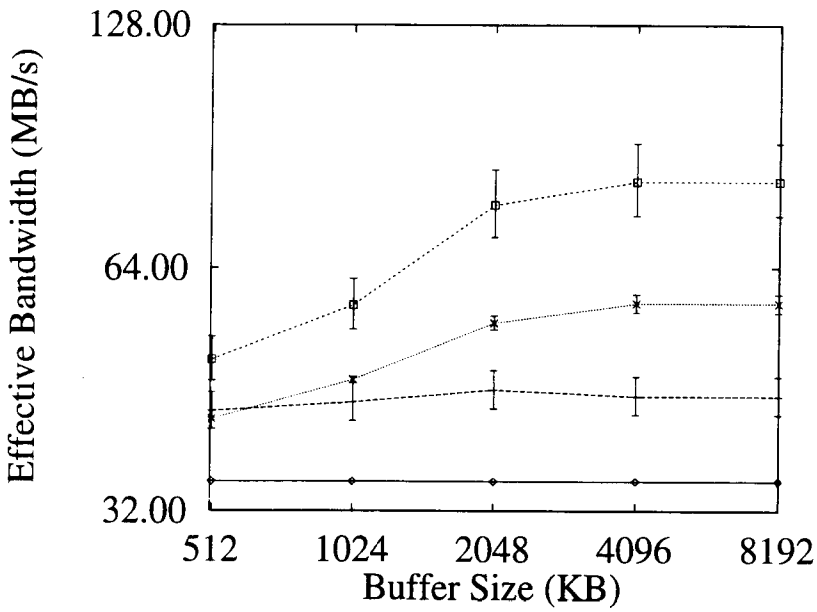
FIGURE 6.2. Effective bandwidths for varying prefetch buffer sizes using Meiko CS-2 parameters and random computation time for a large-grained redistribution of a 50 MB file.



10MB file, $P = 32, I = 4, e = 8$

		Prefetch Buffer Size				
Key	Method	512KB	1MB	2MB	4MB	8MB
◊	NO-PRE	21.5	21.5	21.5	21.5	21.5
+	DISK-PRE	21.5	21.5	21.5	21.5	21.5
□	PB-PRE	25.2	28.6	34.5	36.1	36.1
×	PBX-PRE	24.7	27.1	29.7	30.3	30.3

FIGURE 6.3. Effective bandwidths for varying prefetch buffer sizes using Meiko CS-2 parameters and random computation time for a fine-grained redistribution of a 10 MB file.



10MB file, $P = 32, I = 4, e = 256$

		Prefetch Buffer Size				
Key	Method	512KB	1MB	2MB	4MB	8MB
◊	NO-PRE	34.8	34.8	34.8	34.8	34.8
+	DISK-PRE	42.6	43.6	45.2	44.4	44.4
□	PB-PRE	49.3	57.6	76.6	81.9	81.9
×	PBX-PRE	41.6	46.5	54.8	57.9	57.9

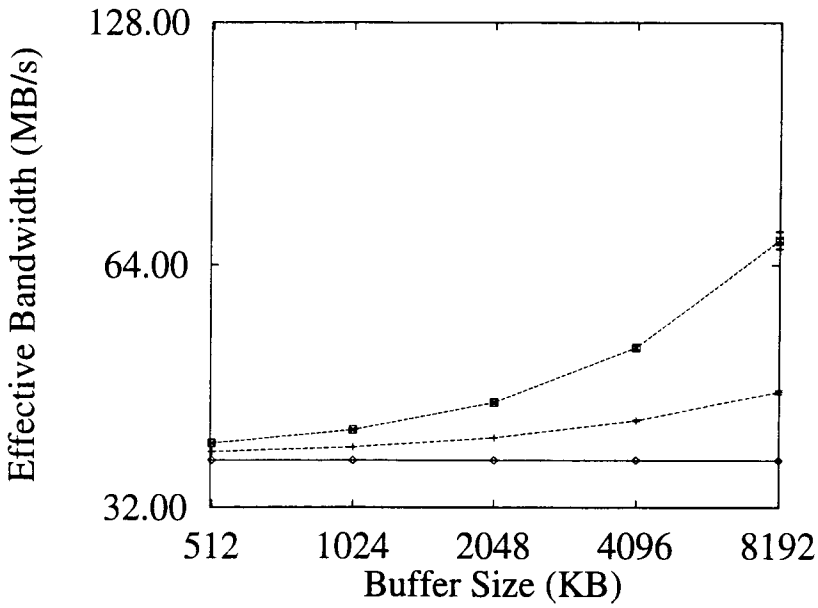
FIGURE 6.4. Effective bandwidths for varying prefetch buffer sizes using Meiko CS-2 parameters and random computation time for a large-grained redistribution of a 10 MB file.

orthogonal dimensions to see if this conclusion is valid in a broad set of circumstances.

6.3.3. Determining Robustness of Results

In the last chapter, we found that varying the communication overhead impacts the relative performance of file redistribution approaches. Clearly, a high message-passing overhead slows the PBX-PRE scheme. We perform here a comparison while modeling a low-latency machine like the DDIO machine (2 μs message latency, 600 MB/s bandwidth) described in the last chapter. With 8-byte elements, the result is qualitatively the same: PB-PRE outperforms PBX-PRE. Figure 6.5 shows a slightly different outcome when $\epsilon = 256$. With a large I/O node buffer (8 MB) and enough time to utilize it, PBX-PRE achieves bandwidth slightly above that of PB-PRE alone. The width of the confidence intervals in Figure 6.5, though, shows that the difference is statistically insignificant.

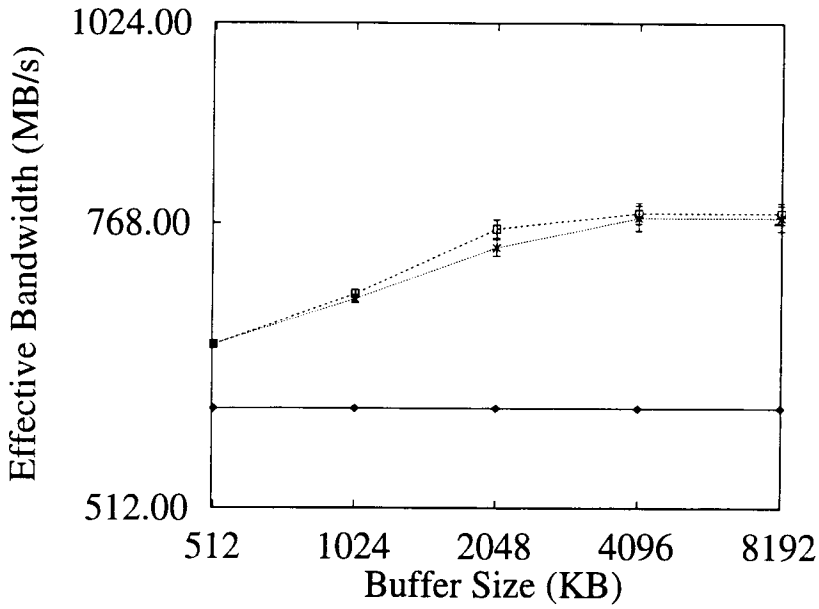
In the next experiments we combine two more configuration alternatives. First, we increase P and increase the file size to ensure that each I/O node contains a non-trivial amount of data. Second, we decrease P/I. This should negatively impact PB-PRE, since a P/I ratio closer to one means each compute node must unpack more data, reducing the chance that unpacking is overlapped with packing on I/O nodes. Despite this apparent disadvantage, Figures 6.6 and 6.7, in which $P = 256$ and $I = 128$, show that PB-PRE continues to outperform PBX-PRE. Interesting results are shown in Figure 6.7, which contains the large-grained redistribution results. There, the DISK-PRE bandwidth drops below the NO-PRE bandwidth. This is because MB-DDIO used by NO-PRE overlaps unpacking by compute nodes with packing and message-sending by I/O nodes. When most or



50MB file, $P = 32, I = 4, e = 256$

		Prefetch Buffer Size				
Key	Method	512KB	1MB	2MB	4MB	8MB
◊	NO-PRE	36.7	36.7	36.7	36.7	36.7
+	DISK-PRE	37.6	38.1	39.1	41.1	44.6
□	PB-PRE	38.5	40.0	43.3	50.6	68.6
×	PBX-PRE	38.5	40.0	43.3	50.6	68.8

FIGURE 6.5. Effective bandwidths for varying prefetch buffer sizes using DDIO machine (low latency, high bandwidth) parameters and random computation time for a large-grained redistribution of a 50 MB file.

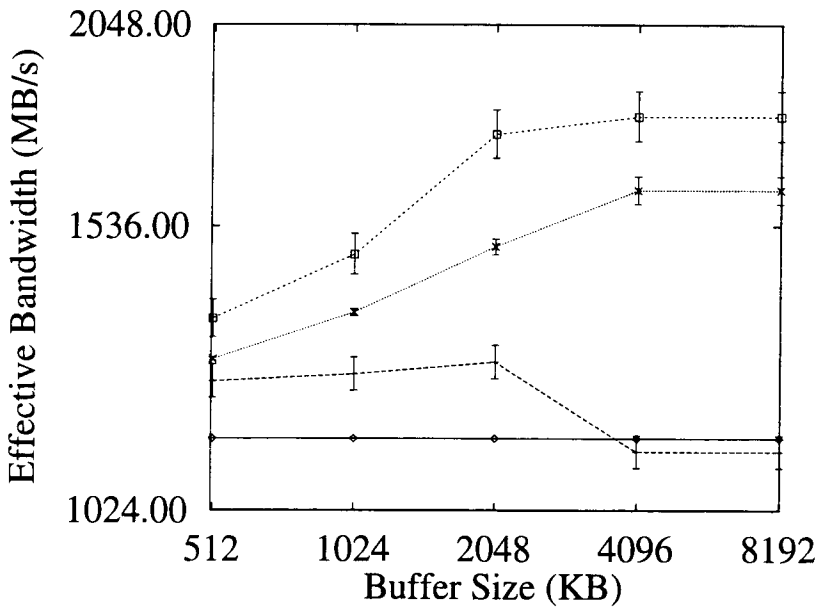


500MB file, $P = 256, I = 128, e = 8$

		Prefetch Buffer Size				
Key	Method	512KB	1MB	2MB	4MB	8MB
◊	NO-PRE	589.8	589.8	589.8	589.8	589.8
+	DISK-PRE	589.8	589.8	589.8	589.8	589.8
□	PB-PRE	646.0	694.5	761.8	779.1	779.1
×	PBX-PRE	646.4	689.6	741.7	773.8	773.8

FIGURE 6.6. Effective bandwidths for varying prefetch buffer sizes using Meiko CS-2 parameters and random computation time for a fine-grained redistribution of a 500 MB file.

all of the data are prefetched on I/O nodes, the DISK-PRE scheme requires that data be packetized on I/O nodes with no overlapping unpacking on compute nodes. Prefetched data are then unpacked on compute nodes, now with little or no overlap with I/O node functions. Further, the low P/I ratio means that the unbuffering is not divided among enough compute nodes to make its cost insignificant relative to the I/O node buffering cost.



500MB file, $P = 256, I = 128, e = 256$

		Prefetch Buffer Size				
Key	Method	512KB	1MB	2MB	4MB	8MB
◊	NO-PRE	1134.4	1134.4	1134.4	1134.4	1134.4
+	DISK-PRE	1230.4	1243.0	1264.2	1112.5	1112.5
□	PB-PRE	1345.8	1475.1	1751.1	1796.2	1796.2
×	PBX-PRE	1269.6	1357.6	1491.6	1616.4	1616.4

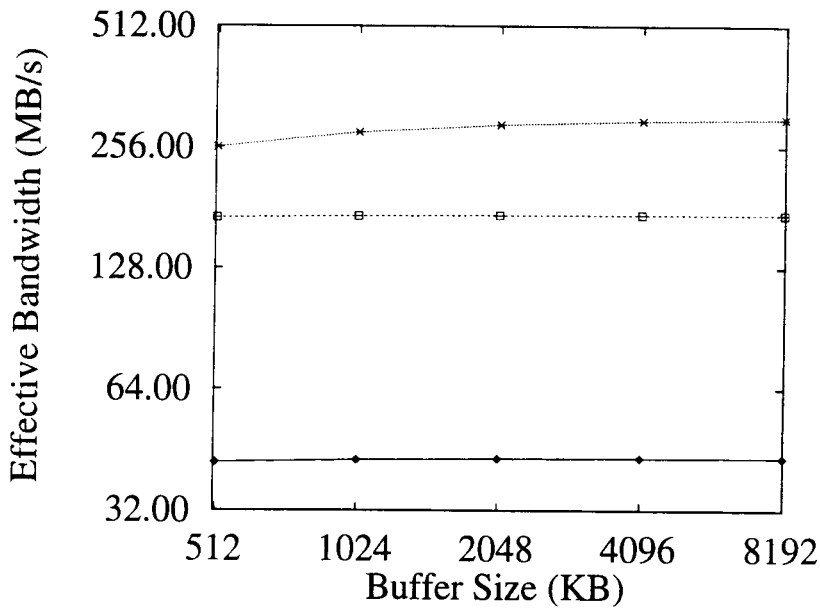
FIGURE 6.7. Effective bandwidths for varying prefetch buffer sizes using Meiko CS-2 parameters and random computation time when $P/I = 2$ for a large-grained redistribution of a 500 MB file.

In closing this section we conclude that, with the modeled workload, the PB-PRE scheme for prefetching and redistributing data is the desired choice. Across a variety of machine configurations, it outperforms all other schemes. The complexity of the PBX-PRE scheme is not justified. The simple PB-PRE scheme, in addition, outperformed the NO-PRE and DISK-PRE schemes for all machine configurations, buffer sizes, and grain sizes simulated.

6.4. Full Buffers for Optimal Prefetch

The previous experiments used an exponential workload distribution to compare the various prefetch schemes under "average" conditions. In this section we explore the best cases for the PB-PRE and PBX-PRE schemes — those in which computation allows enough time for the entire prefetch buffer to be filled for both schemes. In this case, because of its lower compute node unbuffering cost, the PBX-PRE scheme should outperform simple PB-PRE. Here we investigate the differences between the schemes under these optimistic conditions.

The first comparison we perform assumes that the entire file is prefetched. This is the most optimistic scenario, in which the difference in cost between PB-PRE and PBX-PRE will be most noticeable. Figure 6.8 shows the difference in bandwidth for a fine-grained redistribution when the file size is equal to I, in this case 4, times the prefetch buffer size. The fine-grained redistribution emphasizes the bandwidth difference between the PB-PRE and PBX-PRE schemes. The effective bandwidth difference is quite large, but this is due to the fact that the minimal amount of time taken for both operations magnifies the relative differences. The absolute times are shown in the second table below the figure, and we see the differences, in absolute terms, are quite small. The difference between PB-PRE and PBX-PRE



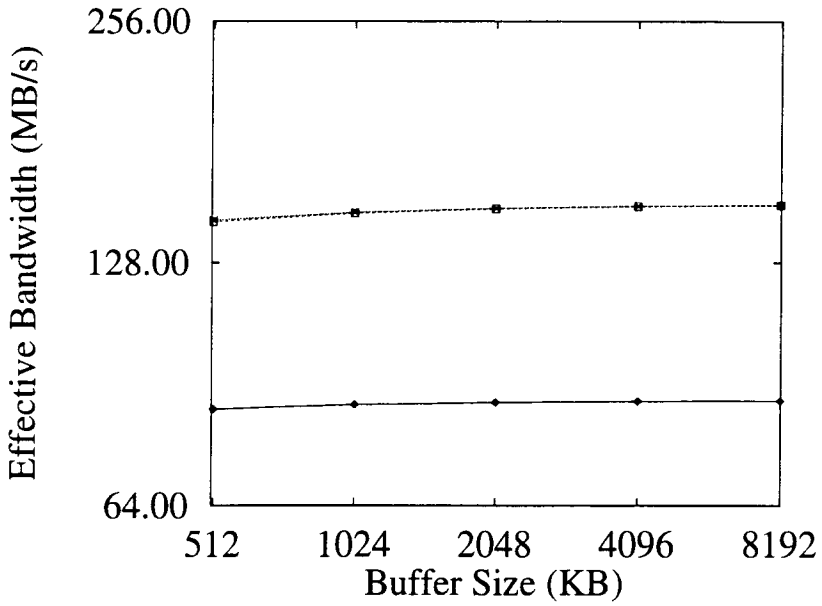
File size is $I\times$ Buffer Size, $P = 32, I = 8, e = 8$

		Prefetch Buffer Size				
Key	Method	512KB	1MB	2MB	4MB	8MB
◊	NO-PRE	42.1	42.6	42.9	43.1	43.1
+	DISK-PRE	42.1	42.6	42.9	43.1	43.1
□	PB-PRE	170.9	172.0	173.0	173.5	173.8
×	PBX-PRE	256.2	278.6	291.3	298.1	301.6

Actual times in ms

		Prefetch Buffer Size				
Key	Method	512KB	1MB	2MB	4MB	8MB
◊	NO-PRE	9.5	18.8	37.3	74.3	148.4
+	DISK-PRE	9.5	18.8	37.3	74.3	148.4
□	PB-PRE	2.3	4.7	9.2	18.4	36.8
×	PBX-PRE	1.6	2.9	5.5	10.7	21.2

FIGURE 6.8. Effective bandwidths for varying prefetch buffer sizes using Meiko CS-2 parameters and assuming the entire file is read with fine-grained packet-building into the prefetch buffers before being requested.



File size is Buffer Size $\times 32$, $P = 32, I = 16, e = 8$

		Prefetch Buffer Size				
Key	Method	512KB	1MB	2MB	4MB	8MB
◊	NO-PRE	84.2	85.3	85.8	86.1	86.3
+	DISK-PRE	84.2	85.3	85.8	86.1	86.3
□	PB-PRE	144.0	147.6	149.5	150.4	150.9
×	PBX-PRE	144.7	148.0	149.7	150.5	150.9

FIGURE 6.9. Effective bandwidths for varying prefetch buffer sizes using Meiko CS-2 parameters and assuming half the file is read with fine-grained packet-building into the prefetch buffers before being requested.

is smaller as the e increases and as the P/I ratio increases, because both of these changes decrease the time for unpacking performed at each compute node. Both of these schemes significantly outperform the NO-PRE and DISK-PRE schemes in this comparison.

A less optimistic comparison assumes that, though the prefetch buffers are filled, only a fraction of the file is actually prefetched. In Figure 6.9, we assume

half the file is held in the prefetch buffers when requested. The configuration shown performs a fine-grained redistribution on a system with a P/I ratio of two; recall that a lower ratio favors PBX-PRE. Even with this unrealistically low ratio and optimistically high percentage of the file prefetched, the simple PB-PRE scheme keeps pace with PBX-PRE. Note the significant gains in bandwidth for these schemes, nearly 70%, over the DISK-PRE scheme's bandwidth.

6.5. Conclusions

We have shown that, by adding *collective prefetching* to a file system like Disk-Directed I/O which supports collective operations, prefetching mistakes made by block-at-a-time file systems can be avoided. Simple prefetching of disk data provides little performance enhancement if the data must be redistributed en route to the compute nodes. When packet building is overlapped with disk reads of prefetched data for both fine-grained and large-grained redistributions, significant performance gains can be achieved. These gains depend directly on (1) the space available for prefetch buffers on I/O nodes, (2) the workload, which dictates the time available for prefetching, and (3) the predictability of read operations, which tends to be high for scientific applications. Our studies showed that doing extra work during the prefetching phase to eliminate unpacking of prefetched data by compute nodes rarely keeps up with the simpler I/O node packet-building scheme; it surpasses the simpler scheme even less often. The simpler PB-PRE scheme offers consistently high bandwidth and reduces the workload of the I/O nodes. This should be the choice of file system designers implementing collective prefetching.

7. CONCLUSIONS

7.1. Contributions and Significance

This research has addressed shortcomings of existing parallel I/O facilities at two levels—the language level and the file system level. At the parallel language level, we are the first to craft a flexible I/O interface matching the underlying abstraction of the language. At the same time, our implementation illustrates how such an interface can achieve high performance for common operations and good performance for more general operations. Three key principles have been illuminated by the implementation. First, virtual processor file operations, typically fine-grained by themselves, must be combined whenever possible into efficient large-scale file system calls. Second, machine-independent modes can support both high performance and generality while remaining relatively invisible to the user. Finally, the most commonly encountered file operations are performed using high-performance modes.

In our C* file system implementation, the high-performance modes have been crafted so they can take advantage of collective file operations and related optimizations. One of the most frequently needed optimizations is for redistribution of file data, with fine-grained redistributions being especially costly. Our results show that the cost of these redistributions can be reduced significantly by using collective operations and combining the Two-Phase Access Strategy with Disk-Directed I/O. This research has also identified the key parameters affecting both this hybrid scheme and those relying on packet building for redistribution by I/O nodes. These parameters include the array element size, ratio of compute nodes to I/O nodes, disk

system bandwidth, message-passing bandwidth, and processors' packetizing speed. Our validated model using these parameters allows file system designers and users to decide which approach to file redistribution best suits their needs.

An optimization found in many file systems is prefetching. This research has defined *collective prefetching*, which takes into account the global view of collective file operations when prefetching. Collective prefetching has two advantages over the shortsighted one-block read-ahead approach found in many parallel systems. Collective prefetching supports aggressive read-ahead of data in advance of bursty read requests, and it does not get confused by individual requests for data. For collective prefetching implementations, this research has shown that simple packet building used in conjunction with prefetching effectively increases the bandwidth seen by the user, but a more complicated strategy entailing communication between I/O nodes rarely outperforms the simpler scheme. As more and more file systems rely on collective operations, collective prefetching will supercede block-based prefetching schemes.

In supporting the models used in this research, a detailed model of redistribution costs has been developed. In particular, the model focuses on the most expensive part of the redistribution, packet-building and unpacking costs. The impacts of relevant architectural features such as translation lookaside buffer size and replacement policy, cache size, and compute node count are described qualitatively and mathematically. The model is accurate for a wide variety of redistributions and validated on three different architectures. It provides a solid foundation upon which other models, namely those for I/O node redistribution of data, can be built. This model also has more far-reaching significance in the parallel computing community at large, where redistributions play an important role in optimizing different algorithmic steps in parallel programs.

7.2. Future Directions

While our Stream* implementation has validated the notion of user-transparent, machine-independent modes, more tuning of the system can be done, especially for reading while in Independent Buffering mode. Reading in Independent Buffering mode can be especially inefficient when data must be redistributed, so file system techniques optimizing these inherently non-collective, fine-grained operations must be found.

Parallel languages other than C* and with paradigms other than data parallelism must be studied. We hope to generalize and update the design principles we found for C* for a broader domain of languages. These languages should be studied to find ways parallel I/O operations can be seamlessly included into the programming paradigm with support for high performance. For some languages we may have to conclude that *parallel* I/O does not make sense within the language. The ultimate goal is to ensure that language and environment designers consider parallel I/O as a major factor in the system design rather than as an afterthought.

Our file system models form a strong theoretical foundation for the types of optimizations which should be used in conjunction with Disk-Directed I/O. Further research combining an actual Disk-Directed file system with a production workload, preferably on a real parallel computer, will provide an ideal testbed for validating or suggesting modifications to these models. More research can also be done in applying these and related techniques to non-scientific (e.g., database) workloads.

In the context of array redistributions, this research has shown where significant redistribution costs come from (e.g., cache misses and TLB misses). With this understanding, cache- and TLB-efficient redistribution schemes, perhaps some requiring little auxiliary memory, should be buildable. Further, an objective study

which tests many redistribution schemes under both favorable and unfavorable cache and TLB conditions should be done to point out when each scheme is superior.

BIBLIOGRAPHY

[1] M. Arunachalam, A. Choudhary, and B. Rullman. Implementation and evaluation of prefetching in the Intel Paragon parallel file system. In *Proceedings of the Tenth International Parallel Processing Symposium*, April 1996.

[2] R. K. Asbury and D. S. Scott. FORTRAN I/O on the iPSC/2: Is there read after write? In *Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 129–132, 1989.

[3] M. J. Bach. *The Design of the Unix Operating System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[4] S. Batra, P. J. Hatcher, and R. Russell. The design and implementation of data-parallel files. In *Workshop on Modeling and Specification of I/O*, 1995. Publication via `http://www.cs.duke.edu/~dev/msio95`.

[5] R. Bennett, K. Bryant, A. Sussman, R. Das, and J. Saltz. Jovian: A framework for optimizing parallel I/O. In *Proceedings of the 1994 Scalable Parallel Libraries Conference*, pages 10–20. IEEE Computer Society Press, October 1994.

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

[7] R. Bordawekar, A. Choudhary, and J. M. D. Rosario. An experimental performance evaluation of Touchstone Delta Concurrent File System. In *International Conference on Supercomputing*, pages 367–376, 1993.

[8] R. Bordawekar, J. M. del Rosario, and A. Choudhary. Design and evaluation of primitives for parallel I/O. In *Proceedings of Supercomputing '93*, pages 452–461, 1993.

[9] P. Brezany, M. Gernt, P. Mehotra, and H. Zima. Concurrent file operations in a High Performance FORTRAN. In *Proceedings of Supercomputing '92*, pages 230–237, 1992.

[10] S. Chatterjee, J. R. Gilbert, F. J. E. Long, R. Schreiber, and S.-H. Teng. Generating local addresses and communication sets for data-parallel programs. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 149–158, May 1993.

[11] P. Chen and E. K. Lee. Striping in a RAID level 5 disk array. In *Proceedings of the 1995 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 136–145, May 1995.

[12] A. Choudhary, R. Bordawekar, M. Harry, R. Krishnaiyer, R. Ponnusamy, T. Singh, and R. Thakur. PASSION: parallel and scalable software for input-output. Technical Report SCCS-636, ECE Dept., NPAC and CASE Center, Syracuse University, September 1994.

[13] P. F. Corbett, S. J. Baylor, and D. G. Feitelson. Overview of the Vesta parallel file system. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 1–16, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 7–14.

[14] P. F. Corbett and D. G. Feitelson. Design and implementation of the Vesta parallel file system. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 63–70, 1994.

[15] P. F. Corbett, D. G. Feitelson, S. Fineberg, Y. Hsu, B. Nitzberg, J.-P. Prost, M. Snir, B. Traversat, and P. Wong. Overview of the MPI-IO parallel I/O interface. In *IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 1–15, April 1995.

[16] P. F. Corbett, D. G. Feitelson, Y. Hsu, J.-P. Prost, M. Snir, S. Fineberg, B. Nitzberg, B. Traversat, and P. Wong. MPI-IO: A parallel file I/O interface for MPI, November 1994. Version 0.3.

[17] P. F. Corbett, D. G. Feitelson, J.-P. Prost, and S. J. Baylor. Parallel access to files in the Vesta file system. In *Proceedings of Supercomputing '93*, pages 472–481, 1993.

[18] T. H. Cormen. Fast permuting on disk arrays. *Journal of Parallel and Distributed Computing*, 17(1–2):41–57, January and February 1993.

[19] T. H. Cormen and L. F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. In *Proceedings of the Fifth Symposium on Parallel Algorithms and Architectures*, pages 130–139, June 1993.

[20] P. E. Crandall, R. A. Aydt, A. A. Chien, and D. A. Reed. Input/output characteristics of scalable parallel applications. In *Proceedings of Supercomputing '95*, December 1995.

[21] T. W. Crockett. File concepts for parallel I/O. In *Proceedings of Supercomputing '89*, pages 574–579, 1989.

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

[23] E. DeBenedictis and J. M. del Rosario. nCUBE parallel I/O software. In *Eleventh Annual IEEE International Phoenix Conference on Computers and Communications (IPCCC)*, pages 0117–0124, April 1992.

[24] E. DeBenedictis and P. Madams. nCUBE's parallel I/O with Unix capability. In *Sixth Annual Distributed-Memory Computer Conference*, pages 270–277, 1991.

[25] E. P. DeBenedictis and J. M. del Rosario. Modular scalable I/O. *Journal of Parallel and Distributed Computing*, 17(1–2):122–128, January and February 1993.

[26] E. DeLano, W. Walker, J. Yetter, and M. Forsyth. A high speed superscalar PA-RISC processor. In *Proceedings of the COMPCON Spring 1992, Digest of Papers*, pages 116–121, Feb 1992.

[27] J. M. del Rosario, R. Bordawekar, and A. Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 56–70, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 31–38.

[28] J. M. del Rosario, M. Harry, and A. Choudhary. The design of VIP-FS: A virtual, parallel file system for high performance parallel and distributed computing. Technical Report SCCS-628, NPAC, Syracuse, NY 13244, May 1994.

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

[30] Digital Equipment Corporation. Alpha AXP workstation and server specification summary, Oct 1993. URL: `ftp://ftp.digital.com/pub/Digital/info/misc/axp-ws-summary.ps`.

[31] S. A. Fineberg. Implementing the NHT-1 application I/O benchmark. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 37–55, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 23–30.

[32] R. J. Flynn and H. Hadimioglu. A distributed hypercube file system. In *Third Conference on Hypercube Concurrent Computers and Applications*, pages 1375–1381, 1988.

[33] C. S. Freedman, J. Burger, and D. J. Dewitt. SPIFFI — a scalable parallel file system for the Intel Paragon. Submitted to IEEE TPDS, 1994.

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

[35] F. N. Fritsch, R. E. Schafer, and W. P. Crowley. Solution of the transcendental equation $we^w = x$. *Communications of the ACM*, 16(2):123–124, Feb 1973.

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

[37] G. A. Gibson, R. H. Patterson, and M. Satyanarayanan. Disk reads with DRAM latency. In *Third Workshop on Workstation Operating Systems*, pages 126–131, 1992.

[38] G. A. Gibson, D. Stodolsky, P. W. Chang, W. V. Courtwright II, C. G. Demetriou, E. Ginting, M. Holland, Q. Ma, L. Neal, R. H. Patterson, J. Su, R. Youssef, and J. Zelenka. The Scotch parallel storage systems. In *Proceedings of 40th IEEE Computer Society International Conference (COMPCON 95)*, pages 403–410, San Francisco, Spring 1995.

[39] S. Gupta, S. D. Kaushik, S. Mufti, S. Sharma, C.-H. Huang, and P. Sadayappan. On the generation of efficient data communications for distributed-memory machines. In *Proceedings of the International Computing Symposium*, pages I:504–513, 1992.

[40] P. J. Hatcher. Elemental functions. Technical Report X3J11.1/92-076, Numerical C Extensions Group (ANSI X3J11.1), 1992.

[41] P. J. Hatcher. Extending C* for data-parallel I/O. Technical Report TR 94-16, University of New Hampshire Department of Computer Science, 1994.

[42] P. J. Hatcher and M. J. Quinn. *Data-Parallel Programming on MIMD Computers*. The MIT Press, Cambridge, Massachusetts, 1991.

[43] Hewlett Packard Corporation. PA-RISC 1.1 architecture and instruction set reference manual, Feb 1994. Hewlett-Packard part no. 09740-90039. URL: `http://www1.hp.com:80/cgi-bin/wmSendData/nsa/acd.ps.Z`.

[44] J. Huber, C. L. Elford, D. A. Reed, A. A. Chien, and D. S. Blumenthal. PPFS: A high performance portable parallel file system. Technical Report UIUCDCS-R-95-1903, University of Illinois at Urbana Champaign, January 1995.

[45] Intel. *Paragon OSF/1 User's Guide*, 1993.

[46] S. Johnsson and C.-T. Ho. The complexity of reshaping arrays on boolean cubes. In *Proceedings of the Fifth Distributed Memory Computing Conference*, pages I:370–377, 1990.

[47] E. T. Kalns and L. M. Ni. DaRel: A portable data redistribution library for distributed-memory machines. In *Proceedings of the 1994 Scalable Parallel Libraries Conference II*, Oct 1994.

[48] E. T. Kalns and L. M. Ni. Processor mapping techniques toward efficient data redistribution. In *Proceedings of the Eighth International Parallel Processing Symposium*, pages 469–476, Apr 1994.

[49] S. Kaushik, C.-H. Huang, R. W. Johnson, and P. Sadayappan. An approach to communication-efficient data redistribution. In *Proceedings of the 8th ACM International Conference on Supercomputing*, pages 364–373, July 1994.

[50] S. Kaushik, C.-H. Huang, J. Ramanujam, and P. Sadayappan. Multi-phase array redistribution: Modeling and evaluation. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 441–445, April 1995.

[51] D. Kotz. Multiprocessor file system interfaces. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, pages 194–201, 1993.

[52] D. Kotz. Disk-directed I/O for MIMD multiprocessors. Technical Report PCS-TR94-226, Dartmouth College Department of Computer Science, July 1994.

[53] D. Kotz. Personal communication, April 1995.

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

[55] D. Kotz and C. S. Ellis. Practical prefetching techniques for multiprocessor file systems. *Journal of Distributed and Parallel Databases*, 1(1):33–51, January 1993.

[56] D. Kotz and N. Nieuwejaar. Dynamic file-access characteristics of a production parallel scientific workload. In *Proceedings of Supercomputing '94*, pages 640–649, November 1994.

[57] D. Kotz, S. B. Toh, and S. Radhakrishnan. A detailed simulation model of the hp 97560 disk drive. Technical Report PCS-TR94-220, Dartmouth College Department of Computer Science, July 1994.

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

[59] K. Kunchithapadam and B. P. Miller. Optimizing array distributions in data-parallel programs. In *Proceedings of the Seventh International Workshop on Languages and Compilers for Parallel Computing*, pages 470–484, August 1995.

[60] A. J. Lapadula, K. P. Herold, and P. J. Hatcher. A retargetable C* compiler and run-time library for mesh-connected MIMD computers. Technical Report TR 92-15, University of New Hampshire Department of Computer Science, 1992.

[61] J. S. Lee, S. Ranka, and R. V. Shankar. Communication-efficient and memory-bounded external redistribution. Technical report, Computer Science Department, Syracuse University, February 1995.

[62] S. J. LoVerso, M. Isman, A. Nanopoulos, W. Nesheim, E. D. Milne, and R. Wheeler. *sfs*: A parallel file system for the CM-5. In *Proceedings of the 1993 Summer USENIX Conference*, pages 291–305, 1993.

[63] E. L. Miller and R. H. Katz. Input/output behavior of supercomputing applications. In *Proceedings of Supercomputing '91*, pages 567–576, 1991.

[64] J. A. Moore. Parallel I/O requirements of four oceanography applications. Technical Report 95-80-1, Oregon State University Department of Computer Science, 1995.

[65] J. A. Moore, P. J. Hatcher, and M. J. Quinn. Stream*: Fast, flexible data-parallel I/O. In *Parallel Computing '95*, September 1995.

[66] J. A. Moore and M. J. Quinn. Analysis and modeling of array redistributions. Technical Report 96-80-1, Oregon State University Department of Computer Science, 1996.

[67] S. A. Moyer and V. S. Sunderam. PIOUS: a scalable parallel I/O system for distributed computing environments. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 71–78, 1994.

[68] J. R. Nickolls. The MasPar scalable Unix I/O system. In *Proceedings of the Eighth International Parallel Processing Symposium*, pages 390–395, 1994.

[69] N. Nieuwejaar and D. Kotz. A multiprocessor extension to the conventional file system interface. Technical Report PCS-TR94-230, Dept. of Computer Science, Dartmouth College, September 1994.

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

[71] B. K. Pasquale and G. C. Polyzos. A static analysis of I/O characteristics of scientific applications in a production workload. In *Proceedings of Supercomputing '93*, pages 388–396, November 1993.

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

[73] R. H. Patterson and G. A. Gibson. Exposing I/O concurrency with informed prefetching. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, pages 7–16, September 1994.

[74] R. H. Patterson, G. A. Gibson, and M. Satyanarayanan. Informed prefetching: Converting high throughput to low latency. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 41–55, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies.

[75] R. H. Patterson, G. A. Gibson, and M. Satyanarayanan. A status report on research in transparent informed prefetching. *ACM Operating Systems Review*, 27(2):21–34, April 1993.

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

[77] A. Purakayastha, C. S. Ellis, D. Kotz, N. Nieuwejaar, and M. Best. Characterizing parallel file-access patterns on a large-scale multiprocessor. In *Proceedings of the Ninth International Parallel Processing Symposium*, pages 165–172, April 1995.

[78] S. Ramaswamy and P. Banerjee. Automatic generation of efficient array redistribution routines for distributed memory multicomputers. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 342–349, February 1995.

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

[80] D. A. Reed, C. Catlett, A. Choudhary, D. Kotz, and M. Snir. Parallel I/O: Getting ready for prime time. *IEEE Computer Society Technical Committee on Computer Architecture Newsletter, Winter 1994-95*, pages 45–55, 1994. Edited transcript of panel discussion at the 1994 International Conference on Parallel Processing, August 1994.

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

[82] Seagate Technology. Seagate storage products: Barracuda 4lp family, January 1996. Available at URL `http://www.seagate.com`.

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

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

[85] T. P. Singh and A. Choudhary. ADOPT: A Dynamic scheme for Optimal PrefeTching in parallel file systems. Technical Report SCCS-627, Northeast Parallel Architectures Center, Syracuse University, 1994.

[86] A. J. Smith. Disk cache — miss ratio analysis and design considerations. *ACM Transactions on Computer Systems*, 3(3):161–203, 1985.

[87] Sun Microsystems. UltraSPARC programmer reference manual, 1995.

[88] R. Thakur, R. Bordawekar, A. Choudhary, R. Ponnusamy, and T. Singh. PASSION runtime library for parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 119–128, October 1994.

[89] R. Thakur and A. Choudhary. All-to-all communication on meshes with wormhole routing. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 561–565, April 1994.

[90] R. Thakur and A. Choudhary. Efficient algorithms for array redistribution. Technical Report SCCS-601, Northeast Parallel Architectures Center, Syracuse University, June 1994.

[91] R. Thakur, A. Choudhary, and G. Fox. Runtime array redistribution in HPF programs. In *Proceedings of Scalable High-Performance Computing Conference*, pages 309–316, 1994.

[92] Thinking Machines Corporation. *C* Programming Guide*, June 1991.

[93] Thinking Machines Corporation. *Connection Machine I/O System Programming Guide*, October 1991.

[94] Thinking Machines Corporation. *CM-5 I/O System Programming Guide*, September 1993.

[95] D. E. Vengroff. A transparent parallel I/O environment. In *Proceedings of the 1994 DAGS/PC Symposium*, pages 117–134, Hanover, NH, July 1994. Dartmouth Institute for Advanced Graduate Studies.

[96] J. S. Vitter and P. Krishnan. Optimal prefetching via data compression. In *Foundations of Computer Science*, pages 121–130, 1991.

[97] A. Wakatani and M. Wolfe. A new approach to array redistribution: Strip mining redistribution. In *Parallel Architectures and Languages Europe (PARLE 94)*, July 1994.

[98] A. Wakatani and M. Wolfe. Optimization of array redistribution for distributed memory multicomputers. *Parallel Computing*, 21, 1995.

[99] L. F. Wisniewski. Structured permuting in place on parallel disk systems. In *Proceedings of the Fourth IOPADS Workshop on I/O in Parallel and Distributed Systems*, May 1996.

[100] A. Witkowski, K. Chandrakumar, and G. Macchio. Concurrent I/O system for the hypercube multiprocessor. In *Third Conference on Hypercube Concurrent Computers and Applications*, pages 1398–1407, 1988.