The Galley Parallel File System

Nils Nieuwejaar David Kotz

PCS-TR96-286
Department of Computer Science
Dartmouth College, Hanover, NH 03755-3510
{nils,dfk}@cs.dartmouth.edu

Abstract

Most current multiprocessor file systems are designed to use multiple disks in parallel, using the high aggregate bandwidth to meet the growing I/O requirements of parallel scientific applications. Many multiprocessor file systems provide applications with a conventional Unix-like interface, allowing the application to access multiple disks transparently. This interface conceals the parallelism within the file system, increasing the ease of programmability, but making it difficult or impossible for sophisticated programmers and libraries to use knowledge about their I/O needs to exploit that parallelism. In addition to providing an insufficient interface, most current multiprocessor file systems are optimized for a different workload than they are being asked to support. We introduce Galley, a new parallel file system that is intended to efficiently support realistic scientific multiprocessor workloads. We discuss Galley's file structure and application interface, as well as the performance advantages offered by that interface.

1 Introduction

While the speed of most components of massively parallel computers have been steadily increasing for years, the I/O subsystem has not been keeping pace. Hardware limitations are one reason for the difference in the rates of performance increase, but the slow development of new multiprocessor file systems is also to blame. One of the primary reasons that multiprocessor file-system performance has not improved at the same rate as other aspects of multiprocessors is that, until recently, there has been limited information available about how applications were using existing multiprocessor file systems and how programmers would like to use future file systems.

Several recent analyses of production file-system workloads on multiprocessors running primarily scientific applications show that many of the assumptions that guided the development of most multiprocessor file systems were incorrect [KN94, NK96a, PEK+95]. It was generally assumed that scientific applications designed to run on a multiprocessor would behave in the same fashion as scientific applications designed to run on sequential and vector supercomputers: accessing large files in large, consecutive chunks [Pie89, PFDJ89, LIN+93, MK91]. Studies of two different multiprocessor file-system workloads,

This research was funded by NSF under grant number CCR-9404919 and by NASA Ames under agreement numbers NCC 2-849 and NAG 2-936.

Copyright 1996 by the authors

running a variety of applications in a variety of scientific domains, on two architectures, under both data-parallel and control-parallel programming models, show that many applications make many small, regular, but non-consecutive requests to the file system [NKP+95]. These studies suggest that the workload that most multiprocessor file systems were optimized for is very different than the workloads they are actually being asked to serve.

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

The remainder of this paper is organized as follows. In Section 2 we describe the specific goals Galley was designed to satisfy. In Section 3 we discuss a new, three-dimensional way to structure files in a multiprocessor file system. Section 4 describes the design and current implementation of Galley. Section 5 discusses the interface available to applications that intend to use Galley, and Section 6 shows how Galley's interface can improve an application's performance. In Section 7 we discuss several other multiprocessor file systems, and finally in Section 8 we summarize and describe our future plans.

2 Design Goals

Most current multiprocessor file systems designs are based primarily on hypotheses about how parallel scientific applications would use a file system. Galley's design is the result of examining how parallel scientific applications actually use existing file systems. Accordingly, Galley is designed to satisfy several goals:

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

  Galley is targeted at distributed memory, MIMD machines such as IBM's SP-2 or Intel's Paragon.

3 File Structure

Most existing multiprocessor file systems are based on a Unix-like model [BGST93, Pie89, LIN+93]. Under this model, a file is seen as an addressable, linear sequence of bytes. Applications can issue requests to read or write data contiguous subranges of that sequence of bytes. A parallel file system typically *declusters* files (i.e., scatters the blocks of each file across multiple disks), allowing parallel access to the file. This parallel access reduces the effect of the bottleneck imposed by the relatively slow disk speed. Although the file is actually scattered across many disks, the underlying parallel structure of the file is hidden from the application.

Galley uses a more complex file model that allows greater flexibility, which should lead to higher performance.

3.1 Subfiles

The linear file model offered by most multiprocessor file systems can give good performance when the request size generated by the application is larger than the declustering unit size, as a single request will involve data from multiple disks. Under these conditions, the file system can access multiple disks in parallel, delivering higher bandwidth to the application, and possibly hiding any latency caused by disk seeks. The drawback of this approach is that most multiprocessor file systems use a declustering unit size measured in kilobytes (e.g., 4 KB in Intel's CFS [Pie89]), but our workload characterization studies show that the typical request size in a parallel application is much smaller: frequently under 200 bytes [NKP+95]. This disparity between the request size and the declustering unit size means that most of the individual requests generated by parallel applications are not being executed in parallel. In the worst case, the compute processors in a parallel application may issue their requests in such a way that all of an application's processes may first attempt to access disk 0 simultaneously, then all attempt to access disk 1 simultaneously, and so on.

Another drawback of the linear file model is that a dataset may have an efficient, parallel mapping onto multiple disks that is not easily captured by the standard declustering scheme. One such example is the two-dimensional, cyclically-shifted block layout scheme for matrices, shown in Figure 1, which was designed for SOLAR, a portable, out-of-core linear-algebra library [TG96]. This data layout is intended to efficiently support a wide variety of out-of-core algorithms. In particular, it allows blocks of rows and columns to be transferred efficiently, as well as square or nearly-square submatrices.

To avoid the limitations of the linear file model, Galley does not impose a declustering strategy on an application's data. Instead, Galley provides applications with the ability to fully control this declustering according to their own needs. This control is particularly important when implementing *I/O-optimal algorithms* [CK93]. Applications are also able to explicitly indicate which disk they wish to access in each request. To allow this behavior, files are composed of one or more *subfiles*, which may be directly addressed by the application. Each subfile resides entirely on a single disk, and no disk contains more

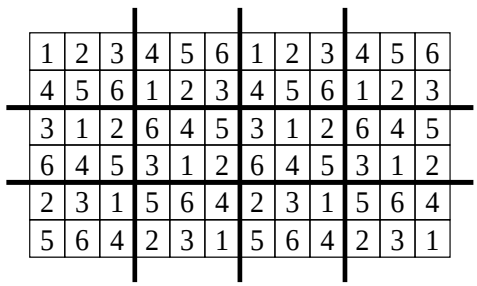

Figure 1: An example of a 2-dimensional, cyclically-shifted block layout, as described in [TG96]. In this example there are 6 disks, logically arranged into a 2-by-3 grid, and a 6-by-12 block matrix. The number in each square indicates the disk on which that block is stored.

than one subfile from any file. The application may choose how many subfiles a file contains when the file is created. The number of subfiles remains fixed throughout the life of the file.

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

3.2 Forks

Each subfile in Galley is structured as a collection of one or more independent *forks*. A fork is a named, addressable, linear sequence of bytes, similar to a traditional Unix file. Unlike the number of subfiles in a file, the number of forks in a subfile is not fixed; libraries and applications may add forks to, or remove forks from, a subfile at any time. The final, three-dimensional file structure is illustrated in Figure 2. There is no requirement that all subfiles have the same number of forks, or that all forks have the same size.

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

While typical application programmers may find forks helpful, they are most likely to be useful when implementing libraries. In addition to storing data in the traditional sense, many libraries also need to store persistent, library-specific 'metadata' independently of the data proper. One example of such a library would be a compression library similar to that described in [SW95], which compresses a data file in multiple independent chunks. Such a library could store the compressed data chunks in one fork and index information in another.

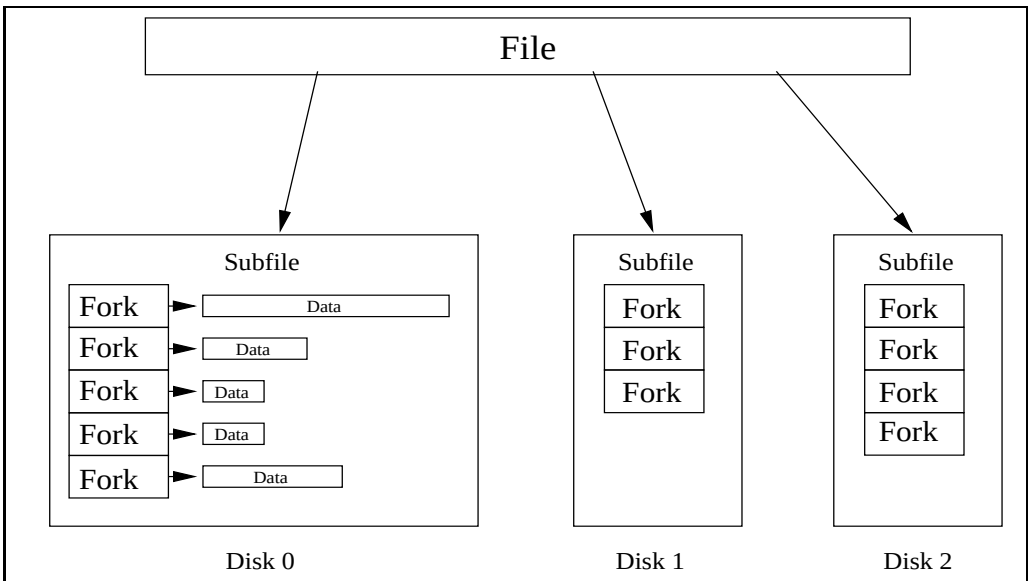

Figure 2: Three dimensional structure of files in the Galley File System. The portion of the file residing on disk 0 is shown in greater detail than the portions on the other two disks.

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

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

4 System Structure

The Galley parallel file system is structured as a set of clients and servers. This model is based on the typical multiprocessor architecture that dedicates some processors to computation and dedicates the rest to I/O. In this system, the *Compute Processors* (CPs) function as clients and the *I/O Processors* (IOPs) act as servers.

4.1 Compute Processors

A client in Galley is simply any user application that has been linked with the Galley run-time library, and which runs on a compute processor. The run-time library receives file-system requests from the application, translates them into lower-level requests, and passes them (as messages) directly to the appropriate servers, running on I/O processors. The run-time library then handles the transfer of data between the I/O processors and the compute node's memory.

As far as Galley is concerned, every compute processor in an application is completely independent of every other compute processor. Indeed, Galley does not even assume that one compute processor is even aware of the existence of other compute processors. This independence means that Galley does not impose any communication requirements on a user's application. As a result, applications may use whichever communication software (e.g., MPI, PVM, P4) is most suitable to the given problem.

Like most multiprocessor file systems, Galley offers both blocking and non-blocking I/O. To simplify the implementation, and to avoid binding Galley too tightly to a single architecture, Galley originally used multithreading to implement non-blocking I/O. Unfortunately, most of the major communications packages cannot function in a multithreaded environment. As a result, Galley is currently forced to use signals to implement non-blocking I/O, using a TCP/IP communications substrate. If support for multithreaded environments ever becomes commonplace in message-passing packages, we will reexamine this decision.

Although applications may interact directly with Galley's interface, we expect that most applications will use a higher-level library or language layered on top of the Galley run-time library. One such library implements a Unix-like file model, which should reduce the effort required to port legacy applications to Galley [Nie96]. Other libraries currently being implemented provide Panda [SCJ+95] and Vesta [CFP+95] interfaces, as well as ViC*, a variant of C* designed for out-of-core computations [CC94].

4.2 I/O Processors

Galley's I/O servers, illustrated in Figure 3, are composed of several units, which are described in detail below. Each unit is implemented as a separate thread. Furthermore, each IOP also has one thread designated to handle incoming I/O requests for each compute processor. When an IOP receives a request from a CP, the appropriate CP thread interprets the request, passes it on to the appropriate worker thread, and then handles the transfer of data between the IOP and the CP. This multithreading makes it easy for an IOP to service requests from many clients simultaneously.

While one potential concern is that this thread-per-CP design may limit the scalability of the system, we have not observed such a limitation in the performance tests shown in Section 6. One may reasonably assume that a thread that is idle (i.e., not actively handling a request) is not likely to noticeably affect the performance of an IOP. By the time the number of active threads on a single IOP becomes great enough to hinder performance, the IOP will most likely be overloaded at the disk, the network interface,

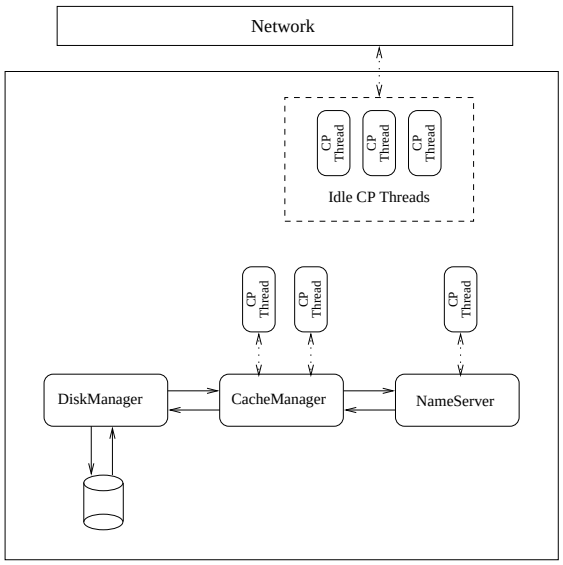

Figure 3: Internal structure of a Galley I/O Processor, showing two active data requests waiting for the CacheManager, one active metadata request waiting for the NameServer, and three idle CP threads.

or the buffer cache, and the effect of the number of threads will be minor relative to these other factors. We intend to explore this issue further as we port Galley to different architectures, which may offer different levels of thread support.

Galley's metadata (not to be confused with a user-level library's 'metadata' discussed above) is distributed across all IOPs, so there is no single point of contention that could limit scalability. Thus, each IOP acts both as a data server and as a metadata server. When a request arrives for a metadata operation (e.g., file open, close, delete), the CP's thread hands the request on to the NameServer, waits for the NameServer to complete the operation, and then passes the result back to the requesting CP. For most operations, the NameServer will need to submit a request to the CacheManager for data stored on disk.

4.2.1 CP Threads

CP threads remain idle until a request arrives from the corresponding CP. After being awakened to service a new data-access request, a CP thread creates a list of all the disk blocks[1] that will be required to satisfy the request. The CP thread then passes the full list of blocks to the CacheManager, and waits on a queue of buffers returned by the CacheManager and DiskManager. As a CP thread receives buffers on its queue, it handles the transfer of data between its CP and those buffers. When a CP thread completes the transfer of data to or from a buffer, it decreases that buffer's reference count, and handles the next buffer in the queue. When the whole request has been satisfied, or if it fails in the middle, the

[1] The current implementation of Galley uses a logical disk-block size of 32 KB.

thread passes a success or failure message back to its CP, and idles until another request comes in.

The order in which a fork's blocks are placed on the CP thread's buffer queue is determined by which blocks are present in the buffer cache and the order in which that fork's blocks are laid out on disk. As a result, it is not possible for Galley's client-side run-time library to know in advance in which order an IOP will satisfy the individual pieces of a request. When writing, this approach is somewhat unusual in that the IOP is essentially 'pulling' the data from the CP, rather than the traditional model, where the CP 'pushes' the data to the IOP.

4.2.2 CacheManager

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

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

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

4.2.3 DiskManager

The DiskManager is responsible for actually reading data from and writing data to disk. The DiskManager maintains a list of blocks that the CacheManager has requested to be read or written. As new requests arrive from the CacheManager, they are placed into the list according to the disk scheduling algorithm. The DiskManager currently uses a Cyclical Scan algorithm [SCO90]. When a block has been read from disk, the DiskManager updates the cache status of that block's buffer from 'not ready' to 'ready', increases its reference count, and adds it to the requesting thread's ready queue.

Galley's DiskManager does not attempt to prefetch data for two reasons. First, indiscriminate prefetching can cause thrashing in the buffer cache [Nit92]. Second, prefetching is based on the assump-

tion that the system can intelligently guess what an application is going to request next. Using the higher-level requests described below, there is frequently no need for Galley to make guesses about an application's behavior; the application is able to explicitly provide that information to each IOP.

To increase portability, Galley does not use a system-specific low-level driver to directly access the disk. Instead, Galley relies on the underlying system (presumably Unix) to provide such services. Galley's DiskManager has been implemented to use raw devices, Unix files, or simulated devices as "disks". Galley's disk-handling primitives are sufficiently simple that modifying the DiskManager to access a device directly through a low-level device driver is likely to be a trivial task.

5 Application Interface

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

5.1 File Operations

Files in Galley are created using the `gfs_create_file()` call. In addition to specifying a file name, an application may specify on how many IOPs, and even on which IOPs, the file is to be created. A `gfs_create_file` call is completed in three steps. The first step is to verify that the name chosen for the file is not already in use, and to *reserve* the name if it is available. This step requires that a single message be sent to the IOP that will be responsible for maintaining the metadata for the new file. The responsible IOP is chosen by applying a simple hash function to the file name. Vesta uses a similar scheme [CFP+95]. The second step is to create subfiles on each of the appropriate IOPs. This step requires that a message be sent to each IOP, asking that a *subfile-header block* be assigned to the file. Like an inode in a Unix file system, a subfile-header block contains all the metadata information for that subfile. Unlike the Unix practice of statically creating inodes, however, any block in the file system may become a subfile-header block. Each IOP returns either the ID of the assigned header block, or an error code. If this step fails on any IOP (e.g., if it is out of disk space), then each IOP is instructed to release the newly assigned header blocks, the reserved file name is released, and the appropriate error code is returned to the application. The final step of a successful file-creation process is to store the file name, along with all the subfile-header block IDs, on disk at the responsible IOP and to return a success code to the application. Note that after the file is created, all the subfiles are empty; that is, no forks are created as part of the file-creation process.

As far as Galley is concerned, each compute node in an application is a completely independent entity. Therefore, Galley has no notion of a *leader*, a node that can issue requests on behalf of other

processors. As a result, each node in an application that wishes to use a file in Galley must explicitly open that file using the `gfs_open_file()` call. When an application issues a `gfs_open_file()` call, the run-time library sends a request to the appropriate metadata server (again, determined by hashing the file name). If the file exists, the metadata server returns a list of all the subfile-header block IDs to the requesting CP. The run-time library assigns the open file a *file ID*, and caches the list of header block IDs in an *open-file table* to avoid repeated requests to the metadata server. Since these IDs do not change during the course of the file's lifetime, we do not have to be concerned that the cached IDs will become inconsistent with the IDs stored at the metadata server. The run-time library then sends messages to each of the IOPs on which the file has a subfile, notifying the IOP that the subfile has been opened. The IOP then either sets up a small amount of state, or increases a reference count if another CP has already opened the subfile.

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

5.2 Fork Operations

Forks are created using the `gfs_create_fork()` call, which takes as parameters the ID of an open file, the subfile in which the fork is to be created, and a name for the new fork. Galley's run-time library looks up the ID of the appropriate subfile-header block in its cached list, and sends both the header ID and the fork name to that subfile's IOP. By sending the header ID to the IOP, there is no need for an extra indexing operation to take place at the IOP; the IOP is able to retrieve the appropriate subfile-header block immediately. The IOP adds the name of the fork to the subfile-header block, and returns a success or error code to the CP. For the convenience of application programmers, Galley also provides a `gfs_all_create()` call, which creates a fork of the given name in each of the file's subfiles.

As with files, each process in an application that intends to access a fork's data must explicitly open that fork. Forks are opened using the `gfs_open_fork()` call, which takes the same parameters as the fork-creation call. If the fork-open request is successful, Galley returns a *fork ID*, which is used in subsequent calls, much like a file descriptor is used in Unix. Forks are closed with `gfs_close_fork()`,

and deleted with `gfs_delete_fork()`. As with files, if a CP attempts to delete a fork that has a non-zero reference count, that fork is marked for deletion, but is not actually deleted until its reference count reaches 0. For convenience, there are `gfs_all_open`, `gfs_all_close`, and `gfs_all_delete` calls as well.

5.3 Data Access Interface

The standard Unix interface provides only simple primitives for accessing the data in files. These primitives are limited to `read()`ing and `write()`ing consecutive regions of a file. As discussed above, recent studies show that these primitives are not sufficient to meet the needs of many parallel applications [NK96a, NKP+95]. Specifically, parallel scientific applications frequently make many small requests to a file, with *strided* access patterns.

We define two types of strided patterns. A *simple-strided* access pattern is one in which all the requests are the same size, and there is a constant distance between the beginning of one request and the beginning of the next. A group of requests that form a strided access pattern is called a *strided segment*. A *nested-strided* access pattern is similar to a simple-strided pattern, but rather than repeating a single request at regular intervals, the application repeats either a simple-strided or nested-strided segment at regular intervals. Studies show that both simple-strided and nested-strided patterns are common in parallel, scientific applications [NK96a, NKP+95].

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

The higher-level interfaces offered by Galley are summarized below. These interfaces are described in greater detail, and examples are provided, in [NK96a, Nie96]. Note that each request accesses data from a single fork; Galley has no notion of a file-level read or write request.

5.3.1 Simple-strided Requests

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

Beginning at `offset` in the open fork indicated by `fid`, the file system will read `quant` records, of `rec_size` bytes each. The offset of each record is `f_stride` bytes greater than that of the previous record. The records are stored in memory beginning at `buf`, and the offset into the buffer is changed by `m_stride` bytes after each record is transferred. Note that either the file stride (`f_stride`) or the memory stride (`m_stride`) may be negative. The call returns the number of bytes transferred.

When `m_stride` is equal to `rec_size`, data will be *gathered* from disk, and stored contiguously in memory. When `f_stride` is equal to `rec_size`, data will be read from a contiguous region of a file, and *scattered* in memory. It is also possible for both `m_stride` and `f_stride` to be different than `rec_size`, and possibly different than each other.

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

5.3.2 Nested-strided Requests

```
gfs_read_nested(int fid, void *buf, long offset, long rec_size,
                struct stride *vec, int levels)
```

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

5.3.3 Nested-batched Requests

While we found that most of the small requests in the observed workloads were part of either simple-strided or nested-strided patterns, there may well be applications that could benefit from some form of high-level, regular request, but would find the nested-strided interface too restrictive. An example of such an application is given in [Nie96]. For those applications, we provide a *nested-batched* interface. The data structure involved in a nested-batched I/O request is called a *request vector*:

```
struct batch {
    long f_offset;
    long m_offset;
    char f_offset_type;  /* ABSOLUTE or RELATIVE */
    char m_offset_type;  /* ABSOLUTE or RELATIVE */
    char subreq_type;    /* SIMPLE or VECTOR */
    long f_stride;       /* File stride between repetitions */
    long m_stride;       /* Memory stride between repetitions */
    int quant;           /* Number of repetitions */
    int elements;        /* Number of elements in subvec */
    union {
        long size;                /* Simple request */
        struct batch *subvec;     /* Request vector */
    } sub;
};
```

Each request in the vector specifies the offset into the file from which to begin servicing the request. This offset may be absolute or it may be specified relative to the previous request's offset. In addition to simple reads and writes, each request in the vector may be a *strided* request. That is, the application may specify that the request is to be repeated a number of times (`quant`), and may specify the change in both file and memory offsets between each request. Finally, the requests themselves may be vectors of requests, to allow nesting.

This interface gives applications the ability to submit multiple simple or strided requests at once.

5.3.4 List Requests

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

6 Performance

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

6.1 Experimental Platform

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

To determine what effect, if any, our use of TCP/IP would have on the overall performance of our system, we performed some simple benchmarking of the SP-2's TCP/IP performance. According to IBM (verified by own testing), the maximum TCP/IP throughput between two nodes on the SP-2 is approximately 17 MB/s. Unfortunately, as the number of nodes increases, it becomes difficult to maintain this throughput at each node, as shown in Figure 4. In each test, we used 16 *sinks*, and varied the number of *sources* from 4 to 64. Each source sent the same amount of data to each sink, using a fixed record size. For each sink/source configuration, we measured the throughput for a variety of message

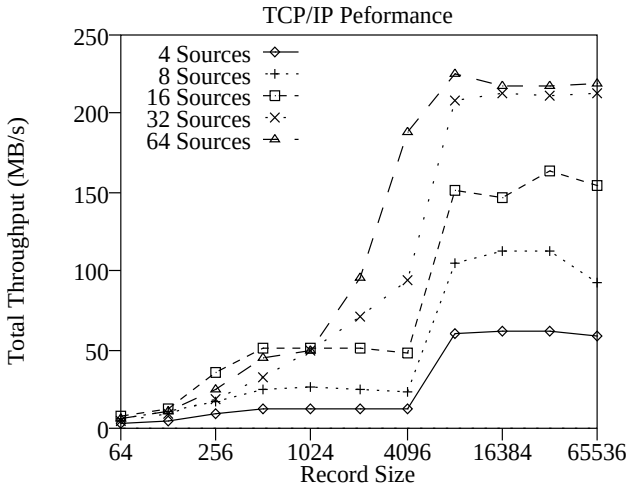

Figure 4: Measured TCP/IP throughout on the SP-2. For each test, there were 16 *sinks* (similar to CPs reading a file), and a variable number of *sources* (similar to IOPs servicing read requests).

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

Each IOP in Galley controls a single disk, logically partitioned into 32KB blocks. For this study, each IOP had a buffer cache of 24 megabytes, large enough to hold 750 blocks. Although each node on the SP-2 has a local disk, access to that disk must be performed through AIX's Journaling File System. While Galley was originally implemented to use these disks, our performance results appeared to be inflated by the prefetching and caching provided by JFS. Specifically, we frequently measured apparent throughputs of over 10 MB/s from a single disk. To avoid these inflated results, we examined Galley's performance using a simulation of an HP 97560 SCSI hard disk, which has an average seek time of 13.5 ms and a maximum sustained throughput of 2.2 MB/s [HP91].

Our implementation of the disk model was based on earlier implementations [RW94, KTR94][2]. Among the factors simulated by our model are head-switch time, track-switch time, SCSI-bus overhead, controller overhead, rotational latency, and the disk cache. To validate our model, we used a trace-driven simulation, using data provided by Hewlett-Packard and used by Ruemmler and Wilkes in their study.[3] Comparing the results of this trace-driven simulation with the measured results from the actual disk, we obtained a demerit figure (see [RW94] for a discussion of this measure) of 5.0%,

[2]The source code for this disk simulator is available online at `http://www.cs.dartmouth.edu/~nils/disk.html`.

[3]Kindly provided to us by John Wilkes and HP. Contact John Wilkes at wilkes@hplabs.hp.com for information about obtaining the traces.

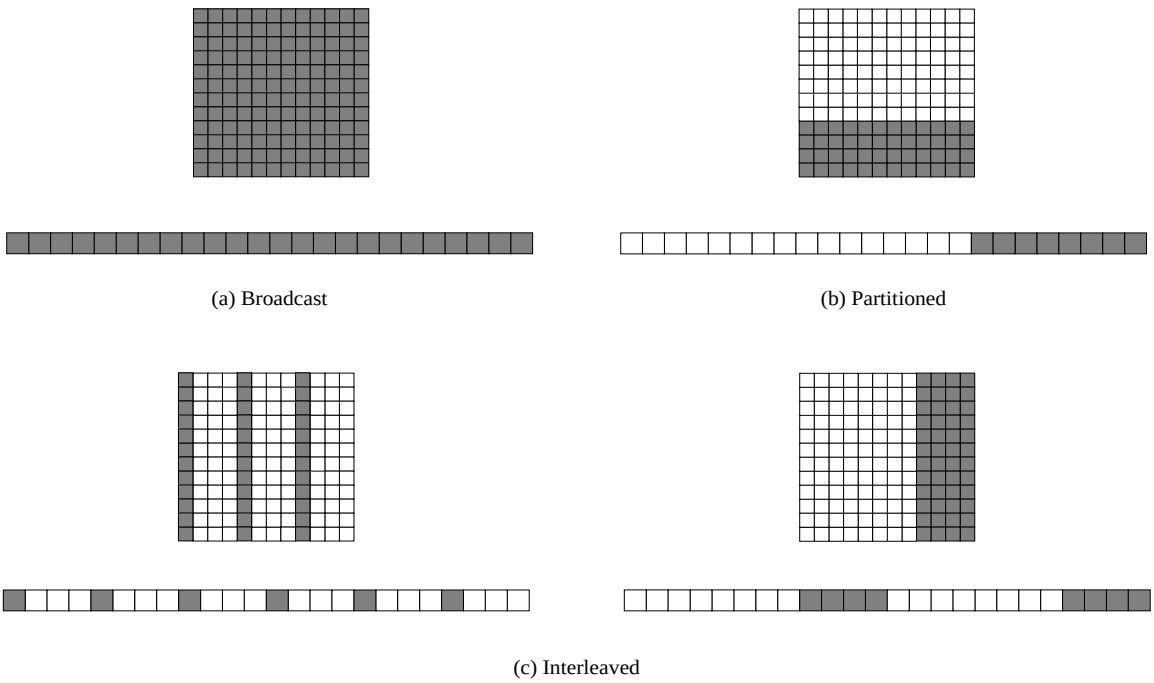

Figure 5: The three access patterns examined in this study. Two views of each pattern are displayed: the pattern as applied to a linear file, and matrix distributions that could give rise to the pattern. For these examples, we assume that the matrices are stored in row-major order. Each square corresponds to a single *record* in the file, and the highlighted squares represent the records accessed by a single compute node in a group of four.

indicating that our model was extremely accurate.

The simulated disk is integrated into Galley by creating a new thread on each IOP to execute the simulation. When the thread receives a disk request, it calculates the time required to complete the request, and then suspends itself for that length of time. While, in most cases, the disk thread does not actually load or store the requested data, metadata blocks must be preserved. To avoid losing that data, the disk thread maintains a small pool of buffers, which is used to store 'important' data. When the disk simulation thread copies data to or from a buffer, the amount of time required to complete the copy (which we calculate at system startup) is deducted from the amount of time the thread is suspended. It should be noted that the remainder of the Galley code is unaware that it is accessing a simulated disk.

6.2 Access Patterns

We examine the performance of Galley under several different access patterns, shown in Figure 5, each of which is composed of a series of requests for fixed-size pieces of data, or *records*. Although these patterns do not directly correspond to a particular 'real world' application, they are representative of the general patterns we observed to be most common in production multiprocessor systems, as described above. Our measurements were performed using a file that contained a subfile on each IOP, and a single fork within

each subfile. To allow us to better understand the system's performance, by removing one variable, the forks were laid out contiguously on disk. The patterns shown in Figure 5 reflect the patterns that we access *from each IOP*. The correspondence between the IOP-level access patterns used in this study, and the file-level patterns observed in actual applications, is discussed for each pattern below.

The simplest access pattern is called *broadcast*. With this access pattern every compute node reads the whole file. In other words, the IOPs *broadcast* the whole file to all the CPs. This access pattern models the series of requests we would expect to see when all the nodes in an application read a shared file, such as the initial state for a simulation. Since, in order to read all the data in a file, an application must read all the data in every subfile, a broadcast pattern at the file level clearly corresponds to a broadcast pattern at each subfile. Although it may seem counterintuitive for an application to access large, contiguous regions of a file in small chunks, we observed such behavior in practice. One likely reason that data would be accessed in this fashion is that records stored contiguously on disk are to be stored non-contiguously in memory. In the simplest case, this pattern would be similar to the interleaved pattern described below, with the interleaving occurring in memory rather than on disk. Since it seems unlikely that an application would want every node to rewrite the entire file, we did not measure the performance of the broadcast-write case.

Under a *partitioned* pattern, each compute node accesses a distinct, contiguous region of each file. This pattern could represent either a one-dimensional partitioning of data or the series of accesses we would expect to see if a two-dimensional matrix were stored on disk in row-major order, and the application distributed the rows of the matrix across the compute nodes in a BLOCK fashion (using HPF terminology [HPF93]). There are two different ways a partitioned access pattern at the file level can map onto access patterns at the IOP level. The first occurs if the file is distributed across the disks in a BLOCK fashion; that is the first $1/n$ of the file bytes in the file are mapped onto the first of the n IOPs, and so forth. For each IOP, this mapping results in an access pattern similar to a broadcast pattern with 1 compute processor. The other mapping distributes blocks of data across the disks in a CYCLIC fashion. This second mapping is more interesting and corresponds to the mapping used by most implementations of a linear file model. This distribution results in accesses by each CP to each IOP. In a system with 4 CPs, the first CP would access the first 1/4 of the data in each subfile, and so forth. Thus, using the second mapping, a partitioned pattern at the file level leads to a partitioned pattern at each IOP. As with the broadcast pattern, applications may access data in this pattern using a small record size if the the data is to be stored non-contiguously in memory.

In an *interleaved* pattern, each compute node requests a series of noncontiguous, but regularly spaced, records from a file. For the results presented here, the interleaving was based on the record size. That is, if 16 compute nodes were reading a file with a record size of 512 bytes, each node would read 512 bytes and then skip ahead 8192 (16*512) bytes before reading the next chunk of data. This pattern models the accesses generated by an application that distributes the columns of a two-dimensional matrix across

the processors in an application, in a CYCLIC fashion. To see how this file-level pattern maps onto an IOP-level pattern, assume the linear file is distributed traditionally, with blocks distributed across the subfiles in a CYCLIC fashion. In the simplest case, the block size might be evenly divisible by the product of the record size and the number of CPs. In this case, every block in the file is accessed with the same interleaved pattern, and any rearrangement of the blocks (between or within disks) will result in the same subfile-access pattern. Thus, the blocks can be declustered across the subfiles, but the access pattern within each subfile will still be interleaved. There are, of course, more complex mappings of an interleaved file-level pattern to an IOP-level pattern, but we focus on the simplest case.

For this performance analysis, we held the number of compute processors constant at 16, and varied the number of IOPs (each with one disk) from 4 to 64. Thus, the CP:IOP ratio varied from 1:4 to 4:1. Each test began with an empty buffer cache on each IOP, and each write test included the time required for all the data to actually be written to disk. While the size of each fork was fixed, the amount of data accessed for each test was not. Since the system's performance on the fastest tests was several orders of magnitude faster than on the slowest tests, there was no fixed amount of data that would provide useful results across all tests. Thus, the amount of data accessed for each test varied from 4 megabytes (writing 64-byte records to 4 IOPs) to 2 gigabytes (reading 64-KB records from 64 IOPs). The results presented here represent the average of three executions of each test.

6.3 Traditional Interface

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

Figure 6 shows the total throughput achieved when reading a file with various record sizes for each access pattern. Figure 7 presents similar results for write performance. The performance curves have the same general shape as throughput curves in most systems; that is, as the record size increased, so did the performance. As in most systems, eventually a plateau was reached, and further increases in the record size did not result in further performance increases. The precise location of this plateau varied

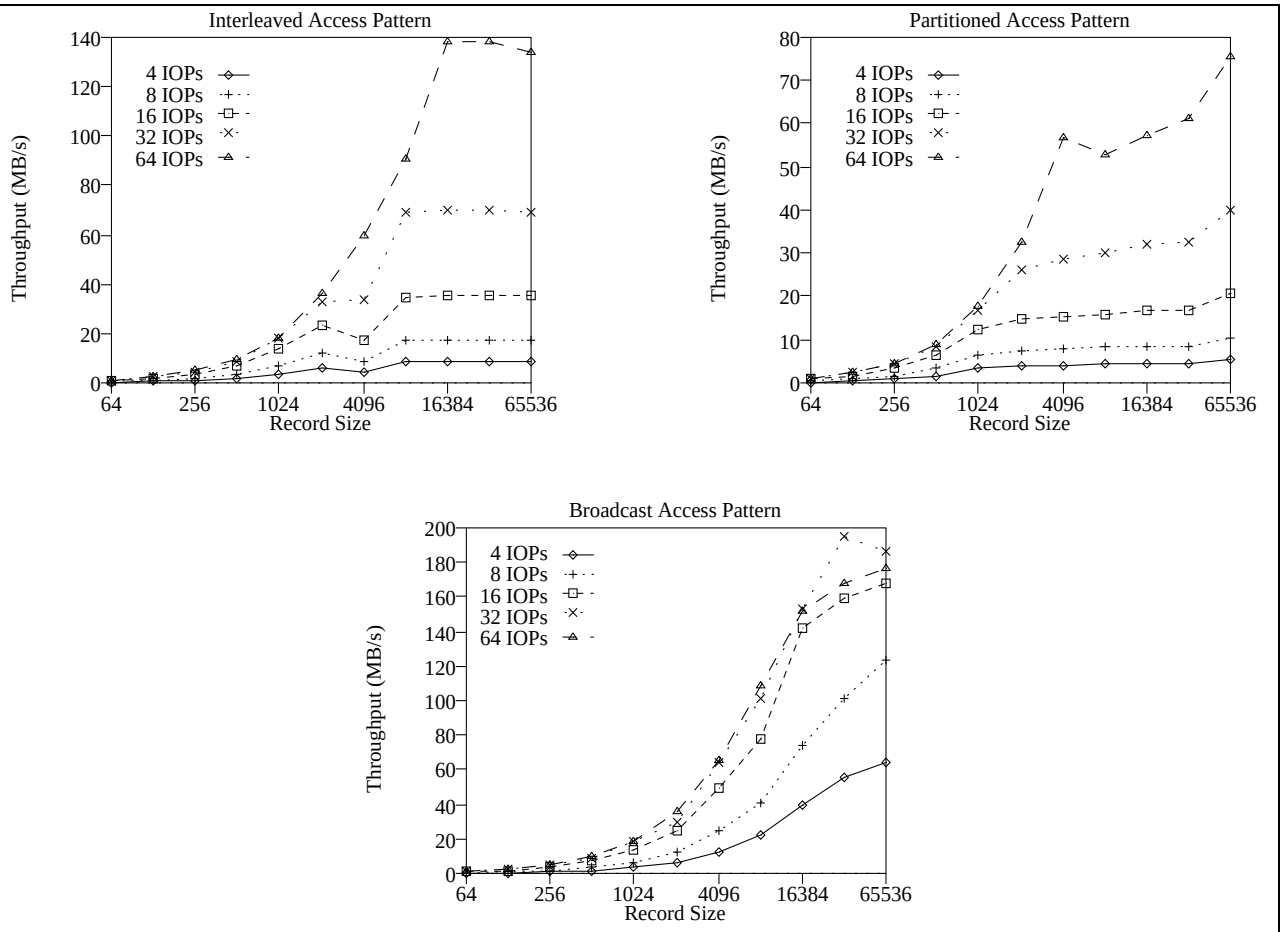

Figure 6: Throughput for read requests using the traditional Unix-like interface. There were 16 CPs in every case. Note the different scales on the y-axis.

between patterns and CP:IOP ratios. Unsurprisingly, when accessing data in small pieces, the total throughput was limited by a combination of software overhead and by the high latency of transferring data across a network, regardless of the access pattern.

The choice of access pattern had the greatest effect on performance when reading data with large blocks. When reading an interleaved pattern, the system's peak performance was limited by the sustainable throughput of the disks on each IOP (about 2.2 MB/s). There was a small dip in performance as the record size increased from 2 KB to 4 KB, with small numbers of IOPs. With records of 2 KB or smaller, every CP reads data from every block. With a record size of 4 KB, each CP reads data only from alternate blocks. As a result, it is possible for a request for block $n + 1$ to arrive before a request for block n, slightly degrading disk performance. These out-of-order requests are less likely to occur with larger records. The overall performance when reading the partitioned pattern was limited by the time the disk spent seeking from one region of the file to another. The small spike in performance with 64 IOPs and a 4 KB record size is repeatable, but it is not clear what causes it.

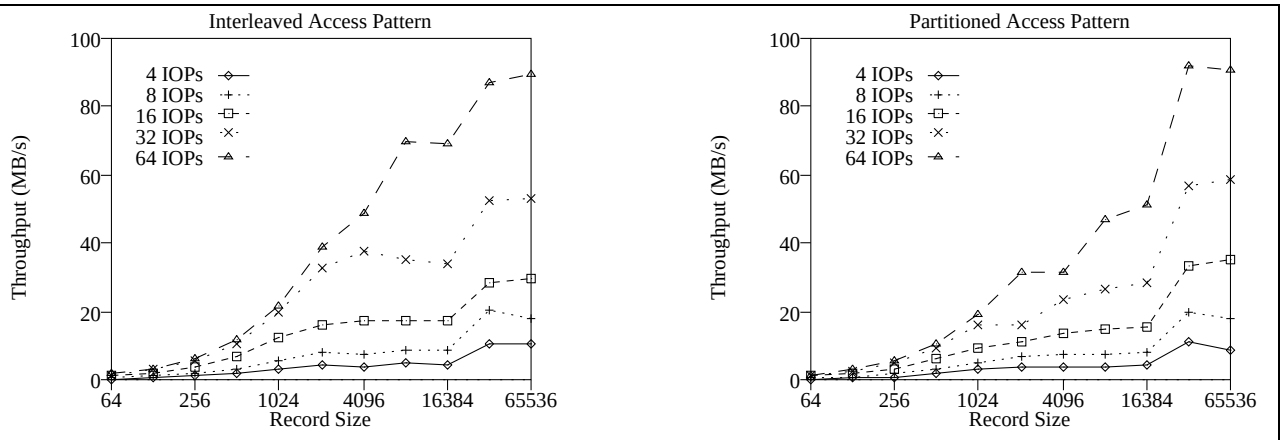

Figure 7: Throughput for write requests using the traditional Unix-like interface. There were 16 CPs in every case.

When testing an earlier version of Galley we found that with large numbers of IOPs, the network congestion at the CPs was so great that the CPs were unable to receive data and issue new requests to the IOPs in a timely fashion [NK96b]. As a result, the DiskManagers on the IOPs were unable to make intelligent disk scheduling decisions, causing excess disk-head seeks and thrashing of the on-disk cache. The combination of the network congestion and the poor disk scheduling led to dramatically reduced performance with large record sizes in the interleaved and partitioned patterns. To avoid this problem, we added a simple flow-control protocol to Galley's data-transfer mechanism.

Under the broadcast access pattern, data was read from the disk once, when the first compute processor requested it, and stored in the IOP's cache. When subsequent CPs requested the same data, it was retrieved from the cache rather than from the disk. Since each piece of data was used many times, the cost of accessing the disk was amortized over a number of requests, and the limiting factors were software and network overhead. In this case, the total throughput of the system was limited by the SP-2's TCP/IP performance, as discussed above.

Consider Figure 7. When writing data with records of less than 32 KB, the file system had to read each block off the disk before the new data could be copied into it. Without this requirement, any data that was stored in that block would be lost – even data that was not being modified by the write request. As a result the system's performance was slower when writing small records than when reading them. Furthermore, with small records, the interleaved pattern had higher total throughput than the partitioned pattern. As when reading data, the interleaved pattern had higher throughput because the partitioned pattern forced the disk to spend time seeking between one region of the file and another.

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

Furthermore, since much of the data was not actually written to disk until the CPs called `gfs_sync()`, at the end of the test, the system could avoid many of the excess seeks in the partitioned case.

6.4 Strided Interface

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

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

Figure 8 shows the total throughput achieved when reading a file with various record sizes for each access pattern using the new interface, and Figure 9 shows corresponding results for writing.

Given the traditional interface, the disk scheduler had to handle each request in the order they arrived from the CPs. This requirement led to excess disk-head movement primarily in the partitioned pattern, but also in the interleaved pattern when the record size was larger than 2 KB (32 KB/16 CPs). Since each CP read from the same data blocks in the broadcast case, and in an interleaved pattern with small records, the disk schedule was optimal even with the traditional interface. Since many of the disk accesses in the traditional write cases occurred after a call to `gfs_sync()`, the disk scheduler was able to make intelligent decisions then as well. Therefore, the tests on which the new interface lead to the greatest improvements in the disk schedule were the interleaved and partitioned read tests, and these were the two tests where the peak throughput to the CPs improved most dramatically.

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

While the increase in peak performance is interesting, the most striking difference between the two sets of tests is that, in most cases, Galley was able to achieve peak performance with records as small as 64 bytes—two or three orders of magnitude smaller than the request sizes required to achieve peak throughput using the traditional interface. Other than increased opportunities for intelligent disk scheduling, the primary performance benefit of our interface was a reduction in the number of messages, accomplished by packing small chunks of data into larger packets before transmitting them to the

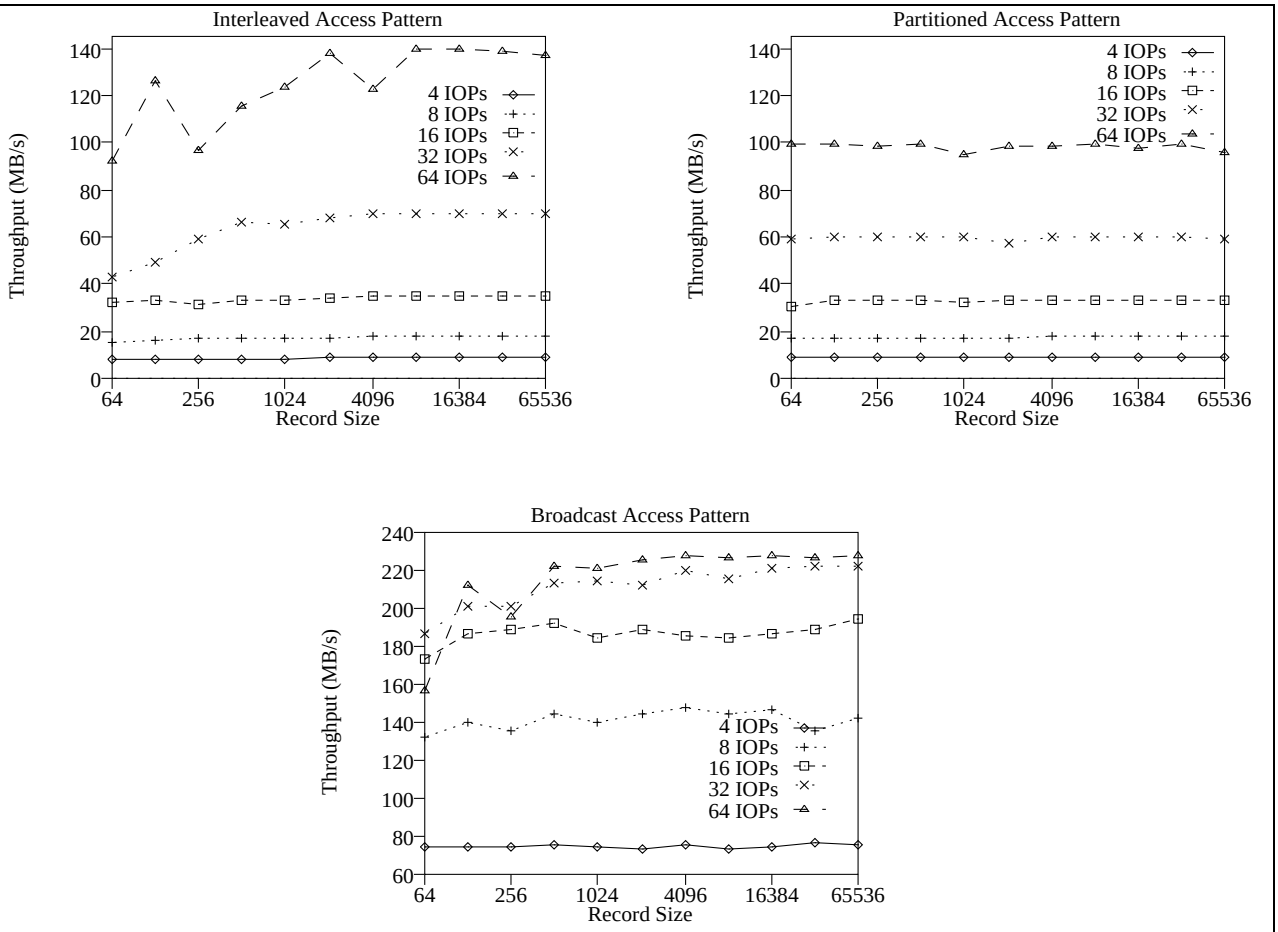

Figure 8: Throughput for read requests using the strided interface. There were 16 CPs in every case. Note the different scales on the y-axis.

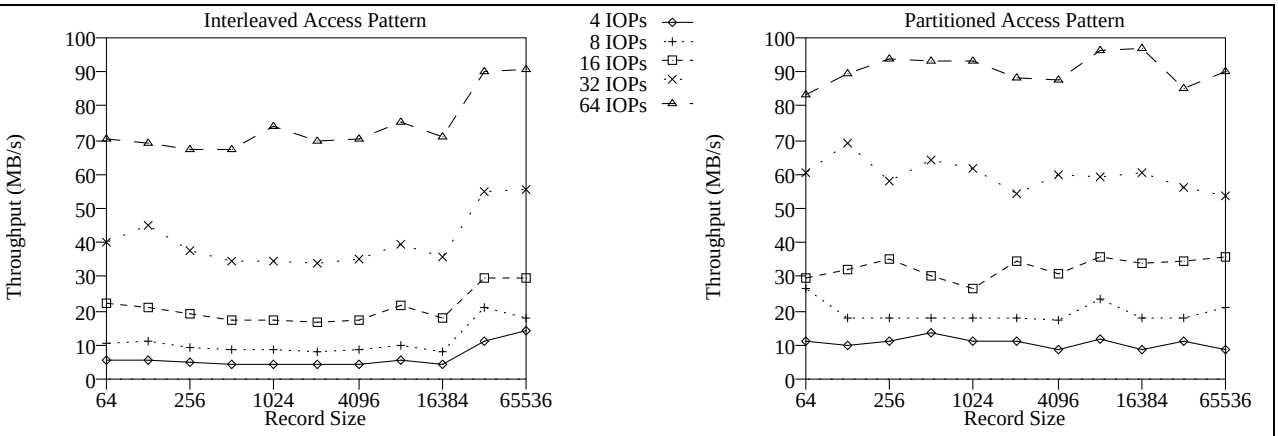

Figure 9: Throughput for write requests using the strided interface. There were 16 CPs in every case.

receiving node.

One short-term goal for Galley is reducing the variability of the system's performance. This variability is most obvious in the interleaved-read pattern with 64 IOPs. Performance was also variable when writing data, particularly in the partitioned pattern. We are investigating two possible solutions to this problem: refining our flow-control strategy and modifying the DiskManager to consider the impact on network performance when designing a disk schedule.

While it is clear that the strided interface allowed the file system to deliver much better performance, the throughput plots shown in Figures 8 and 9 present only part of the picture. Figure 10 shows the speedup of the strided-read interface over a traditional read interface, and Figure 11 shows similar results for the write interfaces. When using an interleaved pattern with small records, the strided interface led to speedups of up to 78 times when reading, and 45 times when writing. The increase in performance for small records in a partitioned pattern was even greater: up to 83 times when reading and 54 times when writing. The broadcast-read pattern had the largest speedups for small records, ranging from 150 to 325. Although there was less room for improvement with large records, better disk scheduling when reading interleaved and partitioned patterns led to higher performance even for large records.

7 Related Work

A variety of multiprocessor file systems have been developed over the past ten years or so. While many of these were similar to the traditional Unix-style file system, there have been also several more ambitious attempts.

Intel's Concurrent File System (CFS) [Pie89, Nit92], and its successor, PFS, are examples of multiprocessor file systems that use a linear file model and provide applications with a Unix-like interface. Both systems provide limited support to parallel applications in the form of *file pointers* that may be shared by all the processes in the application. CFS and PFS provide several *modes*, each of which provides the applications with a different set of semantics governing how the file pointers are shared. Other multiprocessor file systems with this style of interface are SUNMOS and its successor, PUMA [WMR+94], *sfs* [LIN+93], and CMMD [BGST93].

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

Ironically, the multiprocessor file system most removed from the traditional Unix-like model also provides the most Unix-like interface. PIOFS, the file system for IBM's SP-2, allows users and applications to interact with it exactly as they would interact with any AIX file system. Administrators and advanced

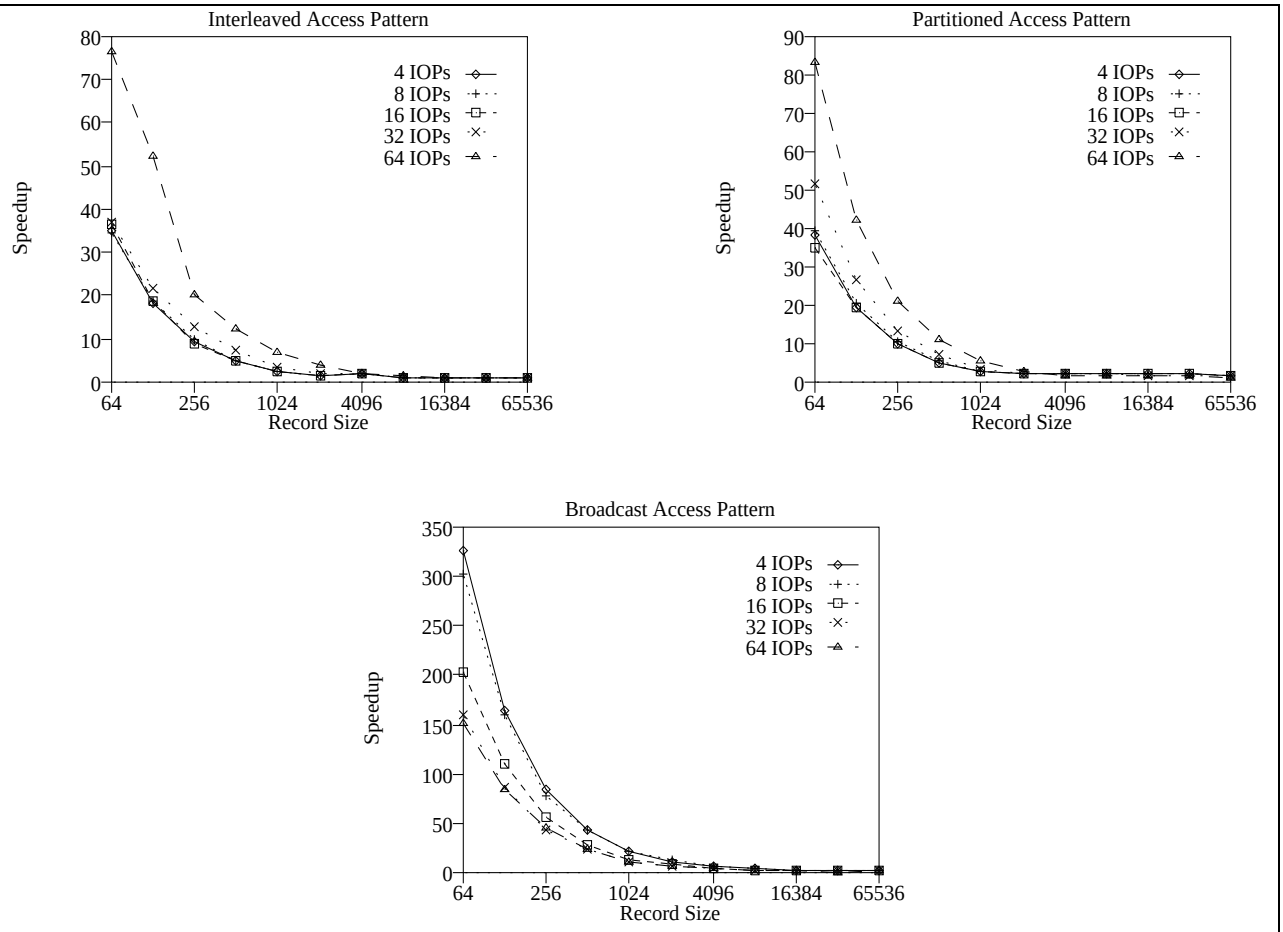

Figure 10: Increase in throughput for read requests using the strided interface. Note the different scales on the y-axis.

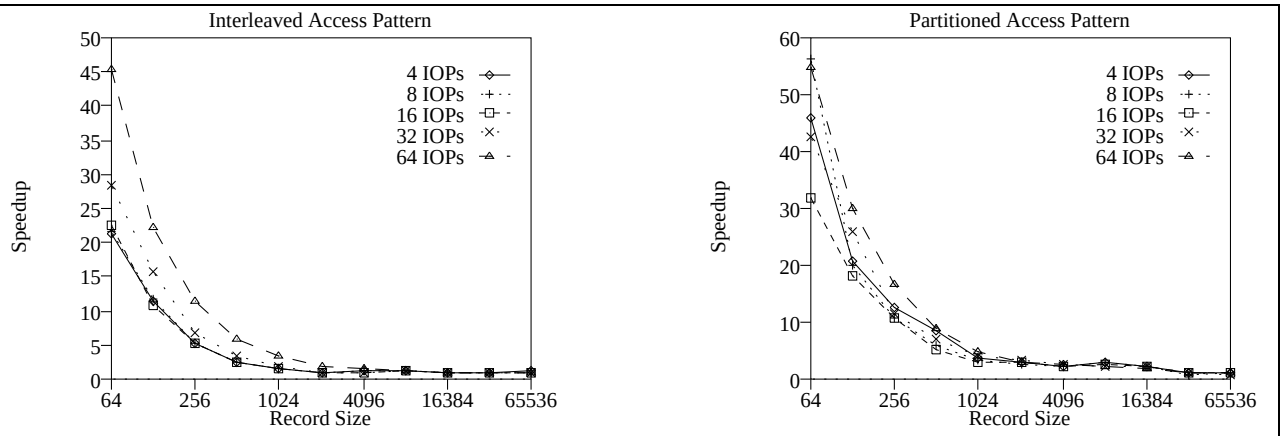

Figure 11: Increase in throughput for write requests using the strided interface.

users may also choose to interact with PIOFS's underlying parallel file system, which is based on the Vesta file system [CF94, CFP+95]. Files in Vesta are two-dimensional, and are composed of multiple *cells*, each of which is a sequence of *basic striping units*. BSUs are essentially records, or fixed-sized sequences of bytes. Like Galley's subfiles, each cell resides on a single disk. While Galley only allows a file to have a single subfile per disk, in Vesta a single disk may contain many cells. Equivalent functionality could be achieved on Galley by mapping cells to forks rather than subfiles. Vesta's interface includes *logical views* of the data. These views are essentially rectangular partitionings of the two-dimensional file, and can provide the application with much of the functionality of Galley's strided interfaces. Vesta provides users with a different and powerful way of thinking about data storage. Its largest drawback is that it is ill-suited to datasets that cannot be partitioned into rectangular, non-overlapping sub-blocks of a single size. Like Galley, Vesta uses a hashing scheme to distribute metadata. In addition to the functionality of Vesta, PIOFS provides applications with a Unix-like interface. We have built a library that provides a Vesta-like interface for Galley.

8 Summary and Future Work

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

The results of our experiments indicate that our new style of interface increased performance by several orders of magnitude. More importantly, this new interface allows high performance on access patterns that are known to be common in scientific applications, and which are known perform poorly on most current multiprocessor file systems.

Future Work. We are exploring several areas for further work. First, Galley currently supports only a single disk per IOP. Since our maximum throughput is frequently limited by the disk's maximum throughput, adding support for multiple disks at the IOP is a high priority. Second, we have only examined the performance of the system running microbenchmarks. To really understand Galley's performance, we plan to study how real applications perform on Galley. Finally, we intend to examine how Galley performs when asked to service requests from multiple applications to multiple files at once.

References

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

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

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

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

[CF94] Peter F. Corbett and Dror G. Feitelson. Design and implementation of the Vesta parallel file system. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 63–70, 1994.

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

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

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

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

[HPF93] High Performance Fortran Forum. *High Performance Fortran Language Specification*, 1.0 edition, May 3 1993. http://www.erc.msstate.edu/hpff/report.html.

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

[KN94] David Kotz and Nils Nieuwejaar. Dynamic file-access characteristics of a production parallel scientific workload. In *Proceedings of Supercomputing '94*, pages 640–649, November 1994.

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

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

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

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

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

[Nie96] Nils Nieuwejaar. A linear file model and an implementation of the NHT-1 I/O application benchmark on the Galley Parallel File System. Technical Report In progress, Dept. of Computer Science, Dartmouth College, June 1996.

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

[NK96a] Nils Nieuwejaar and David Kotz. Low-level interfaces for high-level parallel I/O. In Ravi Jain, John Werth, and James C. Browne, editors, *Input/Output in Parallel and Distributed Computer Systems*, chapter 9, pages 205–223. Kluwer Academic Publishers, 1996.

[NK96b] Nils Nieuwejaar and David Kotz. Performance of the Galley parallel file system. In *Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 83–94, May 1996.

[NKP+95] Nils Nieuwejaar, David Kotz, Apratim Purakayastha, Carla Schlatter Ellis, and Michael Best. File-access characteristics of parallel scientific workloads. Technical Report PCS-TR95-263, Dept. of Computer Science, Dartmouth College, August 1995. To appear in IEEE TPDS.

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

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

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

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

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

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

[SW95] K. E. Seamons and M. Winslett. A data management approach for handling large compressed arrays in high performance computing. In *Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 119–128, February 1995.

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

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