NORTHWESTERN UNIVERSITY

Optimizations for High-Performance I/O Libraries

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Electrical and Computer Engineering

By

Kenin Coloma

EVANSTON, ILLINOIS

December 2007

ABSTRACT

Optimizations for High-Performance I/O Libraries

Kenin Coloma

I/O performance has perpetually lagged well behind that of processors. This gap is made even more pronounced in the field of high-performance computing. While simply adding more network bandwidth and increasing the number of storage units is a simple way to increase theoretical peak I/O bandwidth, the software libraries for harnessing parallel storage fall significantly short of delivering this ideal. The typical software stack from low-level communication libraries through several middleware layers all the way up to the application all need to work in a concerted effort in order to efficiently access files. Each layer offers unique opportunities for I/O optimization, but the MPI layer is particularly interesting for its portability and its preservation of higher-level semantic relationships.

MPI is a critical component of almost every parallel computing system. Without MPI the computational power of these systems would be far less accessible to users. As the number of processors in both clusters and more specialized parallel systems continually grows, I/O is often a bottleneck for many applications ranging from climate modeling to computational physics. MPI-IO provides a portable means of accessing storage while also allowing for system specific optimizations. The MPI-2 collective I/O routines were included

to preserve the semantic relationships between the I/O operations of several processes. The ROMIO implementation of MPI-IO is conveniently open source and frequently encountered, as it is distributed with Argonne National Laboratory's MPICH and several other MPI implementations. Working with ROMIO allows for a potentially broad impact on many high-performance computing installations.

Caching is of particular interest at the MPI-IO level. I/O caching has always held promise as a performance booster in high-performance I/O, but the overhead and involved in ensuring coherency have mitigated these performance benefits. Though traditionally done at the file system level, there are significant benefits to manipulating I/O within an MPI-IO library to leverage an I/O cache within either the file system or MPI-IO. All these changes and experiments can be made with minimal impact on existing MPI applications.

The Persistent File Realm (PFR) technique works only when I/O calls are confined to collective I/O, but is able to efficiently rearrange I/O access patterns to ensure coherence in a file system cache that otherwise may not be coherent. The Direct Access Cache (DACche) system remedies the collective I/O requirement by implementing the client-side cache itself at the user-level. Because DACche is started and controlled by the application, it can be tuned to the particular characteristics of the application. Existing techniques like two phase I/O work well with DACche, and DACche allows for a mix of both independent and collective I/O calls.

Acknowledgments

I would like to take this space to thank my advisor, Alok Choudhary for all the time, effort, and guidance he's afforded me in the past 6 years. Additionally, I thank my committee members, Rajeev Thakur, Gokhan Memik, and Wei-keng Liao. I'm also indebted to: my collaborators at Sandia National Laboratories, Lee Ward, Ron Oldfield, Ruth Klundt, Ron Brightwell, and Neil Pundit; my collaborators at Argonne National Laboratory, Rob Ross, and Rob Latham; my collaborators at Los Alamos National Laboratory, Gary Grider, James Nuñez, and John Bent; and my lab mates Avery Ching, Jianwei Li, Ying Liu, Joseph Zambreno, Steve Chiu, and Jayaprakash Pisharath for their innumerable contributions without which this research would have been all but impossible. I'm also gratefull to all my friends for their indispensable encouragement: Daniel Honbo, Adrienne "Zelda" Ancheta, Rae Inafuku, Randall Wong, Simona Petrutiu, Grace Nijm, Meng-ting Shieh, and Sandra Wu. Lastly, I must thank my family for their unceasing love and support, especially my parents Geneson Coloma and Theresa Kiehm.

This work was supported in part DOE's SCiDAC program (Scientific Data Management Center), award number DE-FC02-07ER25808, DOE's SCIDAC-2 (Scientific Data Management Center for Enabling Technologies (CET)) award number DE-FC02-07ER25808/A000, NSF/DARPA ST-HEC program under grant CCF-0444405, NSF HECURA CCF-0621443 and NSF SDCI HPC program under grant OCI-0724599.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

I/O performance is well-known to lag well behind that of processors. This gap is made even more obvious in the field of high-performance computing. While simply adding more network bandwidth and increasing the number of storage units is a simple way to increase theoretical peak I/O bandwidth, the software libraries for harnessing parallel storage fall significantly short of this vendor-favored metric. Each layer in the typical software stack in Figure 1.1 from low-level communication libraries, through several middleware layers all the way up to the application, all need to work in a concerted effort in order to efficiently access files. Each layer offers unique opportunities for I/O optimization, but the MPI layer is particularly interesting for its portability and its preservation of high-level semantic relationships.

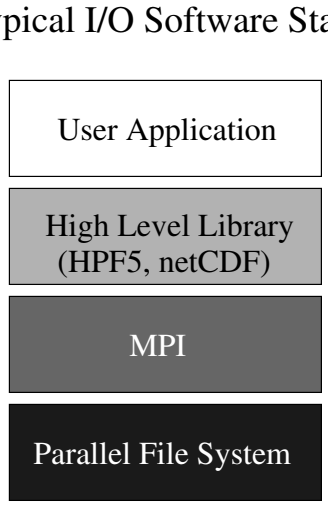

Figure 1.1. Each layer in the I/O software stack must work together.

MPI has become a critical component of most parallel computing systems. Without MPI, the power of these systems would be far less accessible from the programming perspective. As the number of processors in clusters and more specialized parallel systems continually grows, I/O is often a bottleneck for many applications ranging from climate modeling to computational physics. MPI-IO provides a portable means of accessing storage while simultaneously allowing for system-specific optimizations. The MPI-2 collective I/O routines were included to preserve the semantic relationships between the I/O operations of several processes. The ROMIO implementation of MPI-IO is conveniently open source and very common, as it is distributed with Argonne National Laboratory's MPICH and several other MPI implementations. Working with ROMIO allows for a potentially broad impact on many high-performance computing installations.

By targeting a specific set of applications with particular I/O access patterns, certain assumptions and simplifications can be made to enable various optimizations. Scientific applications are characterized by cyclic bursts of regularly spaced small non-contiguous I/O operations[**22, 32**], and while many files may be open at a time, applications rarely access the same files at the same time. We have developed several optimizations for improving performance in this context. If restricted to only collective I/O, then an opportunity for explicit coordination and communication is guaranteed.

Persistent File Realms (PFRs) address cache coherence issues for a file system's non-coherent client-side file cache by carefully managing the I/O calls to the file system.

In an effort to overcome the "collective I/O only" requirement of PFRs, the Direct Access Cache (DACche) system was conceived as a user-level cooperative cache that can handle both collective I/O calls and independent I/O calls. DACche functions as a page based distributed peer cache. Every process manages its own cache for evictions and misses; however these

caches are remotely accessible to any other process through Remote Memory Access (RMA). A logical page in the file can only reside in the cache of no more than one process. This restriction ensures a single globally coherent view of file data at any given time. DACche can, though not enabled by default, enforce sequential consistency as well. Location and other metadata for each logical file page are distributed across compute processes, and are also remotely accessible. The most challenging aspect of the design and implementation of DACche is the lack of coordination between processes, mandating a passive design. Though DACche is similar to peer-to-peer systems, its one-sided design is an added wrinkle. DACche is obviously not based on a client-server model. A more appropriate name for this design is *self-serve*, where each process helps itself to data residing on other processes while observing many rules for fairness and tacit coordination. In this self-serve model, two-sided communication is limited to only certain instances following strict protocols. The basic architecture of DACche is illustrated in Figure 1.2. The primary purpose of the distributed metadata is to maintain location information on each DACche page.

I/O to a page in DACche basically involves the following general steps

(1) lock page metadata
(2) retrieve metadata
(3) modify metadata
(4) write metadata back
(5) unlock page metadata
(6) access page either remotely or locally

Assuming there is no lock contention for the metadata, there are at most five points of communication. All of these communications are one-sided, making them low-latency. Except

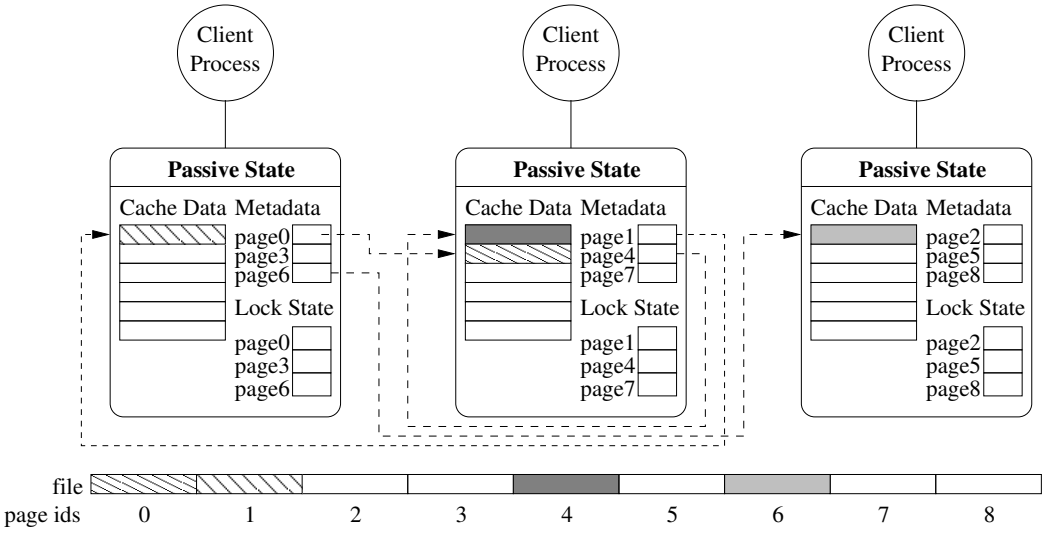

Figure 1.2. DACHe architecture with passive metadata and cache servers on each client. Metadata is striped across clients, but a file page can be cached on any client.

for the last message to actually transfer remote cache data, the control data being passed around is relatively small. The size of the DACHe pages, and consequently the number of accesses, will have a large effect on application performance.

One of the requirements for getting DACHe into a deployable state is building a passive lock system to ensure mutually exclusive access to the DACHe metadata storage. Several passive locking algorithms are considered: polling, distributed queues, and MPI-2 based locking. MPI-2 based locking has the distinct advantage of being portable on any system with an MPI-2 library. The other two locking methods are implemented using lower-level API's. This sacrifices portability, but presumably offers some performance gain because of its proximity to the hardware.

CHAPTER 2

Background

Since several techniques, themes, technologies, and issues recur throughout the text, they are worth describing all at once in the following few sections.

2.1. MPI-IO

MPI-IO is a subset of the MPI-2[**16**] specification. By using MPI's derived datatypes, data can be transferred between complex file and memory layouts in single MPI-IO functions. A *memory datatype* describing how data should be accessed in memory is passed directly to

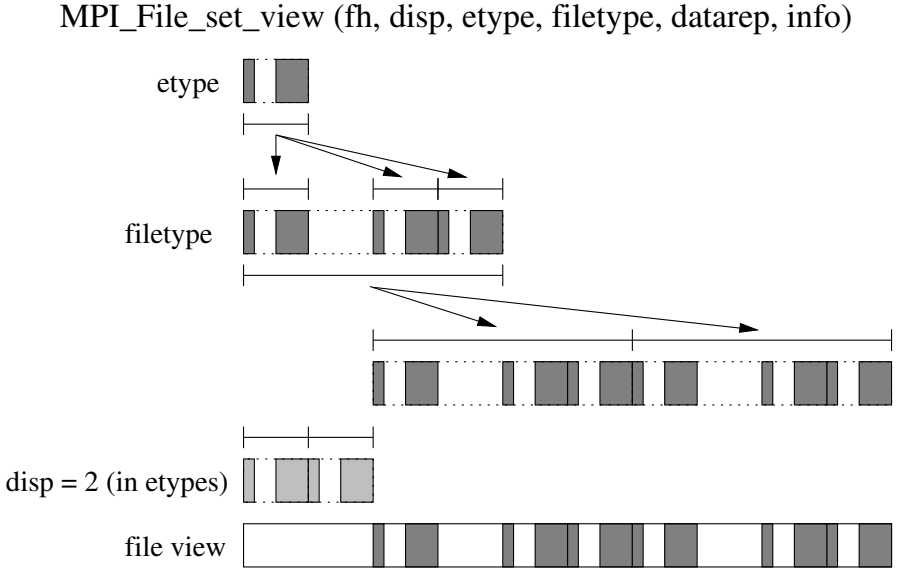

Figure 2.1. File views illustrated: Filetypes are built from *etypes* which themselves may be derived datatypes. The filetype access pattern is implicitly iterated forward starting from *disp*. An actual count for the filetype is not required as it conceptually repeats forever, and the amount of I/O done is dependent on the buffer memory type and count.

the MPI-IO read and write functions. The *file view* for a particular process is described in a separate collective call with a *file datatype* and displacement. The file view sets the accessible regions of a file by using the file datatype as a template and repeating it starting from a byte displacement. Figure 2.1 illustrates the construction of a file view and its components. The primary benefit of such a descriptive I/O interface is the preservation of an application's intent. Rather than performing individual contiguous transfers between memory and file, the MPI library can capture a number of related I/O requests in an application and make optimizations to improve overall I/O performance. One such notable optimization is data sieving[**46**], detailed in Section 2.2.

The collective I/O interfaces for MPI-IO retain the use of derived datatypes both for memory and file description, but they are also designed to allow cooperative I/O optimizations across processes by preserving higher-level access patterns. Collective I/O leverages the fact that processes may be accessing storage in some meaningful way as a group. As the collective I/O functions are virtually identical to the *independent I/O* functions, they do not preclude the use of any independent I/O optimizations.

2.1.1. I/O Semantics

Consistency semantics greatly affect the complexity of any I/O system. Under POSIX specifications[**19**] for file systems, newly written data should always be sequentially consistent and visible to all processes after a write routine has returned. Reads and writes are also atomic, so a read should never return partially written data: a write has either completed or not started. Such strict semantics are extremely hard to enforce in a distributed environment. MPI semantics are slightly more relaxed in that sequential consistency is guaranteed only under certain conditions and data needs to be immediately visible to only the

processes in the same communicator at the completion of a write. MPI guarantees sequential consistency if any of the following conditions exist:

- Atomic mode is explicitly set by the user
- All I/O operations are non-concurrent
- All I/O operations are non-conflicting

At first glance, POSIX consistency semantics seem more stringent than MPI semantics, and that a file system adhering to the POSIX semantics would not need additional support in the MPI-IO implementation to support MPI semantics. Since MPI-IO allows access to multiple noncontiguous regions of a file in one I/O call, MPI semantics require a step beyond what POSIX alone provides. MPI-IO enables atomic operations across noncontiguous file regions. This issue is discussed further by Liao *et al.*[**24**].

2.1.2. ROMIO

ROMIO is an implementation of the I/O functions of the MPI-2 standard, and is developed at Argonne National Laboratory. The ROMIO[**39**] implementation uses both data sieving and the two phase I/O method (a collective I/O optimization)[**47**]. ROMIO will run on any file system that at least implements the POSIX API or has its own Abstract Device (ADIO)[**45**] implementation. ROMIO presently supports a mix of commonly used local, distributed, and parallel file systems including SGI's XFS, IBM's PIOFS, Panasas' PanFS[**35**], NFS, and the Parallel Virtual File System (PVFS 1[**6**] & 2[**48**]). Because an ADIO layer can be implemented for each individual file system, it can take advantage of any advanced features a specific file system may provide. ROMIO leverages the PVFS noncontiguous I/O interfaces available including list I/O[**7**] and datatype I/O [**9**].

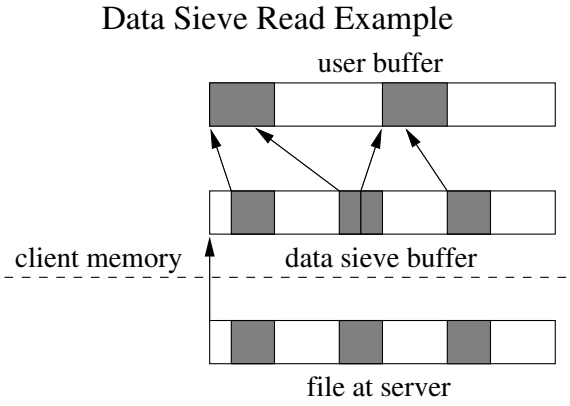

Figure 2.2. In a data sieving read, a section of file that contains the desired data is first read into a temporary buffer before being copied to the user's memory.

2.2. Data Sieving I/O

The data sieving optimization [**4, 44**], operates on large contiguous sections of a file to fulfill several requests at a time. The reduced the number of I/O requests sent to the file system results in less accumulated request overhead. At the server, the reduced number of head seeks and larger request sizes are mechanically much easier for the disk. A data sieving read will read a large contiguous section of the file to satisfy a number of noncontiguous requests and discard the unused data as in Figure 2.2. If the write request is noncontiguous, a data sieving write must first read the contiguous section of the file to modify and then write back the entire region. The underlying file system is required to support locking to ensure correctness in a data sieving write as an independent I/O operation.

2.3. Collective I/O

The collective I/O interface in MPI-2 explicitly groups the I/O calls across multiple processes into one collective unit. Since the I/O of each process is very likely to be related optimizations can be made to the aggregate I/O request to improve overall I/O performance.

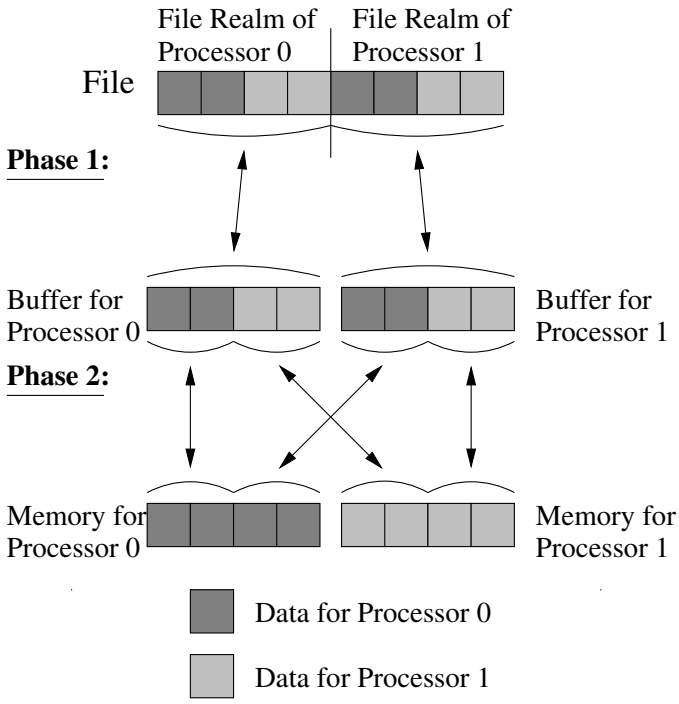

Figure 2.3. In the read case, data flows downward to processor memory. In the write case, data flows up to the file.

The base collective I/O routines are:

```
MPI_File_set_view (fh, disp, etype,
    filetype, datarep, info)
MPI_File_read_all (fh, buf, count,
    datatype, status)
MPI_File_write_all (fh, buf, count,
    datatype, status)
```

2.3.1. Two phase I/O

Two phase I/O[**43, 13**] consists of an I/O phase and a data exchange phase, but there is some additional incidental implementation specific communication. Figure 2.3 illustrates

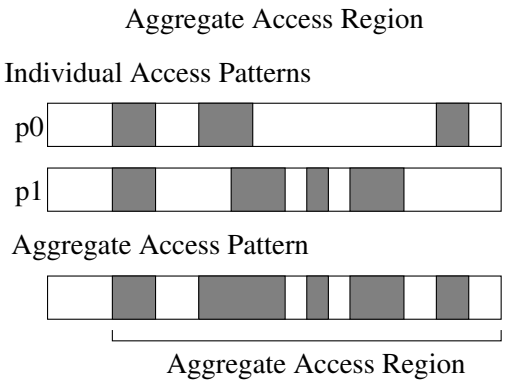

Figure 2.4. Aggregate access region illustrated.

two phase I/O. In the I/O phase, designated *I/O aggregators* in the collective communicator are exclusively responsible for access to non-overlapping sections of the file, and perform I/O on behalf of all the processes. Determining these *file realm* assignments as well as which processes will be aggregators is left up to the implementation, which in turn may choose to defer to the user. In ROMIO, file realms are based on the very first and very last file offset collectively accessed. This *aggregate access region* is depicted in Figure 2.4. Using this method, each file realm is $\lceil \frac{AAR}{N} \rceil$ in length, where N is the number of aggregators. This ought to roughly balance the amount of I/O each aggregator actually does. For clarity, *clients* refers to a process in its capacity as the user source of the I/O requests, and aggregators refer to a process in its capacity as a point of I/O request gathering. All the processes involved in the collective I/O call are clients, but not all them necessarily act as I/O aggregators. The communication phase moves file data to and from the appropriate processes. Whether the I/O call is a read or write determines the order of the two phases. In the read case, data is first read by the aggregators and then distributed to the requesting processes. In the write case, data is first sent to the aggregators and then written to the file by the aggregators. ROMIO uses the data sieving technique described in Section 2.2 to move data between the

actual file and the collective I/O buffer (4MB by default). By combining I/O requests at the aggregators, overall I/O can be made more efficient. The number of I/O calls may be reduced, and the size of the aggregated remaining I/O calls may be increased. IBM's General Purpose File System (GPFS)[**41**] has a feature called data shipping [**36**] that uses I/O aggregators in a similar way, but at the file system level, allowing for use in independent as well as collective I/O routines.

2.4. Remote Memory Access

Remote Memory Access (RMA) interfaces provide what can be thought of as selective shared-memory emulation for a distributed memory environment. RMA in its most basic form allows for the movement of data to or from a remote process without interrupting the host processor. The primary functions of any RMA interface mirror shared-memory operations:

- Get
- Put
- Atomic test&set, atomic swap, compare&swap, or lock

The MPI-2 RMA interface provides two modes. Active one-sided communication requires the use of collective fence functions to ensure communications are complete. The collective nature of this mode rules out its use in a passive environment. The second mode MPI-2 provides is passive one-sided communication. In this mode, non-overlapping RMA calls between an `MPI_Win_Lock` and `MPI_Win_Unlock` are grouped into one atomic set of operations (*epoch*) with no particular internal order.

Portals[**5**] is an open source message-passing library developed at Sandia National Laboratories. Portals is intended as a low-level communication library foundation for MPI

implementations, file systems, job launching, and other subsystems. It is built around a one-sided communication model. The implication of this for an MPI-2 implementation is that its RMA interface is easily mapped to the Portals API. As of Portals v3.3, the atomic swap operation is supported. In addition to a user-level TCP/IP reference implementation portable over most Linux distributions, Portals is natively supported on Cray's XT3/XT4 machines.

2.5. Caching and Coherence

Caching is a good way of hiding the inherent mechanical limitations of disks, latencies, and bottlenecks, but when thrust into a parallel environment, client-side file caching brings its own rather complex problems. Without a cache, data is always up-to-date, but the potential performance benefits of caching make tackling coherence tempting. In file systems, caching can mask the latency of accessing a local or even remote disk by allowing for more responsive reads and writes. Another optimization enabled by client-side caching is write-behind where write data is buffered in the cache while computation continues. Small writes can accumulate in the buffer before they are all written out at once. For local file systems, caching masks the poor latency and throughput involved with mechanical disk access speed, and there are no coherency issues since only local processes use the system.

The core of the cache coherency problem is ensuring globally accessible data is up-to-date when used. Two processes, *p0* and *p1*, may read and cache some shared piece of data. Subsequently, *p0* may write to the data. When *p1* rereads the data, it will read directly from cache and not the new data written by *p0*. This general issue arises at many levels of the memory hierarchy, and is attacked from different angles depending specific features at each level.

Whenever considering client-side caching, it is necessary to also contemplate the kind of semantics to follow. The strict nature of POSIX consistency semantics make them ill-suited for parallel and distributed computing environments. Rigid enforcement of POSIX consistency in these environments is usually at the expense of performance. MPI semantics are, by default, slightly more relaxed for this exact reason. Cache coherency without sequential consistency provides fairly intuitive results. With sequential consistency so hard to provide at the library or file system level, this onus is better left to the application developer who is presumably much better equipped to handle specific ordering needs than any library could be.

Distributed file systems use caching to hide the latency and bandwidth of the network. With caching in a distributed environment, multiple cached copies of data can exist, but normal usage in such an environment is such that files are not usually operated on concurrently. A distributed file system really just needs to be able to let one client at a time cache file data. A typical solution distributed file systems employ to deal with cache coherency is periodic checks with a central server to find out if some cached file is needed by another process. Since checks are only periodic, cached data can sometimes be in an incoherent state. The frequency of the checks determines the probability that data is stale. More frequent checks will lower that probability, but increase overhead associated with the constant checking. It is not unusual for a distributed file system to relax semantics to allow for intermittent incoherence to avoid the overhead of keeping to a strict set of semantics. Typical single user/single client usage of files allows checks with the file server to be rather infrequent. Expiring leases can also be used to ensure only one client at a time has file data cached. The Andrew File System-2[**18**][**40**] uses *callbacks* so client cache invalidation

is server-initiated. A more significant issue in distributed environments is the coherency of directory caches.

In a parallel environment, there are often many more clients than there are I/O servers. Client-side caching can not only provide cached data quicker, but it also reduces the load on and contention for I/O server resources. Parallel environments are similar to distributed ones in that there are multiple clients accessing a single file system. In a parallel environment however, many processes often work on a single problem and concurrently access an output or input file. Allowing a single client at a time to access and cache a file is unreasonable and unacceptable from a performance standpoint. Either multiple client-side caches must be carefully maintained, or there should be no caching at all. Maintaining a client-side cache in a parallel environment is far more challenging than in a distributed one. In a distributed environment, typical usage does not entail multiple clients writing to the same file simultaneously. In a parallel environment, this is exactly the intended use. Ensuring that cached data on each node is safely accessed is not easy to accomplish while maintaining the acceptable performance users demand in the high performance computing space.

CHAPTER 3

High-level Collective I/O Management for Lockless Cache Coherency

In order for I/O systems to achieve high-performance in a parallel environment, they must often either sacrifice client-side file caching, or keep caching and deal with complex coherency issues. A common technique for dealing with cache coherency in multi-client file caching environments uses file locks to bypass the client-side cache. Aside from effectively disabling cache usage, file locking is sometimes unavailable on larger systems.

Because of its relatively high position in the software stack, MPI can effectively tackle cache coherency with additional information and coordination without using file locks. By approaching the cache coherency issue further up, the underlying I/O accesses can be modified in such a way as to ensure access to coherent data while satisfying the user's I/O requests. We can effectively exploit the benefits of a file system's client-side cache while minimizing its management costs.

3.1. Two Phase I/O and Consistency

While the file consistency issue remains fundamentally the same with respect to two phase I/O, the details are more complex since the process making a read request may not actually perform the I/O for itself. Figure 3.1 illustrates this indirection. Figure 3.1 presents two logical views of the file from each client's perspective. The upper buffer is the file data in the application's memory. The second buffer indicates what sections of the file that are

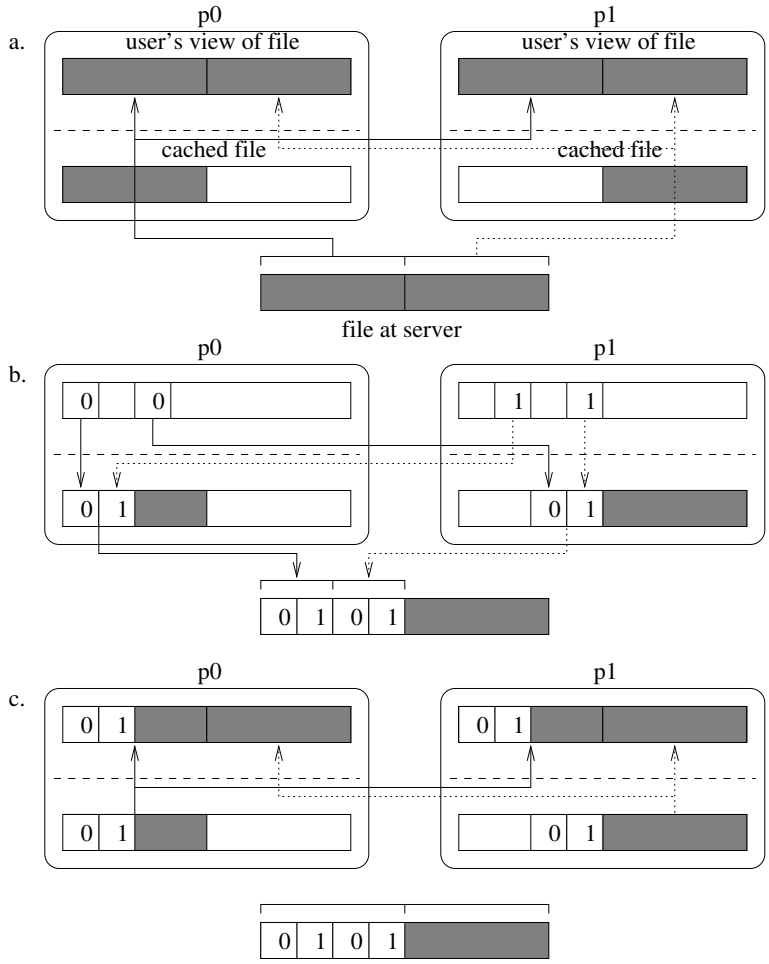

Figure 3.1. Example file consistency problem with collective I/O. In a., the entire file is collectively read. In b., the first half of the file is collectively written, and finally in c., the entire file is collectively read again.

cached in the client-side file cache. With two processes available, *p0* will be responsible for I/O in the first half of the file, and *p1* for the second half, 3.1a. Note that in the initial reads, *p0* has cached the first half of the file, and *p1* the second. Each process will then set a new interleaved view of the file and collectively write only the first half of the file in Figure 3.1b. The new set of file realms reflects the new access pattern, and assigns I/O for the first quarter of the file to *p0*, and the second quarter to *p1*. After exchanging the appropriate

data, the correct data is written to both the caches and to disk. The last collective read is the same as the first, and after the new file realms are assigned, *p0* read its file realm from its client-side cache where the last half is already stale as in Figure 3.1c. This behavior violates both sequential consistency and visibility rules of MPI. Since each process is trying to write exclusive regions in the collective write, the data must be sequentially consistent whether or not atomic mode has been explicitly set. This is the file consistency problem using collective I/O.

3.2. Related Work

The disk-directed collective I/O technique [**21**] allows I/O servers to optimize disk use across process requests. Because collective I/O is synchronized, I/O servers can accumulate the I/O requests from all the processes and then order block accesses in the best way possible. Performance is likely much better than if I/O requests had been handled in the order they happen to arrive in the collective I/O call.

The "Clusterfile" Parallel File System [**20**] uses both disk-directed I/O and two phase I/O together with a global file cache. This tight integration, as well careful attention paid to extracting parallelism from other subsystems such as memory and the global file cache, result in significant performance improvements.

The datatype I/O method [**10**], demonstrates the benefits of passing memory and file datatypes over the network describing non-contiguous regions rather than a one-to-one file mapping as in list I/O [**8**] where each contiguous file access is explicitly listed in some data structure. Operating on the datatypes directly instead of "flattening" the datatypes or entire accesses into offset/length pairs can also ease memory requirements of representing and storing data layout descriptions [**51**]. There is a threshold where storing datatypes does

require more memory than storing the flat offset/length pairs representing the datatypes. The relative efficiency of each method, of course, depends on how exactly datatypes are stored, and the particular datatype in question. By using a stack-based structure to store datatypes instead of a recursive tree structure to describe datatypes run-time recursion can be reduced[**50**].

IBM's MPI library uses an optimization called data-shipping[**36**] in GPFS that is quite similar to Persistent File Realms, but implemented at the file system level. At the system-level, processes may be tied to particular regions of a file. They are responsible for all I/O to their regions, and perform all I/O in those regions on behalf of any other nodes I/O requests. Because it is done beneath MPI it is not immediately portable across systems.

3.3. Intuitive Solutions

While locking is used by ROMIO and GPFS to resolve cache coherency issues, a solution without locking is preferable. File locking itself incurs a high enough cost that some large-scale systems forego file locking altogether. Without locking, there are a couple intuitive file consistency solutions that revolve around circumventing the client-side file cache. The obvious drawback to this is the loss of any performance benefits client-side caching provides. Each read and write will always go over the network to the server for data. Write-behind caches, allow several smaller write requests accumulate before actually sending the data to the network.

One technique for dealing with cache coherency problems lets the application developer explicitly invalidate and synchronize the client-side cache on demand. This functionality is relatively easy to implement and is already in place on Sandia National Laboratory's NFSv2 variant, Extended NFS (ENFS). Invalidating the client-side cache prior to every read

guarantees that stale data is never retrieved by ensuring absolutely no data is ever read from the cache. Synchronizing the client-side cache after every write ensures modified data is written all the way to disk. Following this scheme, MPI consistency semantics can be guaranteed. With the ability to manually invalidate the client-side cache, this responsibility could be left to application developers, or perhaps more desirable, it could be incorporated into an MPI-IO implementation. Cached data will never be read, and data will always be written all the way out to disk, keeping file data consistent.

Another simplistic option would be to completely disable client-side caching behavior of the file system. This in and of itself is a significant burden, and would adversely affect performance on the entire system regardless of whether or not any concurrent I/O is actually being performed. On parallel file systems like PVFS that don't include client-side caching, cache consistency problems do not exist.

3.4. Persistent File Realms

On a file system that includes non-coherent client-side caching, file inconsistencies in the two phase I/O scheme stem from changes in the aggregate access regions (and therefore file realms), in a sequence of collective read and write calls. By making sure file realms remain the same, or persist, over a number of collective I/O routines, each aggregator is guaranteed to access only the latest data. In essence, each process is given exclusive access to its own file realm for the duration of an open to close session. Any data within the process's file realm that it has cached must be coherent since no other process is allowed to modify that data directly. In Figure 3.2 the same example I/O accesses as in Figure 3.1 are used, but with Persistent File Realms (PFRs). The file realms used in Figure 3.2a are used in the

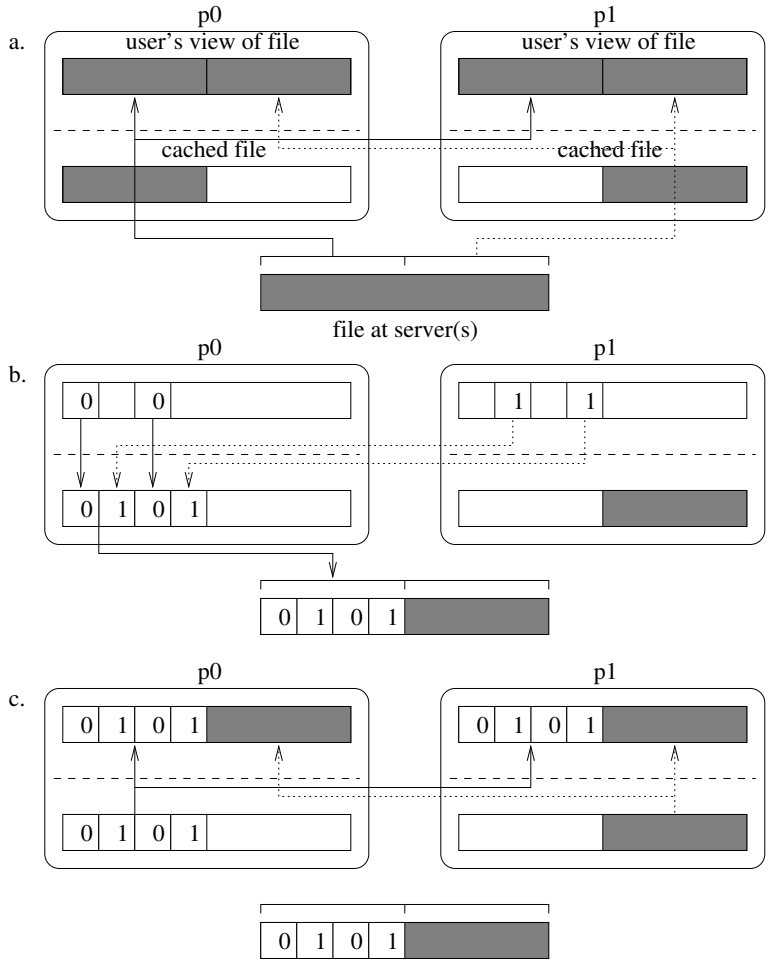

Figure 3.2. Example file consistency problem solved with persistent file realms.

subsequent collective I/O operations, and the data read at the end of Figure 3.2 is now the same as the data residing on disk.

The ROMIO two phase implementation determines file realms using ceiling division to evenly partition the aggregate access region across aggregating processes with the intent of balancing I/O. Using this technique and an evenly distributed aggregate I/O pattern, all the aggregators will do approximately the same amount of actual I/O. Given the noncontiguous and regular nature of the I/O access patterns scientific applications use [**51**], this heuristic

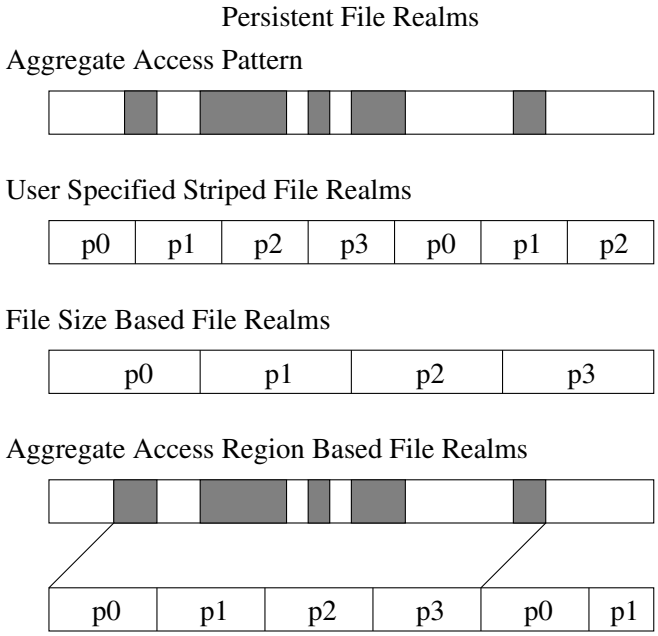

Figure 3.3. Assignment methods for persistent file realms given an aggregate access pattern and 4 processes.

should work well enough and will be considered the ideal I/O balancing scheme for the remainder of this chapter. Better schemes are discussed further in Chapter 4. With this basic balancing scheme in mind, PFR will usually be less balanced than the original ROMIO implementation, but PFR allow MPI applications to safely leverage the performance benefits of the underlying file system's client-side caching. This is the primary advantage of PFRs. So potentially less-than-optimal I/O balance is compromised for safe use of the cache. The results of this tradeoff is application and file system cache dependent. Secondarily, the same aggregators accessing repeatedly accessing the same file regions on the server introduces better locality at the server. This leaves the PFR implementation a decision on how to assign file realms such that I/O responsibility is at least somewhat balanced across collective calls.

3.4.1. User Specified Striped File Realms

By assigning PFRs based on a fixed PFR size, different access patterns can be specifically accommodated. The persistent file realm size passed down by the user through an MPI_File_Hint is used to cyclically assign PFRs to aggregating processes, Figure 3.3. Using this technique, users can optimize the persistent file realm size according to the application's file access pattern. To optimize the PFR assignments, one should try to minimize the number of I/O-communication phases while using as many aggregating processes available to do equal amounts of I/O in each collective call. The main drawback to this method is the user must have a clear understanding of what kind of access pattern the application has, and what the appropriate PFR size is. A PFR size that is too large may result in unbalanced I/O where some processes may not have to do any I/O while only a few must do extra work. A PFR size that is too small will probably even the distribution of I/O responsibilities, but will generate more I/O-communication phases and I/O requests. As long as the file access pattern remains consistent, the performance of the specified PFR size should remain consistent as well.

3.4.2. File Size Based File Realms

Rather than evenly dividing the aggregate access region among processes, the entire file itself could be divided, Figure 3.3. File size based PFR sizes allows the user to concentrate more on the application task, but the file realm sizes may not always balance the I/O responsibilities of each process well. For collective operations that span a relatively large percentage of the file, file size based persistent file realm sizes should perform well unless the file gets very large and the file realm size gets larger than the collective buffer size. A collective operation that

spans a relatively small portion of the file will result in less than the available aggregators actually doing I/O.

A file size based file realm assignment scheme is the most natural extension of the original two phase I/O implementation in ROMIO. While very easy to implement in ROMIO using the existing infrastructure, using such a simplistic implementation presents a couple of interdependent problems. The first is the question of how the size of the file is found at run time, considering a file may be newly created or appended to. Since the original file realms in two phase I/O did not exist outside of a single collective call, this was not an issue. The user in this case would have to provide additional information; namely the largest size a file would reach within an open/close session (i.e. the life time of PFRs). Since these demands on the user were deemed unreasonable, this implementation was used as a proof-of-concept to show PFRs would indeed solve the file consistency problem. Instead we opted to use the implementation of striped file realms in Section 3.4.1 to achieve file size based file realms.

By leveraging the striped file realms implementation, we can emulate the file size based file realms of the initial simplistic implementation, while retaining the more flexible attributes of the striped implementation. In this assignment strategy, a PFR size is determined based on the impending file size during the first collective I/O call. The impending file size is used because a collective write call could create or otherwise alter the size of the file. The PFR size is the impending file size divided by the number of aggregating processes. This way, the user does not need to know exactly how large the file will be in advance since the striped assignment will allow for any later appending to the file.

3.4.3. Aggregate Access Region (AAR) Based File Realms

AAR based PFRs, like file size based PFRs relieve the application developer from having to manually calculating an optimal PFR size. Additionally, AAR based PFRs should accommodate a larger variety of access patterns than file size based PFRs. Both calculate a PFR size at run-time during the first collective I/O call. The difference is that in this assignment strategy, PFR size is size of the AAR of the first collective I/O call divided by the number of aggregating processes, Figure 3.3. In doing this, we aim to achieve the I/O balancing advantages of the original ROMIO implementation, while guaranteeing safe cache access. I/O should remain balanced unless there is substantial variation in the access patterns of subsequent collective I/O calls, since the persistent file realms will have only been optimized for the initial access pattern. This last shortcoming leads to seemingly paradoxical dynamic PFRs.

3.4.4. Dynamic Persistent File Realms

Not every collective I/O call within an open/close session may have the same or similar access patterns. Typically, a change in the memory datatype or file datatype indicates a change in the file access pattern. In dynamic PFRs, this event triggers a recalculation of PFR sizes based on the latest access pattern. Before these new file realms can be safely used however, data in the cache must first be synchronized and invalidated to avoid the possible use of stale data from the old file realms. Whenever a collective I/O call or a new filetype is set, the new memory datatype or filetype is checked against the previous memory or file datatype. In this way, changes in the file access pattern can be accommodated. Datatype

comparison can become rather expensive, so it is best if the application does not change its access patterns frequently.

3.5. Performance Implications

We ran several applications to illustrate the impact of PFRs on performance. The first is an artificial access pattern that should, by design, benefit from cached data. The second application models the I/O requirements of a computational flow problem from the NAS Parallel Benchmarks, and the last simulates the check-pointing I/O behavior of the University of Chicago's ASCI FLASH code. The basic labeling convention is as follows:

- 64K uses a 64 KB PFR size
- 4MB uses a 4 MB PFR size
- fsize uses a calculated PFR size based on file size
- AAR uses a calculated PFR size based on the aggregate access region
- no caching invalidates the cache before every read and synchronizes the file after every write, thereby bypassing the client-side cache
- intelligent uses a user specified PFR size aimed at matching the access pattern (sliding window application only)

3.5.1. Machine Configuration

All of the following tests were run on ASCI Cplant[**12**] at Sandia National Laboratory.

Cplant is an Alpha Linux cluster with each compute node configured with one 600 MHz EV-6, 512 MB of RAM, no disk, and a 64-bit Myrinet card. Each compute node was running Red Hat 6.x with kernel 2.4.x. On Cplant, each compute node is bound to one I/O server in a pool of twelve in a round robin manner at boot-time. Each I/O server runs ENFS,

nodes	intelligent PFRs	AAR PFRs	fsize PFRs
8	6144 KB	3072 KB	20.4 MB
16	4608 KB	3072 KB	12 MB
24	4096 KB	3072 KB	8192 KB
32	3840 KB	3072 KB	6144 KB
40	3686 KB	3072 KB	4915 KB
48	3584 KB	3072 KB	4096 KB
56	3510 KB	3072 KB	3507 KB
64	3072 KB	3072 KB	3072 KB

Table 3.1. Library calculated persistent file realm sizes for the sliding window experiment.

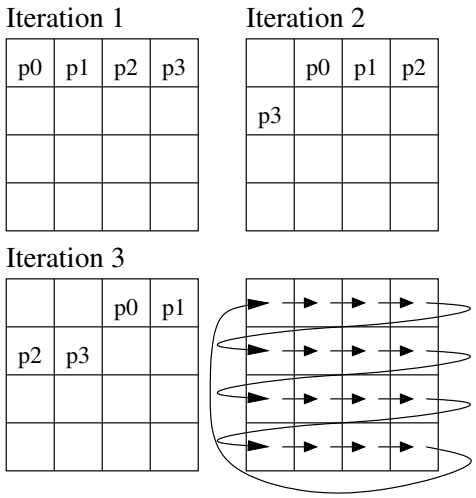

Figure 3.4. Several iterations of the sliding window I/O pattern for a 4x4 example. This continues until each process has read each tile.

a variant of NFSv2 developed at Sandia National Laboratory, in a single global file space. Each I/O server is itself an NFS client to a central XFS server. No caching is done on the I/O servers. Since we did not have the luxury of dedicated I/O servers, we present the best bandwidths from between three to five runs. This implementation of PFRs is developed and tested with MPICH 1.2.4.

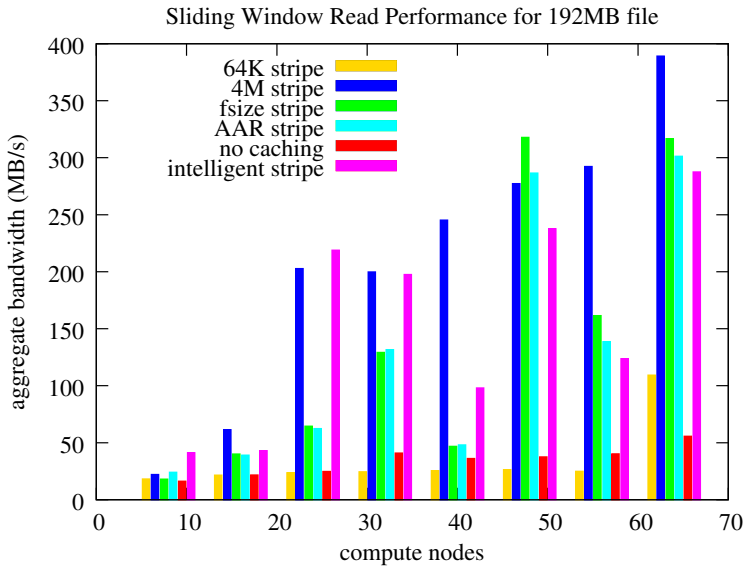

Figure 3.5. For most numbers of clients, the 4 MB PFR allows the most prefetching.

3.5.2. Sliding Window Benchmark

To illustrate the potential benefits of PFRs, we wrote an application with a regularly strided access pattern that should be able to take advantage of cached data. A two dimensional array is broken up into a number of sub-arrays, or tiles. Each process starts at the tile corresponding to its rank. In every iteration, every process reads and then writes its tile. In the following iteration, each process shifts over one tile to the right, wrapping to the next row of tiles (or wrapping to the first row) and perform a read and write to its new tile, see Figure 3.4. This repeats until each process has done I/O on every tile in the array. The same could be accomplished by reading tiles once each, modifying them, and then communicating to the next process over. This manual method may even yield higher performance. If the tiles are small enough, however, we may be able to implicitly capture this data communication with minimal I/O with the much simpler programming model used by our code. For each run, there are 8×8 tiles and each tile is $4096 \times 768 = 3MB$, making the total file size 192 MB.

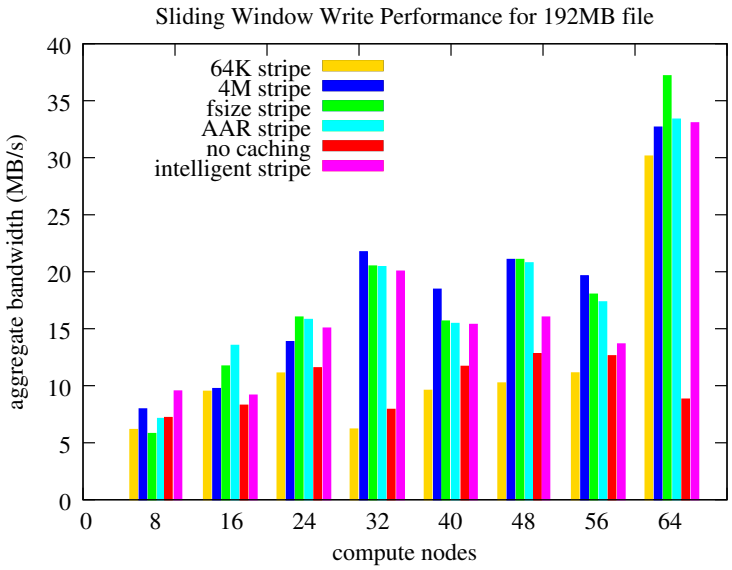

Figure 3.6. Write results are far more consistent between PFR sizes (except for 64 KB PFR size).

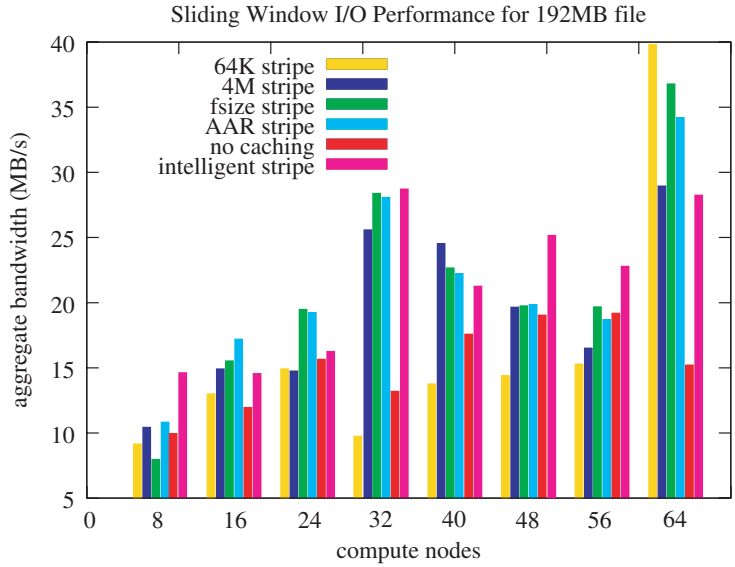

Figure 3.7. Write performance significantly brought down the overall I/O performance of the sliding window application.

The sliding window application was run with the default 4 MB collective buffer size. Table 3.1 lists the different PFR sizes generated for the 192 MB file. The actual amount of total

I/O is nproc*file_size, so this ranges from 1.5 to 12 GB. The key parameter to throughput performance however, is really the tile size, and therefore the file size, since tile size really determines the effectiveness of any given PFR size. The "intelligent" stripe sizes are set by the sliding window application. This would represent a user specifying a PFRs size based on his known access pattern. Given the number of tiles in a row, and the number of processes, a user can determine how many rows the majority of collective I/O calls will span using the following function:

$$\begin{cases} \lceil \frac{x}{N} \rceil + 1 & \text{if } ((x-1) mod N) > \frac{x}{N} \\ \lceil \frac{x}{N} \rceil & \text{otherwise} \end{cases} \tag{3.1}$$

where N is the number of processes and x is the number of tiles in a row. Since in theory, PFR realm division cannot be better balanced than the original ROMIO realms, the improvements over the non-caching implementation can be attributed to the effects of caching. From Table 3.1, one would notice the PFR sizes exceed the 4 MB, the collective buffer size. In these cases, each node must break a collective I/O call into at least two I/O-communication phases. The number of I/O-communication phases is dictated by the smaller value between the collective buffer and the PFR size. During the I/O phase, an aggregator can read no more than the size of the collective buffer or PFR size. The dynamics between the collective buffer, the PFR sizes, and the actual aggregate access regions of each collective I/O call introduce the large fluctuations exhibited by PFR sizes calculated during the initial I/O call in Figure 3.5. Other than the 64 KB PFR size, the PFR implementations consistently out-perform the no-caching case. The library-calculated PFR sizes resulted in the best overall performances. Write performance in Figure 3.6 is much more stable than read performance. Relative features are about the same, but less pronounced.

	9 nodes	16 nodes	36 nodes	64 nodes	121 nodes
AAR and fsize-write	4606 KB	2591 KB	1151 KB	647 KB	343 KB
fsize-read	180 MB	101 MB	45.0 MB	25.3 MB	13.4 MB

Table 3.2. Library calculated persistent file realm sizes for BTIO. The first row is for aggregate access region (rd/wr) and fsize (wr) based. The second row values are for fsize (rd) based.

3.5.3. BTIO Benchmark

BTIO[**30**] is a benchmark from NASA's Advanced Supercomputing (NAS) Division's parallel benchmark suite (NPB 2.4). The Block-Tridiagonal (BT) flow solver provides a representative means of measuring performance of I/O systems. BTIO uses diagonal multi-partitioning realm decomposition to distribute multiple Cartesian subsets of the global data set to compute nodes. The data on each process are three-dimensional arrays, and are periodically written out. The number of these subsets assigned to each process increases as the square root of the total number of processes. This partitioning of data results in I/O that is non-contiguous in both memory and file, a common trait among scientific workloads.

Since the BTIO benchmark only writes out data, we have modified it to alternatively read data with the same access pattern. We used the class B problem size with a $102 \times 102 \times 102$ element array. Compiled with the full_mpiio.f code, BTIO uses derived datatypes and collective I/O to perform its periodic writes as well as verification. The numbers of clients were chosen to approximate the base 2 exponents. Since the initial collective I/O call creates a new file, the file size based and aggregate access region based PFRs end up using the same PFR size. Table 3.2 lists the dynamically calculated PFR sizes. The first row of data applies to the read and write cases for the aggregate access region based persistent file realms and the read case for the file size based persistent file realms. The write runs actually delete the file before writing to a brand new file, so the file size based PFR sizes

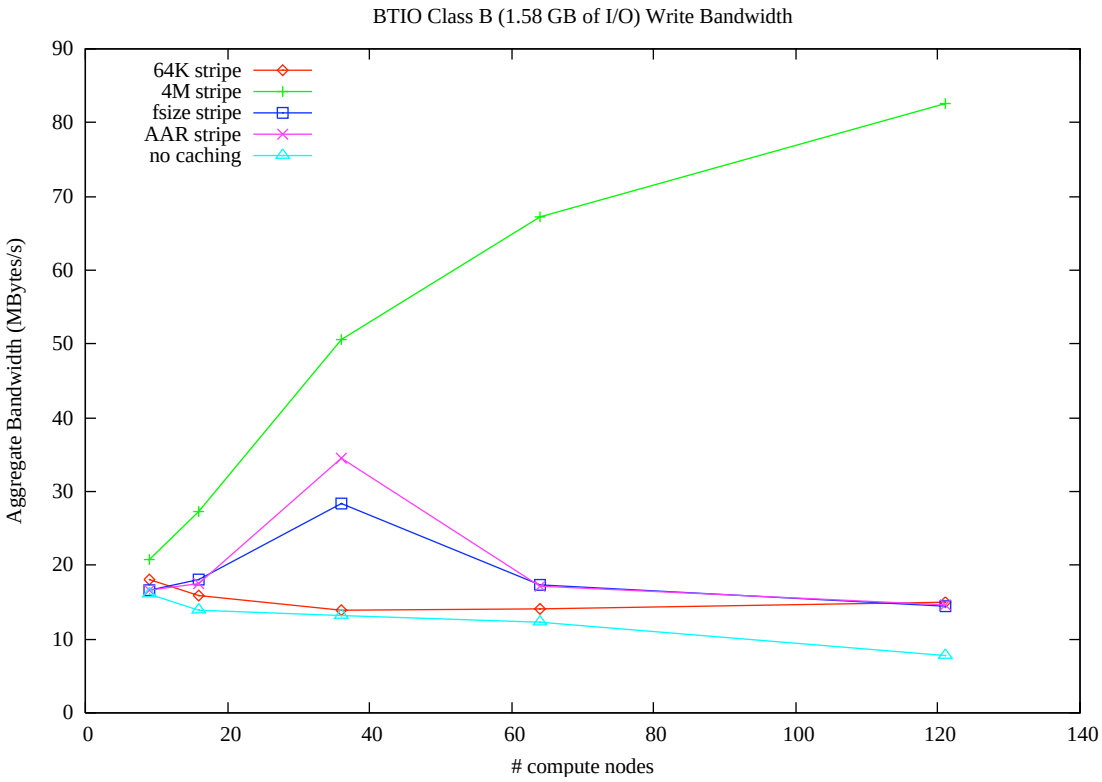

Figure 3.8. The larger 4 MB PFR size allows for more effective use of the client-side cache.

for the write case are the same as the aggregate access region based file realms since that initial collective I/O call is all the file size based method has to go on. For reads, the file must already exist, so the file size based persistent really do reflect the actual size of the file. Because the Class B file is approximately 1.5 GB, the resulting file size based file realms are rather large, ranging from 180 MB to 13 MB in Table 3.2 depending on the number of processes. The dismal fsize results in Figure 3.9 can be explained by the excessively large PFRs. As the number of nodes increases, and the file realm sizes become more reasonable, performance begins to approach that of the other implementations. This is a good example of poorly chosen file realms doing more harm than good.

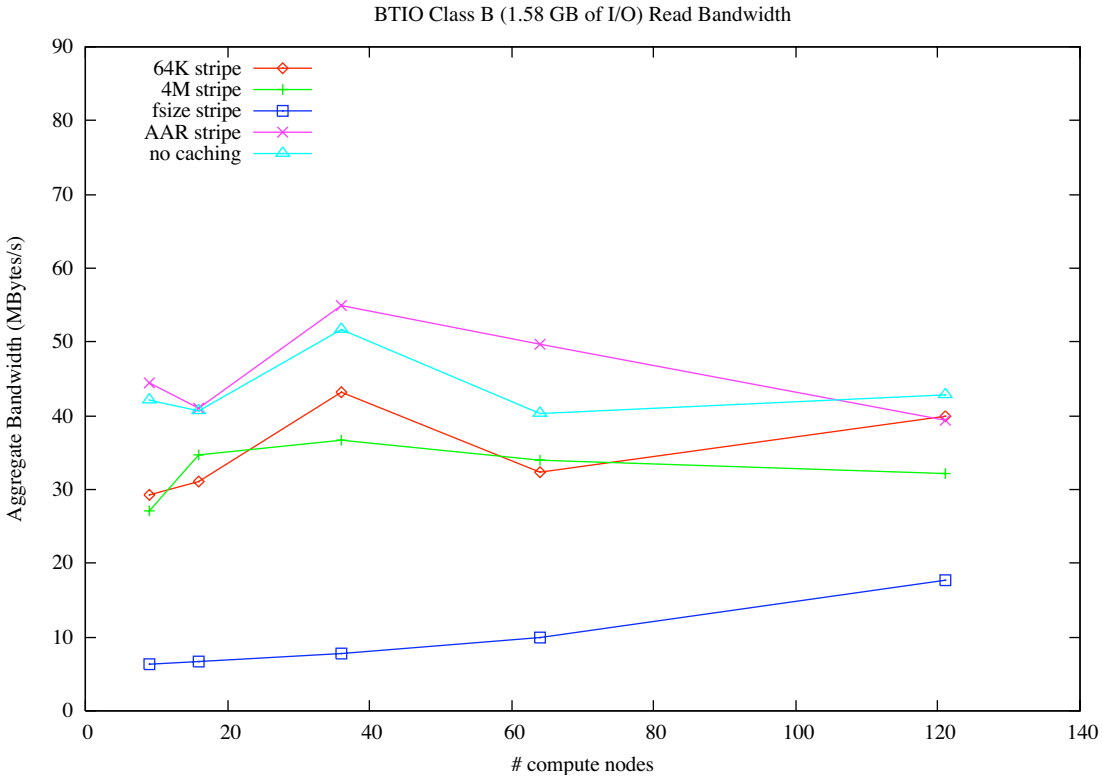

Figure 3.9. The large file realms calculated based on file size result in multiple I/O-communication phases, hurting performance.

If chosen well for BTIO, PFRs seem to offer a marginal amount of benefit over bypassing the cache completely for read performance. For the most part, BTIO's read performance does not benefit PFRs. As far as write performance is concerned, the constricting PFR sizes of the library calculated methods as the number of processes increases keeps performance well below that of the constant 4 MB PFR sizes, Figure 3.8.

3.5.4. FLASH I/O Simulation

The FLASH application[**15**] is used to model several types of thermonuclear flashes: hydrogen flashes on white dwarfs, helium flashes on neutron stars, and carbon flashes within white dwarfs. The FLASH code uses adaptive mesh refinement provided by the PARAMESH

library to increase resolution only in places where it is needed. The cost of checkpointing FLASH is dependent on I/O performance, so I/O performance really determines the frequency of checkpoints.

While the actual FLASH code uses HDF5[**17**] for storage, we mimic the checkpointing access pattern by using the same organization of file variables. Doing so allows us to use MPI's derived datatypes and I/O functions directly. Like BTIO, FLASH I/O is non-contiguous in both memory and file.

Since problem size is directly proportional to the number of processes, the AAR and file size based file realms are always 7680 KB. For every additional client, 80 FLASH (7 MB) blocks are added to the file. So for client counts that range between 4 and 128, the file sizes range between 28 MB and 896 MB. The PFR sizes for AAR and file size based PFRs are the same because the initial collective write is always the same. The 16 MB collective buffer size used easily accommodates all the PFR sizes used. The FLASH benchmark and its I/O access patterns are described further by Ching *et al.* [**9**].

Caching has a minimal impact on the write performance of FLASH except for runs with less than 32 clients, Figure 3.10. Bandwidth quickly plateaus between around 32 clients since the number of I/O servers peaks at 12. Relative to the "ideal" I/O balancing scheme exhibited in the no caching case, the library calculated PFR sizes for AAR and file size based PFRs perform fairly well. The 4 MB and 64 KB PFR sizes generate more I/O-communication phases than the other methods making their performances worse.

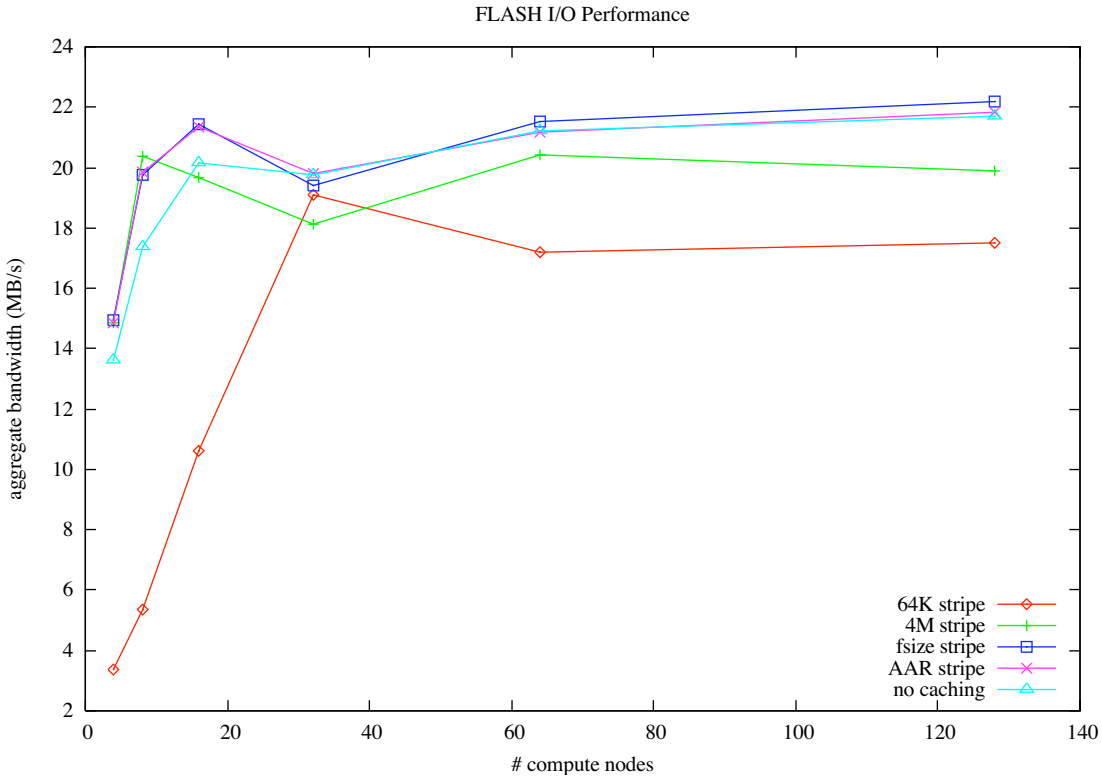

Figure 3.10. Results of the FLASH I/O benchmark with 4 - 128 processors.

3.6. Conclusions

While useful, this work is only implemented in MPI's collective I/O routines. In dealing only with the collective I/O routines, we were afforded some extra luxuries like explicit communication and synchronization. Additionally, the use of PFRs precludes any independent I/O calls since they would have unmanaged access to the client-side cache.

I/O performance of PFRs can be compromised by the size of the collective buffer. The collective buffer essentially acts as a cap on the effective PFR size. Judging by the performance of the aggregate access region based PFR size, it may be more useful when memory is at a premium, otherwise just setting a fairly large PFR size, may be easiest. In some pathological cases, applications are better off circumventing the client-side cache, but in

general, PFRs can offer a significant performance advantage over completely bypassing the client-side cache.

Development on PFRs revealed some challenges and limitations in working with the ROMIO collective I/O implementation.

CHAPTER 4

A New Two phase I/O Implementation for Flexibility and Rapid Experimentation

Work on PFRs and other efforts clearly demonstrated some development shortcomings in the collective I/O implementation in ROMIO. The code was extremely hard to modify to suit different needs. Optimizations for this set of I/O functions can easily become very complex behind the scenes, sometimes limiting both the possible parameter variations that can be easily tested and the number of developers and contributors.

Our new implementation seeks to provide similar behavior to the original ROMIO implementation with respect to the two phase I/O optimization, while simultaneously delivering a cleaner code base and more avenues for research. The key goals for the new implementation are:

- Correctness
- Flexibility
- Developer Friendliness
- Performance

These compose an ideal research platform and important framework for further exploration. As an example, several variations on the two phase I/O optimization are explored.

4.1. Design

The new two phase I/O routines are implemented to reduce communication costs, scale, and take advantage of newer collective communication routines introduced in MPI-2 while retaining the current optimizations.

In addition to occasionally eliminating small inefficiencies, the new design allows for flexible tuning of more parameters. While this may not add direct value for users, it provides systems developers and researchers with many more performance parameters. These parameters can then be tuned at a specific installation, or better yet, automatically determined at run-time with minimal user interaction, so users still enjoy the benefits.

Since supporting the rich descriptions of MPI-IO on any file system is a challenge, any implementation is bound to have a relatively high level of complexity. The necessity of adding in optimizations to realize the benefits of the MPI-IO interface also adds a degree of additional complexity. The new implementation exploits some of the opportunities for code reuse that were left out in the last implementation. The developers of the original ROMIO collective I/O functions had actually opted for less code reuse for a heavier emphasis on performance. While there is no way of completely eliminating complexity from the two phase collective I/O code, the amount of code can certainly be reduced and modularized. The hope is that by making the code easier to study and understand, even more people will be able to use ROMIO as an I/O research platform, and successful ideas can be more easily and reliably integrated into the distribution.

4.2. Changes and Improvements

4.2.1. More Code Paths with Less Code

One potential advantage of the original collective I/O code is its tight integration with data sieving. Since data sieving is implemented directly in the collective I/O routines, the data sieve buffer also serves as the collective I/O buffer. From a performance perspective, this is very good, but from a maintenance and research perspective, it is challenging to have two separate data sieving implementations for collective I/O and independent I/O. The primary benefit of the integrated approach is one less buffer (and the resulting reduction of buffer copies) than the new code which uses the existing independent I/O internal calls and does not directly manage the data sieve buffer. In addition to the maintenance aspect, the new code can also easily leverage any of the optimizations (or fixes) to the internal independent I/O calls. For instance with a simple MPI hint, one could easily use the PVFS list I/O interface through MPI-IO [**7**], instead of data sieving beneath the MPI-IO collective I/O calls. Incidentally, using list I/O would eliminate the double buffering issue since list I/O does not require an extra data buffer. Furthermore, the entire collective I/O call does not need to use only one optimization approach. Within one collective call, an aggregator may have to move data between the collective buffer and storage several times. Because the collective buffer is accessed in a single non-contiguous internal I/O call, it can potentially use a different optimization (preferably the best one) to do so each time. There is more fine-grained control over optimizations even within a single collective I/O call. Neither switching the I/O technique for accessing the collective buffer, nor the fine-grained control of those techniques can be easily added to the present code base. The last code path advantage of the new approach worth mentioning is the more efficient use of the collective I/O buffer.

Unlike the data sieve buffer of the integrated approach, no unnecessary data (data sieving's gap data) resides in the collective buffer.

4.2.2. File Realms

As seen in Chapter 3, file realm assignment is critical to the performance of two phase I/O. These assignments affect the amount of data that needs to be redistributed across the network and the amount of data each aggregator needs to access. The current implementation found in ROMIO partitions the *aggregate access region* evenly among the I/O aggregators. The aggregate access region is the overall start and end file offset of the combined accesses of all the processes. This division and assignment is intended to keep the actual I/O responsibilities of the aggregators balanced so that in any given collective I/O, each aggregator ends up doing approximately the same amount of I/O. Note that this is a heuristic and not necessarily true for every access pattern. This method is best suited for combined accesses that are evenly distributed throughout the aggregate access region. Sparse clusters of data may create severe imbalances where some aggregators perform significantly more I/O than others. Since a collective I/O call can only be as fast as the slowest aggregator to finish, this imbalance is a cause for concern.

By default, the new implementation uses the same aggregate access region based algorithm, but is built in such that any arbitrary set of datatypes can describe the file realms. This feature necessitates a more general-purpose implementation since file realms are no longer assumed to be identical or even contiguous. In the general case, deciding what file realm a particular byte belongs to is no longer a simple $O(1)$ calculation, but a search. Since file realms in the new version are described using a datatype and a file offset (similar to a file view), one can easily plug in a new optimization function to determine the file realms in a

completely different scheme. For instance, aggregator I/O loads could be better balanced for non-uniform access patterns. In hierarchical or very widely distributed systems, file realms could be distributed based on other system factors (such as network proximity). On a BG/L [**42**] machine, it might be advantageous to ensure that aggregators sharing the same I/O node have adjacent file realms, thus improving cache locality on the I/O node.

Because of the flexibility of file realm datatypes, implementing PFRs is a fairly easy task. The original collective I/O implementation had to be heavily modified in order to get persistent file realms to work. PFRs create many more states than the original code had been designed to handle. The primary difference with PFRs is file realms need to designate region assignments for the entire file, not just the region being accessed in one collective I/O call.

4.2.3. Storing, Processing, and Communicating Datatypes

Both data communication and I/O request communication are significantly overhauled. The basic terms used are:

- M is the total number of offset/length pairs for the access
- A is the total number of aggregators
- m_i is the number of offset/length pairs generated for aggregator i where $\sum_{i=0}^{A} m_i = M$
- D is the number of offset/length pairs to represent the datatype

The original I/O access communication is discussed first, and for simplicity, the discussion is in terms of a single client and multiple aggregators. The basic steps involved are as follows:

- client: flatten entire I/O access pattern into offset/length pairs (M)

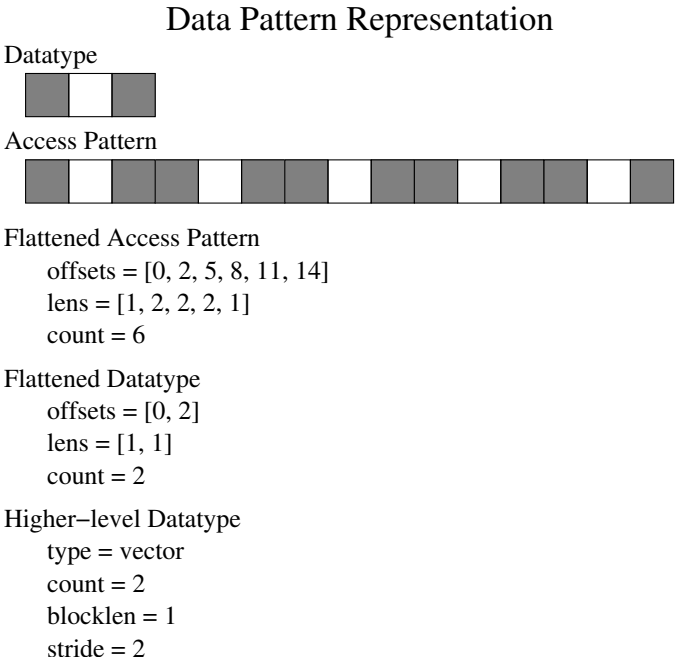

Figure 4.1. While the storage size of data patterns is important, one must consider any extra processing overhead involved. With respect to just storage size, the particular pattern determines which representation is best.

- client: send appropriate offset/length requests to each aggregator (M)
- aggregator: receive applicable requests for its file realm from each client
- aggregator: processes each offset/length pair it receives (m_i)
- client: processes each of its offset/length pairs as aggregators make requests to send or receive data(M)

This basic algorithm is quite computationally efficient since the number of offset/length pairs the client has governs its run-time from the client perspective as well as the aggregator perspective. In the actual implementation, some user's offset/length pairs may be broken up, making the number of offset/length pairs dealt with more than M, but does not increase

algorithmic complexity from $O(M)$. Space and communication time, however, are also linearly correlated with the number of offset/length pairs. For access patterns with numerous offset/length pairs, this memory cost can be prohibitive.

Potentially better alternative representations include storing the offset/length pairs of the datatypes themselves and storing the datatypes in an even higher-level description as in Figure 4.1. Depending on the particular datatypes themselves, any one of the representations may be more space efficient than the other. The penalty for these more succinct access storage mechanisms, however, is more processing. The new algorithm uses flattened datatypes and looks like this:

- client: flatten entire filetype into offset/length pairs (D)
- client: send flattened filetype to each aggregator
- aggregator: receive flattened filetype from each client
- aggregator: process flattened filetype looking for relevant data (M)
- client: process flattened filetype separately for each aggregator (MA)

The final step introduces extra work on the aggregator. Instead of simply being informed of the required accesses, it must calculate them itself. For the old scheme, between the client and aggregators, all offset length/pairs are processed in $O(M)$ times where M is the total number of offset/length pairs. In the new scheme, $O(MA)$ offset/length pairs are processed where A is the number of aggregators. The client needs to keep track of how each aggregator is progressing independently, so it must process its offset/length pairs once per aggregator. Conversely, each aggregator must basically process the client's entire access. While the aggregators can work in parallel, the client cannot, so the run time is $O(MA)$ as opposed to the $O(M)$ run time of the original collective I/O code. On the client side,

a binary heap is used to mitigate this and achieve $O(MAlogA)$ time, and because of the iterative nature of datatypes, not all M offset/length pairs need to be processed. The latter technique, however, does not yield an algorithmic run-time change. Since the new file realms are simply datatypes, these are processed in a similar manner. The generalization of the interface results in a performance tradeoff.

While the original collective I/O code is very efficient from the computation standpoint of processing I/O requests, memory and communication requirements can become quite high for access patterns with many pieces. By only storing the offset/length pairs of the datatypes describing the access patterns and passing them around, memory and communication overheads are reduced. The cost of this design choice is the additional computation of aggregators processing each client's filetype. In the context of checkpointing (where collective I/O is particularly well-suited), taxing the processor is acceptable because it is likely not doing any application computation during the checkpoint.

4.2.4. Data Communication Optimizations

Although the original collective I/O code uses non-blocking communication frequently, it still performs all the communication at once. It first posts all the `MPI_Irecv`s, then posts all the `MPI_ISend`s, and then waits until all communication is complete. The new code uses either `MPI_Alltoallw` (an MPI-2 function) or non-blocking communication. `MPI_Alltoallw` allows aggregators to perform non-contiguous communication directly from the collective buffer. Similarly, it also allows consumer processes to do communication directly from the user buffers. Some recent high-performance computing architectures, most notably IBM's BG/L, incorporate a separate network, or are at least highly optimized for, collective communication. In case such a specialized network is not available, the communication phase

is also implemented with non-blocking communication in such a way as to overlap data communication with internal computation (e.g. file and memory address calculations).

4.3. Performance Testing

4.3.1. Machine Configuration

All of our tests were run on ASC Vplant at Sandia National Laboratories. Vplant is a large Linux cluster where each compute node has dual-processor 2.4 GHz Pentium 4 Xeons with 2 GB of main memory and Myrinet interconnect. Since compiling MPICH2 with GM is tricky, TCP/IP was used. Tests were performed using the Linux 2.6.9 kernel running on a Lustre[**25**] file system. Since the file system is shared between several clusters, each with many users, the best result out of five runs is reported for each experiment. While this methodology should better represent performance characteristics, it is also more susceptible to outliers and transient behaviors.

4.3.2. HPIO, Scalability, and Smarter Datatypes

HPIO[**11**], is a very flexible I/O benchmark useful for both performance and verification tests. HPIO builds regular datatypes that are characterized by a region size, count, and spacing. These datatypes are used to represent memory and/or file. The most common I/O patterns found in scientific computing involve non-contiguous access to files[**22**]. For Figure 4.2, data in both memory and file are non-contiguous. Regions of data are separated by 128 bytes of space, and there are 4096 regions per client. The amount of aggregate data accessed is between 2 MB and 1 GB. The three I/O methods are the new collective I/O code run with a very succinct MPI "struct" datatype describing the non-contiguous patterns, the new collective I/O code with an MPI vector type explicitly enumerating the entire access, and

Figure 4.2. While the new collective I/O implementation provides comparable performance to the old implementation in many cases, in other cases, the new implementation fares significantly worse. This is due primarily the additional overhead required to process datatypes.

finally, the original collective I/O code also with the MPI vector type. Since the original code flattens the entire access out, using the struct type with it makes no difference.

The consistently better performance of the new collective I/O with the struct datatype versus the new code with the vector datatype comes from several places. As discussed in Section 4.2.3, using the shorter struct datatype should reduce the amount of data transferred over the network to exchange data layouts. Most of the difference in performance, however, is attributed rather to more efficient processing of the struct datatype than the full vector datatype. With the full vector datatype describing the entire access, clients must stop and evaluate each offset/length pair. An internal optimization allows processes to skip full datatypes in evaluating offset/length pairs, so that with such a succinct data description, processes may be able to skip many offset/length pairs while the runs with the vector datatype could not. As region sizes get larger, I/O time becomes a more significant factor, mitigating the benefits of the struct type. This is quite apparent when HPIO was run with only 8 aggregators.

Although the new collective I/O implementation fails to consistently match the performance of the older implementation, performance with the struct type is comparable in about half of the cases. MPE logging was used to identify the slow parts of the code, and it turns out that the main cause for the differences is the additional computational overhead tied directly to the number of aggregators. Double buffering between the separate collective and data sieve buffers also contributes to the performance differences. In the 8 aggregator case, the differences are pronounced because with fewer aggregators, each aggregator has to do more I/O, repeatedly going through both the collective and data sieve buffer for each I/O request to the file system. As the number of aggregators increases, the extra buffer copies become less significant, but the amount of computation increases.

With respect to performance, there is some room for improvement, and this is expected to come as the new code matures. The slight loss in performance is part of the trade-off for

opening more research opportunities by providing a relatively easy means for testing new optimizations as demonstrated in the following results sections. Inevitable future improvements in processor performance will reduce the significance of this additional computation.

4.3.3. Conditional Data Sieving with HPIO

In this experiment, the idea of conditional data sieving is introduced to the collective I/O routines using a simple metric. Aggregators can be directed to conditionally use different I/O techniques to access the file based on a characteristic or characteristics of the access pattern. In this test, two methods for writing contiguous data in the collective buffer to non-contiguous file space are tested. One uses a data sieving routine, and the other uses a naïve routine that fulfills each contiguous request to the file with its own file system I/O call. In each graph, the datatype extent is held constant, and between graphs the file size is held constant at 1 GB. The amount of aggregate data accessed grows linearly along the X-axes from 32 MB to 1 GB. The initial theory tested is that a good means for determining when to use data sieving is based on the relative amount of useful data acquired in each file system I/O call. Our experiments in Figure 4.3 show that for HPIO, the extent of the datatype is more indicative of which method is more efficient. Naïve I/O beneath two phase is more suitable than data sieving for datatypes with larger extents. Conversely, data sieving is more efficient for smaller extents. The crossover is right around a 16 KB datatype extent. This threshold is very easily implemented so that user need not worry about where these crossover points are. In HPIO, extent is directly correlated with how many I/O requests data sieving will save. The spikes clearly evident in the naïve case are a result of I/O access alignment. They occur in 4 KB intervals, the page size for Lustre. The final spikes at the 100% points are a result of "the contiguous in memory to contiguous in file" code path being

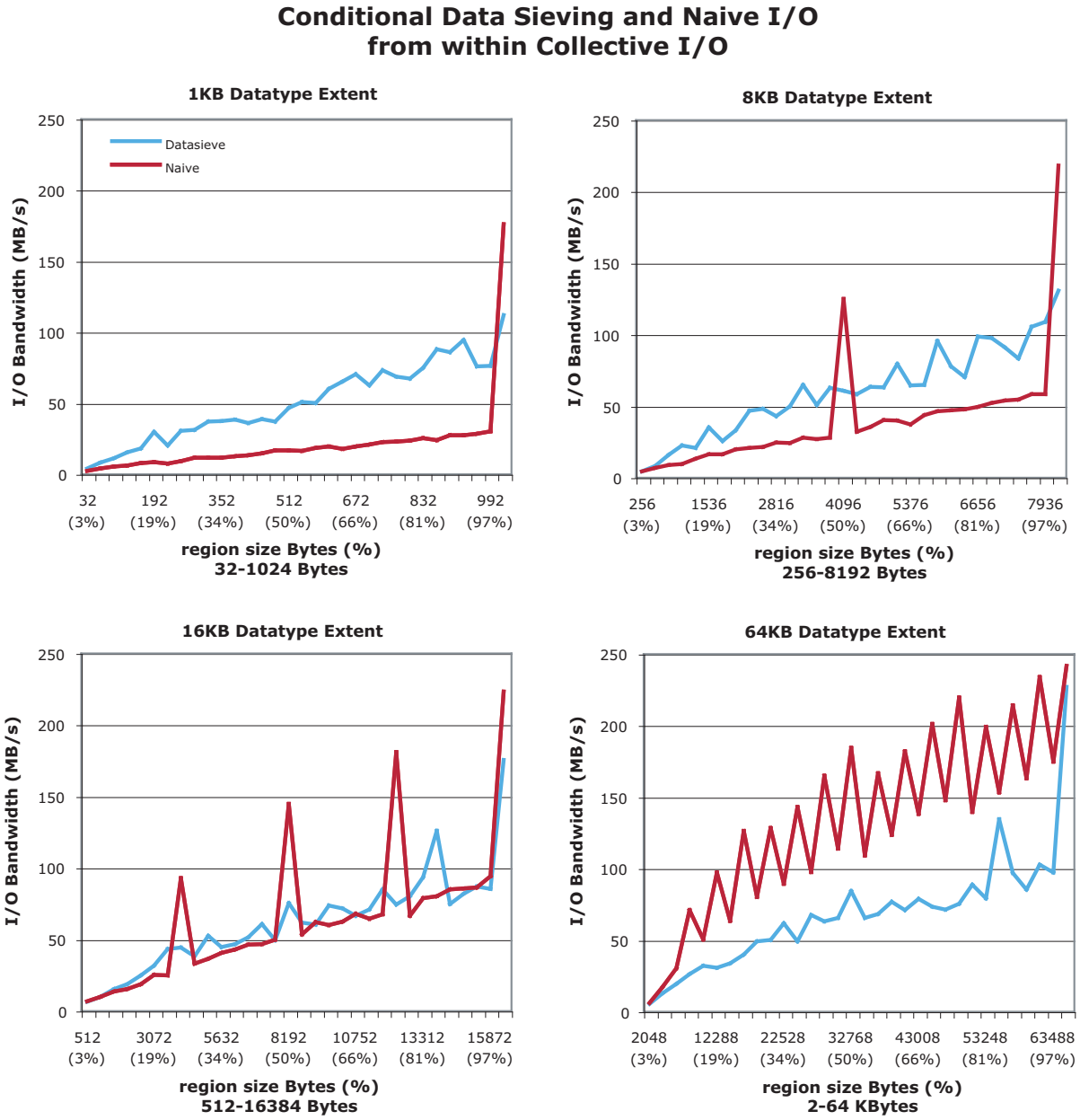

Figure 4.3. The percentages on the X-axes indicate the amount of useful data in the datatype relative to the entire extent of the datatype. This percentage is not as important a factor as the actual datatype extent in deciding whether or not to use data sieving or naïve I/O to fill the collective buffer. The regularly spaced spikes are a result of I/O aligning nicely with the 4 KB page size on the file system.

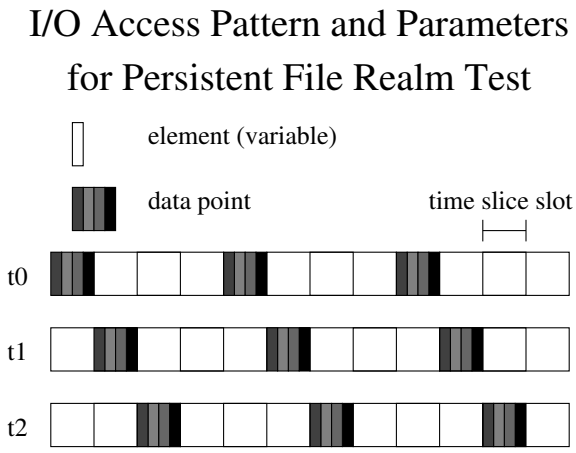

Figure 4.4. One non-contiguous collective write call is made per time step. Each data point is accessed with an interleaved pattern - in this case, four processes access an element each in every data point.

taken where computational overhead is significantly lower for two phase I/O. The conditional formula used to determine the best I/O technique could easily accommodate the alignment behavior as well. The percentage of useful data in the data sieving operations may still become a factor for a very narrow band of extent values somewhere between 8 KB and 16 KB. These exact numbers, of course, are unique to the particular system used. This set of experiments merely demonstrates the potential for a number of parameter and combinational optimization studies enabled by the new implementation. More in-depth analysis of these behaviors is warranted, and it is much easier to do.

4.3.4. Persistent File Realms and File Realm Alignment

In high-performance computing, files are often written but never read back in the same application. The PFR performance test code uses this principle to demonstrate the usefulness of an incoherent client-side file system cache in a write-only scenario. The analogy used in the construction of the program consists of a number of multi-variable data points distributed

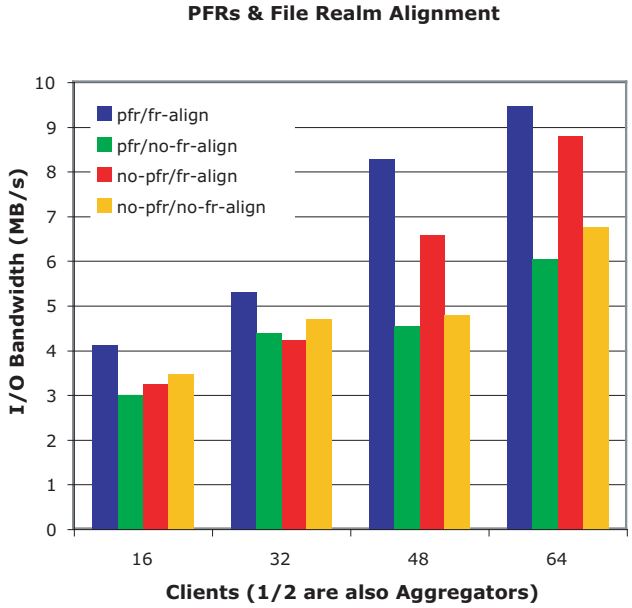

Figure 4.5. Careful alignment of file realms can ease the burden on file systems trying to maintain cache coherency and/or strict POSIX semantics.

across the file where all the time steps for the same data point are kept together. The time steps can alternatively be thought of as a third spatial dimension, and seeing as the time steps are not indefinite, a fixed three-dimensional space with multiple variables makes sense as well. The size in the third dimension determines how many I/O calls are made. This access pattern is illustrated in Figure 4.4, and is similar to what may be generated by a higher-level library such as netCDF [**38**]. Such a pattern ought to make good use of a client-side write-back cache.

File realm alignment is another easy optimization made in the new collective I/O routines. By aligning file realms with the file system stripe size (and more importantly, the page size), much of the overhead involved with locking and cache coherence in the underlying file system can be avoided. In this experiment on Lustre, file realms are aligned with the 2 MB stripe

size. The alignment is added as a new ROMIO hint, as there is really no portable way of determining the stripe size at run time.

One thing is clear from Figure 4.5, using PFRs with file realm alignment is a definite win. File realm alignment allows for nice accesses with respect to Lustre, and PFRs improve locality. Direct comparisons between each method are less straight forward, however. The nominal bandwidth numbers are low because the access pattern consists of sparse small data, and because data sieving is always on. The PFR code was run with 32 byte elements, 100 elements per data point, 2048 data points, and 32 time steps on Lustre. Under these options, only 6.5 MB of aggregate data is written in one collective write. Half of the total clients were also used as aggregators. Working on the assumptions that using file realm alignment is better than not, and that using PFRs is better than not, one would expect that using either of them should always be better than using neither. Experimentally, this is not always the case. Except for the results using just file realm alignment for 48 and 64 processes, using just one optimization is actually worse than not using any at all. This is an artifact of each of these cases generating slightly differing file realms. Without PFRs, the file realm sizes may remain the same as with PFRs, but the starting points move along the file with the actual access region, the PFR file realms are always anchored at byte zero of the file. PFRs establish a file realm assignment for all of the collective writes (because the file realms are anchored at 0), and without file realm alignment the Lustre lock manager is still engaged in lock granting and revocation. Similarly, without PFRs and with file realm alignment, the file realm alignment pays off for 48 and 64 processes, but with only 16 or 32 processes, the file realms that file realm alignment produces are just too imbalanced to effect a performance improvement.

4.4. Conclusions

Performance is certainly one deficiency, but should get better with further development and faster processors. By using datatypes to describe file realms, file realms can be further optimized for the general case as well as carefully tailored for specific applications, systems, and environments. Though the use of datatypes entails more processing overhead, for very large applications and large amounts of data, the network, memory, and I/O will likely be the predominant constraints. Better I/O aggregator load balancing is one obvious opportunity to leverage datatype based file realms, as well as more complex means of determining the appropriate I/O optimizations at run-time with little to no user intervention. The opportunity for improvements like these is the fundamental purpose of the new collective I/O implementation in its role as a powerful framework for future I/O research.

CHAPTER 5

Direct Access Cache (DACche) System Prototype

Client-side file caching in large-scale parallel environments is challenging from the semantics perspective. Depending on end-goals and priorities, a balance must be struck between semantics and scalability. Relaxing coherency and consistency requirements can greatly improve scalability and performance. By default, DACche enforces cache coherency and supports an optional sequentially consistent atomic mode as well. The optional consistency feature is based on two phase locking and subject to false-sharing, *etc.* DACche is designed and developed with three primary characteristics: scalability, coherency, and a passive architecture.

Caching at the MPI-IO level falls in line with the latest trend in high-performance file system design to off-load (or at least allow for off-loading) many responsibilities traditionally shouldered by the file system like security and semantics on libraries closer to and tuned for applications' specific needs. The Light-Weight File system [**33**][**34**] developed at Sandia National Laboratories is one such example. The same trend of trimming system services can be observed in the operating system space. Compute nodes operating systems like IBM's compute node kernel on BG/l and Cray's Catamount on the XT3/4 are both very lightweight operating systems that both happened to forgo threads. This basically forces any service-like scheme in the application space to use passive RMA.

As with any cache system, the performance benefits of DACche are heavily influenced by the I/O characteristics of the application being run. Any application that repeatedly accesses parts of a file either with reads, writes, or some mix, stands to gain from caching.

5.1. Related Work

IBM's General Parallel File System (GPFS) manages cache coherency with its distributed lock manager[**2**]. On extremely large-scale systems, the overhead and coordination of the file system level lock manager make it quite a performance hindrance.

Panasas[**29**] maintains cache coherency through a callback system based on its metadata servers. While file data flow is directly between I/O servers and clients, a client must register its read/write intentions with the metadata servers so callbacks can be made should a region of file become dirty.

DACche is quite similar to the work on active buffering [**26**], the primary differences being the lack of locally attached disks for caching and thread support. DACche is targeted at very large computers running lightweight operating systems. In the process of trimming down such a specialized operating system, thread support is often left out. This is the case for both the Cray XT3/4 and IBM BG/L[**1**] architectures. Both active buffering and DACche have the effect of deferring writes; however active buffering uses threads to overlap I/O with application execution. Local disks in active buffering gives each process virtually unlimited cache space (relative to memory). Again, trends in large-scale architectures make attaching a disk to each compute node infeasible.

ENWRICH[**37**] is a client-side file cache that is used in conjunction with disk-directed I/O. ENWRICH is notable for being log-based and not block-based, avoiding any false-sharing issues. Writes are time-stamped and aggregated in the client-side cache. During

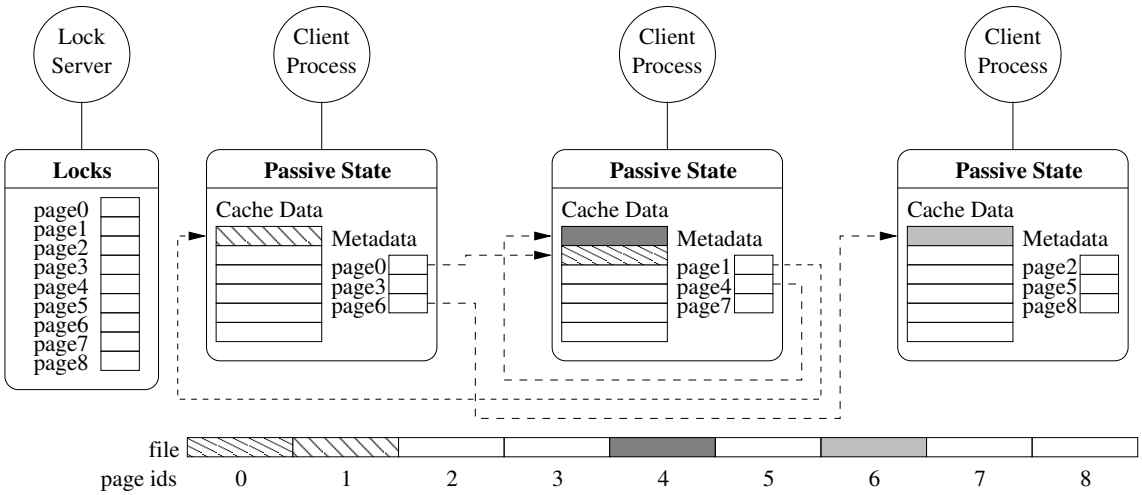

Figure 5.1. DACHe architecture with passive metadata and cache servers on each client. Metadata is striped across clients, but a file page can be cached on any client.

a flush, processes collectively send the appropriate metadata describing their cache state (time-stamp, file, file-region, *etc.*) to the I/O servers. The I/O servers themselves resolve any consistency issues using the time-stamps to determine order, and tell the clients what cached data to send. Since consistency is checked before any file data is actually transferred, ENWRICH can avoid sending unnecessary data that is overwritten anyway.

5.2. DACHe Design

From a design perspective, DACHe can be architecturally divided into 3 primary subsystems: cache metadata, locking, and cache management. Its modular implementation makes it quite easy to port and experiment with, given that the basic RMA requirements are met. One over-arching theme always under consideration during the design of DACHe is to keep all aspects of cache management as decentralized as possible. A secondary theme is minimization communication where possible. The main tenet of DACHe is that only a single copy of any file data can reside in any file cache. This single-copy rule ensures cache coherency and

removes the task of maintaining state for replicated data. Another way to think of it is that there is never more than one usable copy of any given file page. The use of RMA keeps I/O operations in DACHe passive, but requires that `dache_open` and `dache_close` be collective in order to set things up and break things down safely. After `dache_open` completes, all communication is one-sided unless it is with a mutex server described in more detail in the Mutual Exclusion 5.2.2 subsection. The decision to make DACHe block-based is based on the limitations of one-sided communication. A block-based cache is much simpler to manage, and without explicit coordination, managing a more free-form cache would be an exceptional challenge.

The three subsystems in DACHe warrant a closer look to understand how DACHe works. Figure 5.1 illustrates the basic interactions between these subsystems.

5.2.1. Cache Metadata

Cache metadata maintains basic state for each page in the file. Most importantly, this metadata provides the whereabouts of any given file page. File page refers to the logical partitioning of the entire file into blocks of a size matching the page size of the cache. If a page is cached on any process the metadata reflects the caching process as well as an index location into that process's cache. Cache metadata is remotely accessible through RMA and distributed across the application nodes in a deterministic fashion: in this case a basic striping algorithm. By striping the metadata array, or table as it will be called, across nodes deterministically, not only is a potential bottleneck avoided, but there is also no additional communication required to find the metadata associated with any file page.

Creating metadata for each logical file page brings up the issue of metadata allocation. This allocation process requires explicit coordination amongst the application processes, and

this is only available at the collective `dache_open` and `dache_close` functions. Ideally, one would want the size of the metadata table to be directly related to the size of the file. What this basically entails is some level of cooperation among processes for growing or shrinking the table size during run time. Since the write operations are independent however, there is no opportunity to coordinate all the processes in order to modify the size of the table. Without this coordination opportunity, the last resort would be the ability to remotely allocate globally accessible memory. Needless to say, remote memory allocation brings its own set of challenges. For these reasons, the default maximum file size is simply 2 GB, with this value being settable by the user.

Given the distributed and passive nature of cache metadata, mutually exclusive access to metadata is crucial.

5.2.2. Mutual Exclusion

The purpose of the mutual exclusion subsystem is to ensure safe access to read and modify cache metadata. Ideally, mutual exclusion is directly supported in the RMA interface. With RMA support for mutual exclusion, the locking subsystem can be also distributed across the application nodes as passive remote accessible state.

In the absence of an RMA solution to mutual exclusion, one or more processes from the allocated user processes are siphoned off from the main group to act as dedicated mutex servers. They are spun off during `dache_open` and returned during `dache_close`. During this time, the mutex servers cannot execute any user application code. While passive RMA mutual exclusion is preferred, DACheche can still be evaluated using dedicated mutex servers.

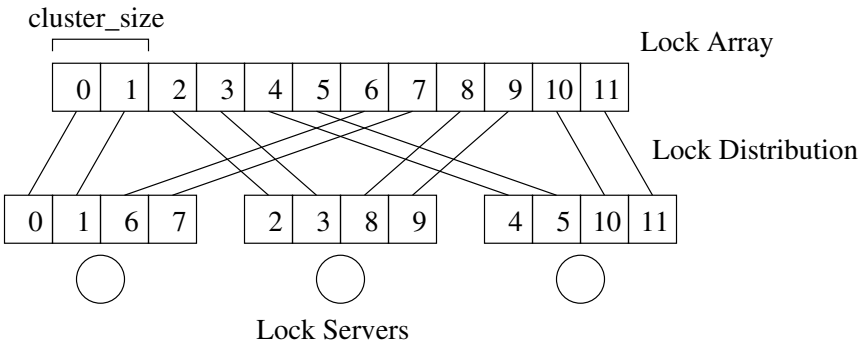

Figure 5.2. Locks, whether passively implemented or on dedicated processes are round robin distributed across the lock hosts.

Eventually, these mutex servers become passive elements on the application processes using the locking techniques described in Chapter 6; in either case, lock responsibilities are distributed across multiple lock hosts as in Figure 5.2.

Initial evaluation of the DACche architecture used dedicated lock servers, but because mutex responsibilities were intended to reside on the clients, the dedicated mutex servers were intentionally kept very simple. Lock responsibility for each file page is spread across the mutex servers in the same way cache metadata is spread across application processes. The mutex servers service locks in the order they come, queuing requests to the same cache metadata. A process must block until the mutex server fulfills its lock request.

Typically, cache metadata is not "held" for extended periods of time. It is locked only briefly for modification. It is not held for the duration of access to the cached data itself. Minimizing the time that the metadata is locked should reduce the amount of simultaneous lock requests to the same metadata. The exception to short lock times is when a particular file page is being brought into the cache or a cache page is being evicted. In either case, the cache metadata must be locked for entire I/O operations in order to prevent early or late cache accesses, respectively.

5.2.3. Caching

Pages cached on one process are globally accessible to any other process through RMA. Although access to metadata is carefully mediated, remote access to cache data is basically a free-for-all. Since any file page can be cached in at most one cache, all accesses to that page are coherent.

Cache management and eviction is handled locally with one exception to be discussed a little further along. What data to cache is determined by the local process's I/O accesses. If a process accesses a file page that is not yet cached on any of the other processes, it caches it itself and updates the corresponding cache metadata to reflect this change. Should a process run out of cache space locally, it must evict a page based on some local policy such as a least recently used (LRU) policy. During eviction, a page's metadata cannot be accessed at all. Another precaution alluded to earlier is that processes accessing a remotely cached page must "pin" the page in cache so the page cannot be evicted while being accessed. This pin is a semaphore contained in the cache metadata along with location information. While a page is pinned, the process on which the cache page resides cannot evict the page, and must either wait until the page is "un-pinned" or try to evict a different page. New data is not written to disk until it is either evicted or written out at `dache_close`.

5.2.4. Results on the DACche prototype

The initial DACche experiments were run on ASCI Cplant[**12**] at Sandia National Laboratory. Cplant was an Alpha Linux cluster where each compute node is configured with one 600 MHz Alpha EV-6, 512 MB of RAM, no disk, and a 64-bit Myrinet card. Each compute node ran Red Hat 6.x with kernel 2.4.x. Early experiments with DACche use a synthetic I/O access

Cache Size/proc	4 MB
Page Size	32 KB
Chunk Size	64 KB
Loops	40

Table 5.1. DACHe and sliding window parameters

pattern that should, by design, benefit from cached data. It is roughly based on the regular noncontiguous access patterns often found in scientific applications and is meant to test the performance of DACHe. The benefits a real application may derive from DACHe are also clearly of interest. The basic labeling convention is as follows:

- mtx-n where n is the number of mutex servers
- DACHe clients is the number of clients actually caching data (non-mutex servers)
- 50:50 refers to an equal mix of clients and mutex servers

The sliding window application from Chapter 3 is used again to test the effectiveness of DACHe. The underlying caching library uses the lock service to gain exclusive access to cache page metadata. Since there is one lock per page, and the benchmark accesses are page-aligned, the number of metadata locks is tied directly to the number I/O operations. The cyclic access pattern makes the amount file system calls increase along a second order function rather than linearly, as seen in Figure 5.3. Direct file system calls that the sliding window application follows the "direct" curve, while the number of file system calls as mediated by DACHe follows the "DACHe" curve. This reduction is achieved transparently from the application point of view. For this specific example, the same effect could be attained by explicit communication in application, but increases application complexity.

It is important to note that the DACHe system is the primary subject of evaluation, and as such, actual I/O is removed from DACHe to prevent interference from any specific

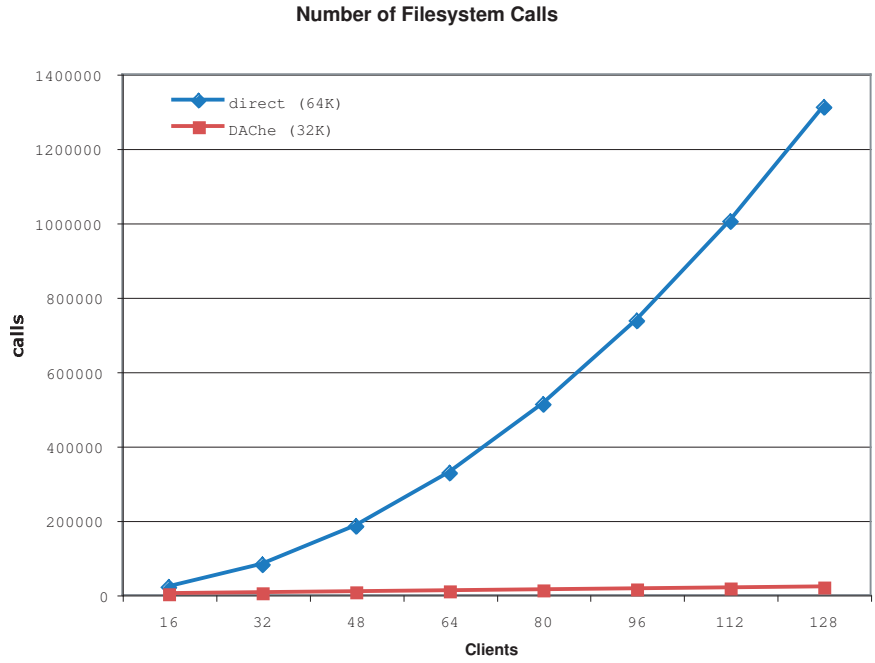

Figure 5.3. The amount of I/O grows as a second order function on the number of clients.

file system. The sliding window access pattern is such that once a file page is brought into cache, it remains there throughout the execution and is only remotely accessed over the network. Though actual I/O would have been performed to bring in the file data, all subsequent accesses really access cached data, possibly on remote nodes. Throughput is calculated based on the number of I/O calls that would have been made to the file system. Since a fixed number of lock requests are generated per cache page access, the number of locks requested by each process is directly proportional to the amount of I/O done.

Since scalability is the primary goal of DACHe, it is tested with various numbers of mutex servers running. Intuitively, the heavier the lock system is taxed, the more mutex servers are needed to accommodate the increased load. This is illustrated clearly in Figure 5.4 where a larger number of mutex servers allows a larger number of clients without severely hindering

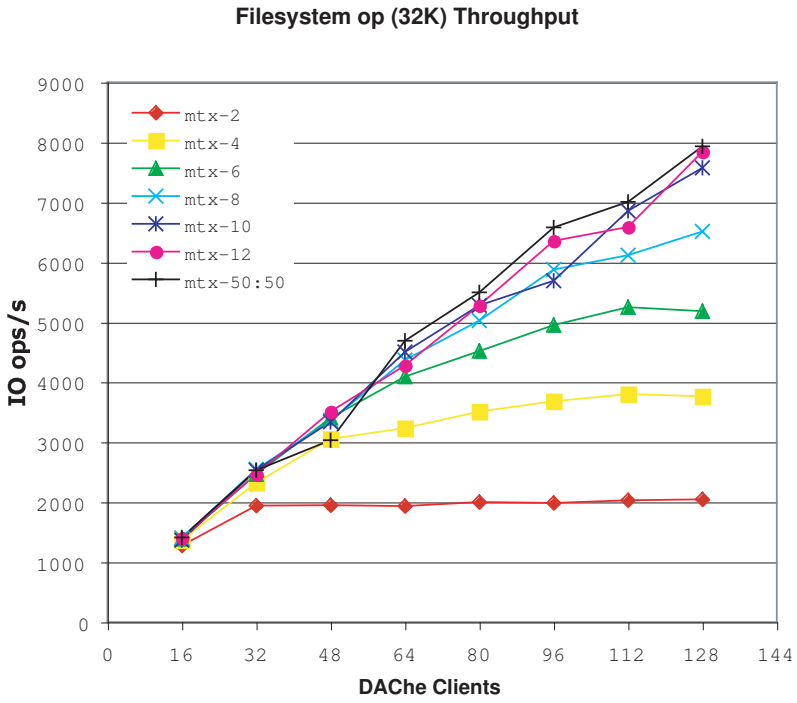

Figure 5.4. The number of clients at which throughput knees over is dependent on how many lock servers there are.

performance when more than the minimum number of mutex servers is used. From a cost efficiency perspective, one would like to stay on the outer curve using the minimum number of processes at each point. This client to mutex server ratio is highly dependent on the properties of the specific machine. A 50:50 mix where there are an equal number of clients and mutex servers allows the mutex service to scale with the application size. As expected from the growth rate of file system calls, this outer bandwidth curve is roughly a square root function. The usefulness of this will become more apparent in Section 5.3. The most important point from Figure 5.4 is the number of mutex servers determines at which number of clients performance will plateau.

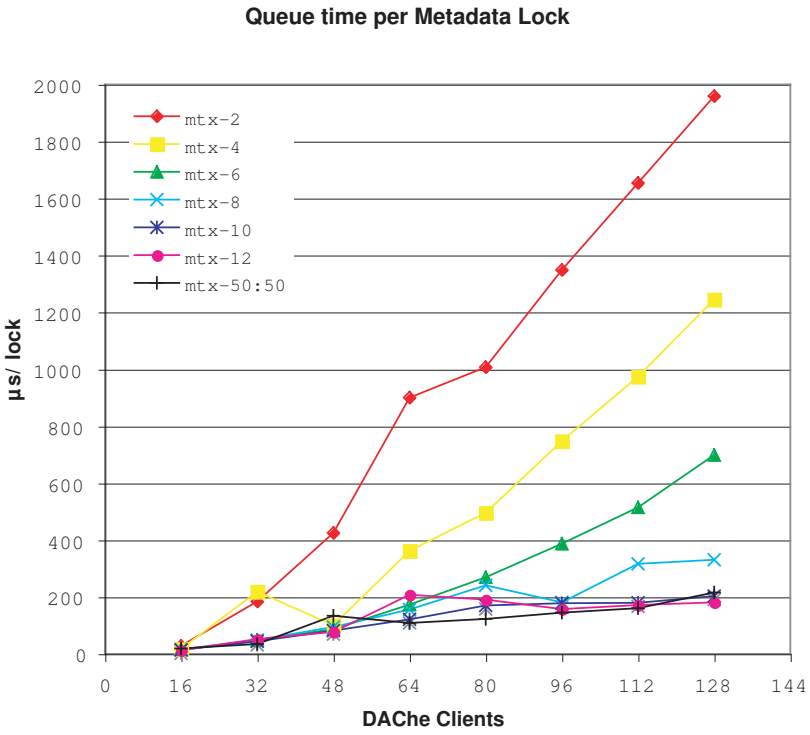

Figure 5.5. Since the number of locks grows as a second order function, increasing the number of mutex servers with the number of clients should yield a linear increase in Queue time.

The lock system's scalability is also demonstrated in Figure 5.5, which illustrates the queue time for a lock. An obvious bottleneck, increasing the number of mutex servers with the number of clients decreases the average queue time for each lock request. Increasing the number of lock servers with the number of clients keeps queue times reasonable. Because the problem size increases at second order rate, the 50:50 mix ratio of clients to servers yields the expected linear increase in queue times. The client service time includes network latency and bandwidth as well any lock contention generated by the sliding window application itself.

Figure 5.6 provides a breakdown of how much time is spent executing a subset of functions in DAChe. Table 5.2 provides a brief explanation of each function included in the timing

Abbreviation	Function
Meta Lock	Obtain access to cache metadata
Meta Unlock	Release access to cache metadata
Eof	(Un)Lock the End-of-file for writing
Get Meta	RMA retrieving cache metadata
Put Meta	RMA writing cache metadata
Rmt xfr	Data transfers to/from remote caches
Application	Application + silent functions

Table 5.2. Some internal DACche functions and their brief descriptions.

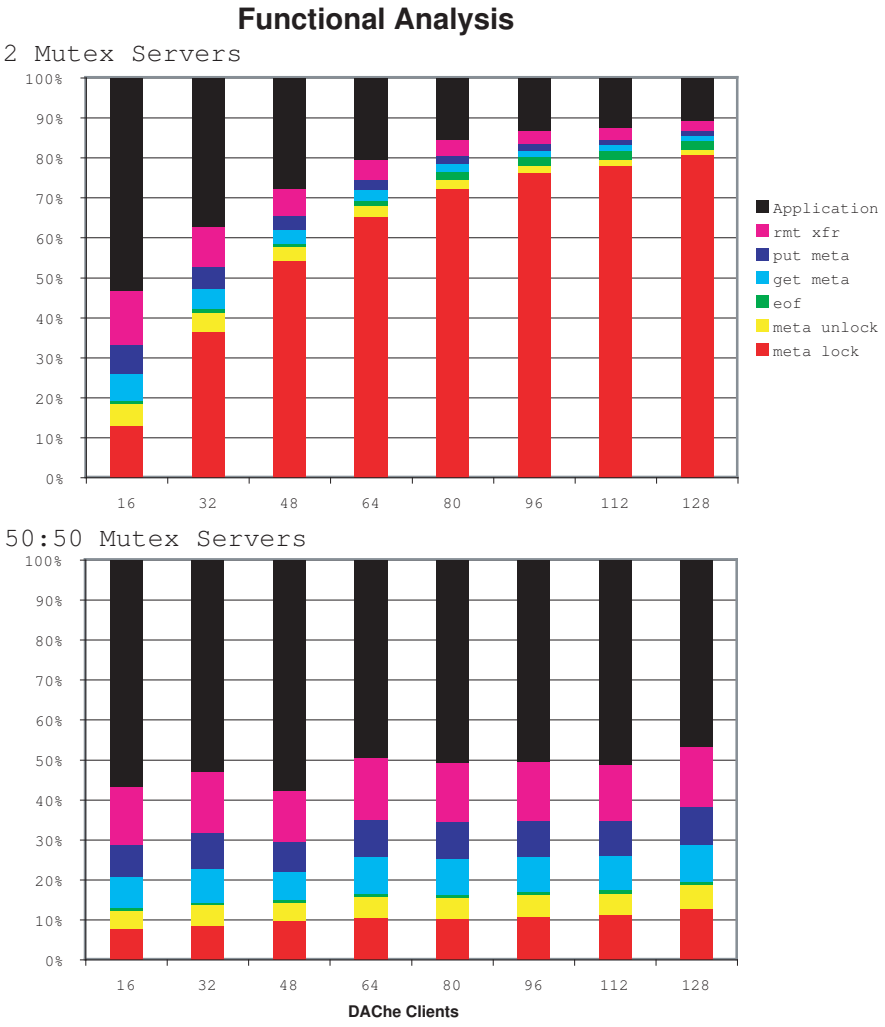

Figure 5.6. Relative amounts of time spent on individual DACche functions for 2 mutex servers and 50:50 mutex servers.

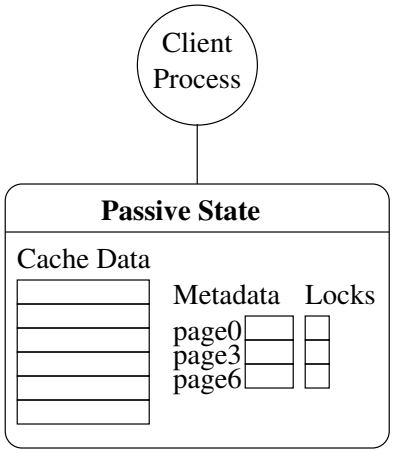

Figure 5.7. With proper support in the RMA interface, it is possible to implement locks passively on the clients.

figure. Indicated by the quickly rising meta lock times, two mutex servers is obviously not sufficient beyond 16 processes. While the proportion of time spent on locking increases as the number of processes gradually increases, it can be explained by the second order increase in locks generated by the sliding window application. Since the other functions in DACHe utilize a fairly constant proportion of the run time, and all include one-sided communication, the network is not being saturated. It is worth noting that since the sliding window application only tests I/O, its non-io related computation is negligible. The dramatic difference in relative communication costs is indicative of the need to scale mutex services with the computation size.

5.3. Conclusion

The DACHe prototype has several outstanding issues.

Using separate lock servers taken from the user's compute nodes is cumbersome and not transparent. The user can no longer treat MPI_COMM_WORLD as the global communicator in the application, because the lock servers will not be executing any application code. An

RMA-based locking solution that can be symmetrically deployed on all the user's compute nodes is necessary for usability and integration with ROMIO.

Here, RMA allows the lock responsibilities of the lock servers to be taken up by client nodes, eliminating the extra cost of separate lock processes and leveraging the performance characteristics of single-sided communication, see Figure 5.7. The algorithms involved have been explored chiefly by Mellor-Crummey *et al.*[**28**] in the context of shared memory machines. Another reason for the move to a passive RMA solution is the architectural trend of parallel computers towards running stripped down threadless operating systems on compute nodes. A challenge of moving to a completely passive lock system is implementing distributed queues in an efficient manner. A distributed waiting queue will result in additional communication costs, but these costs will hopefully be more than compensated for by performance enhancements elsewhere in the system. This passive design goal is the reason behind the lock system's simplistic design. A passive environment prevents any computation on the server side and limits the server to remote accessible state. This is expanded on in Chapter 6.

Preliminary performance results for DAChe suggest it scales well. The extremely distributed characteristics of DAChe get around a number of potential bottlenecks. The most interesting feature of DAChe is its efficient use of the increasingly common one-sided communication model.

CHAPTER 6

Passive Distributed Lock Managers

Avoiding points of globally collective communication in an application can be crucial to achieving acceptable performance. Locking is an important means of selectively synchronizing subsets of processes. Locking allows processes to proceed independently in an application when communication between processes cannot be determined in advance.

User-level locking is traditionally done with separate lock processes. In typical Linux-based clusters, this will work, but recent trends in the high-performance computing space point to a drive toward light weight compute node operating systems that often do not support threads. Additionally, users are often reluctant to give up compute nodes to dedicate to services like locking. Working within these constraints leaves Remote Memory Access (RMA) based locking solutions. Conveniently, many modern high-performance interconnect technologies like Myrinet and Infiniband achieve relatively low latencies in part by being fundamentally built on a one-sided communication model. Distributed, one-sided based lock design is particularly suitable in these environments.

One-sided locking protocols fundamentally hinge on the support of some atomic transaction where in a single indivisible operation, memory is both written and read. Swaps (fetch & set) read and write the same memory space. Atomic increment is also essentially an in-place read-modify-write. Without a similar atomic operation, one-sided locking would be an endless cycle of race conditions. The Portals API provides a swap function as its in-place

atomic read/write operation. MPI-2, however does not allow in-place atomic read/write operations, so a slightly more unusual approach is taken.

6.1. Polling and Its Variations

Polling can be trivially built on any of the in-place atomic memory operations. Basically, a value indicating the lock attempt is placed in memory, and the original value from the memory is evaluated to determine whether the lock was free. Polling consists of repeatedly checking the remote memory value. Naively polling as fast as possible will result in a great many requests, adversely affecting both aggregate network performance and likely the memory host. There are several simple techniques for addressing this.

Instead of continuously polling with the atomic read/write operation used for the initial request, network traffic generated by polling can be nearly halved by switching to just remote reads after the initial read/write lock attempt fails.

Back-off protocols for randomizing subsequent lock requests are more effective at alleviating collective pressure on the lock host and network. Basically, back-off consists of waiting some amount of time before retrying the lock. There are several options for determining the wait time:

(1) fixed wait time

(2) randomized over a fixed range

(3) randomization over linearly growing range

(4) randomization over exponentially growing range

In the first case, a simple fixed wait time is used. The obvious problem with this is that several lock requests arriving at approximately the same time will all retry at approximately the same time, making aggregate requests and network traffic very bursty. The second case

simply randomizes the wait time over some fixed range. In the third case, the range over which wait times are randomly chosen is increased linearly after each failure. In the last case, exponential back-off, the waiting range is increased exponentially.

In all polling algorithms, releasing the lock is a single push of the "free" value to the lock memory.

While polling is exceedingly simple to implement, it has a couple significant drawbacks. The first is there is no "fairness." The initial lock attempt order has no bearing on the actual order in which processes acquire the lock under contention. The second problem is that, assuming there is some degree of contention, the parameters for any of the back-off protocols must be carefully tweaked according to the behavior of the processes (lock arrival rate, how long locks are held, *etc.*). If multiple processes request a lock, the time between the lock being released and one of the processes acquiring it should be minimized. Wait times that are too long leave processes unnecessarily idle, but wait times that are too short may impact the network and the lock host.

6.2. Distributed Queues

Queuing solves both of the major problems with polling, and performance advantages and determinism has been proven [**3**]. The basic passive design of a distributed queue was originally done in the context of shared memory architectures[**28**], and is conceptually simple, though not as simple as polling. Later, the design was reevaluated in the context of RMA on Infiniband[**14**]. In the distributed queue algorithm, each waiting process itself becomes a node in the queue. Figure 6.1, illustrates the locking process, the propagation process, as well as the release process.

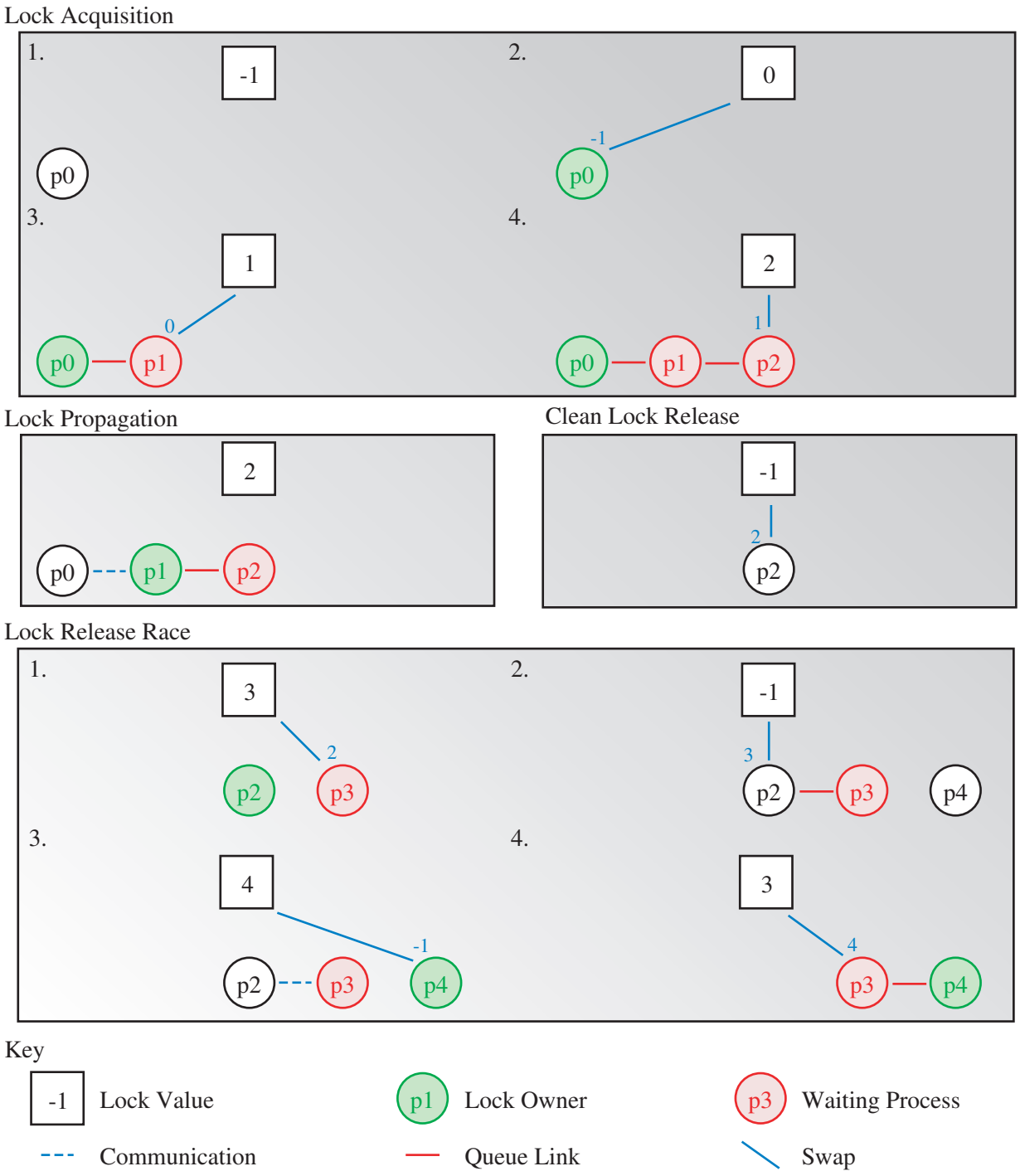

Figure 6.1. Lock acquisition is done with swaps to establish a distributed queue of waiting processes. Propagating the lock is a simple matter of traversing the queue. Completely freeing the lock needs to address a race condition because with only a swap function available, determining whether a lock should be freed or propagated cannot be done atomically with the actual act of freeing the lock.

During the lock request routine, a process swaps in its rank while simultaneously reading the current contents of the lock. If the lock value returned is in its "free" state, then the lock has been successfully acquired. If it is not free, then the value obtained is the rank of the process ahead in the queue. Once this is established, it is possible to notify the previous lock requester that there is a process behind it, thus doubly-linking the queue. If the queue is not doubly-linked, then the process releasing the lock would need to traverse the entire queue from the tail to find the next process waiting. *Wake* messages to propagate the lock are sent through regular paired MPI communication as opposed to the native lower-level API.

Releasing the lock is a little complicated because a race condition may arise depending on the available atomic operations. In the original Mellor-Crummey & Scott (MCS) design, an atomic conditional *compare & swap*, could be used to set the value of remote memory based on some condition. This makes releasing the lock very easy because no other operations can change the lock state between the condition evaluation and the swap. In the absence of a function like this, and based solely on an atomic swap, an extra step needed. While a method exists to guarantee starvation-free MCS implementation with only swap [**31**], the race occurs so infrequently that it is simpler to just deal with it somewhat less efficiently.

The process releasing the lock cannot simply put the free value into the lock. Before trying to free the lock in the remote lock space, the process needs to make sure no other processes are waiting for the lock. If the process is not notified by a subsequent lock process of its presence, the releasing process will go ahead and swap the free state into the lock and get the contents of the lock space back. The race arises from the two distinct steps of checking the lock and freeing it. If the lock space rank matches its own rank, then the lock is successfully released. If it does not, then another process believes that there is a locking

queue, even though the lock is now "officially" free at this point. Any arriving locks after the swap occurs will build a brand new queue. The processes in the queue formed behind the process that mistakenly freed the queue must be joined into the new queue. This can be accomplished by having them all resubmit their lock requests. Alternatively, the head of the disconnected queue may submit a lock request on behalf of the tail node: preserving the queue order and reducing the number of lock retries to one per race encountered regardless of how long the disconnected queue is. This race in the release algorithm means that it may not always be fair, and as long as new lock requests arrive, starvation is possible. The odds of these cases occurring are very slim, and at worst, the design breaks down into polling.

6.3. Locks Based On MPI-2 RMA

MPI-2 does not provide an in-place atomic read/write operation. Instead, MPI-2 allows users to atomically read and write separate regions of memory. Additionally, the read and write spaces may be non-contiguous (but still not overlapping). Using this API, a basic token passing scheme is designed that does not guarantee fairness, but is starvation-free and guarantees progress [**23**].

The data structure consists of a globally remote accessible array of flags, one for each process. A process may only write its own flag and may only read the others. It is worth specifically noting that a process may not read its own flag in the remote array. The flag indicates that a given process either has the lock or is waiting for it.

As illustrated in Figure 6.2 steps 1-3, the lock requests consist of a process atomically reading the flags representing all the other processes and writing (enabling) its flag in the array. If none of the other processes' flags are raised, then the lock has been successfully

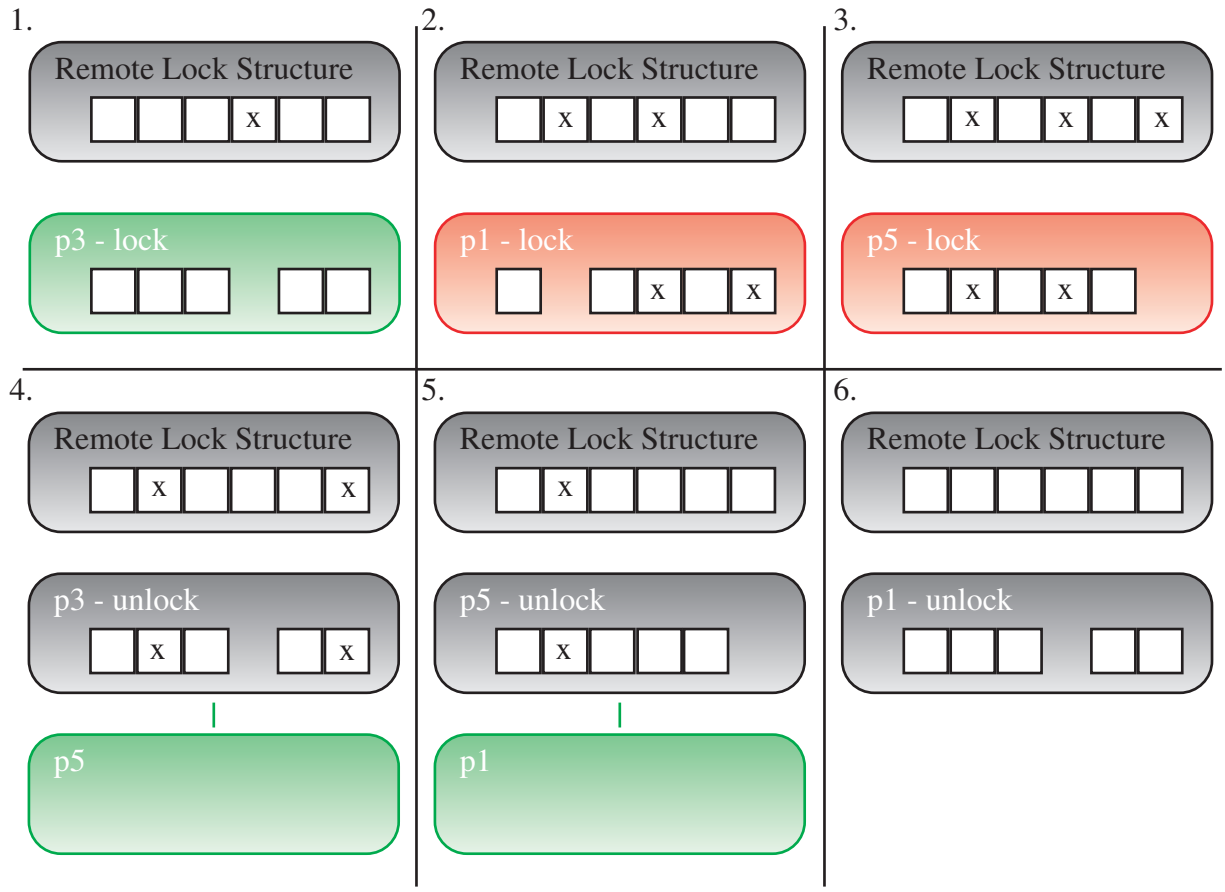

Figure 6.2. Steps 1-3 illustrate the processes 3, 1, and 5 attempting to acquire the lock in that order. Steps 4-6 illustrate the release/propagation of the lock in circularly ascending rank order.

acquired. Otherwise, the process must wait for a notification from someone that passes the lock along.

To release a lock, the process atomically reads all the other processes' flags and unsets its own. If no other processes are waiting for the lock, there is nothing more to do, as in Figure 6.2 step 6. If one or more processes are waiting for the lock, as in Figure 6.2 steps 4 and 5, a *wake* message will be sent to the next higher process in the array (in circularly ascending order in the array).

There are two advantages to this locking scheme. It is portable across any machine with a working MPI-2 implementation, and because of MPI's datatypes, the design can be augmented to piggy-back the actual data being locked along with the lock traffic itself. Two drawbacks are a lack of fairness and a space requirement that grows with the number of lock clients.

6.4. Performance Comparisons

Performance evaluation for the different passive-model locking is on the Cray XT-3/4 at Oakridge National Laboratories. Compute nodes consist of a 2.6 GHz dual-core AMD Opteron with 4 GB of memory. The network topology is a 3-dimensional torus with I/O nodes on the ends. Nodes use Cray's proprietary Seastar interconnect to achieve a theoretical peak bandwidth of 7.6 GB/s.

Before comparing relative lock performance, it is necessary to first establish the baseline performance difference between MPI-2 RMA and the lower-level Portals API. Figure 6.3 shows bandwidths using both the Portals and MPI one-sided communication APIs on the Cray XT-3. The points designate the averages and lines indicate the range of values over ten runs. The throughput curve for the Portals RMA is virtually the same as the performance of Cray's SHMEM interface over Portals[**49**]. For high-performance computing, communication bandwidth is important, but latency is also vital, more so than in other computing domains. The tightly coupled nature of high-performance applications drives the need for low-latency communication. For locking, where the data transferred is very small - on the order of bytes or N integers, latency is especially important. Figure 6.4 compares latency of Portals RMA and MPI-2 RMA.

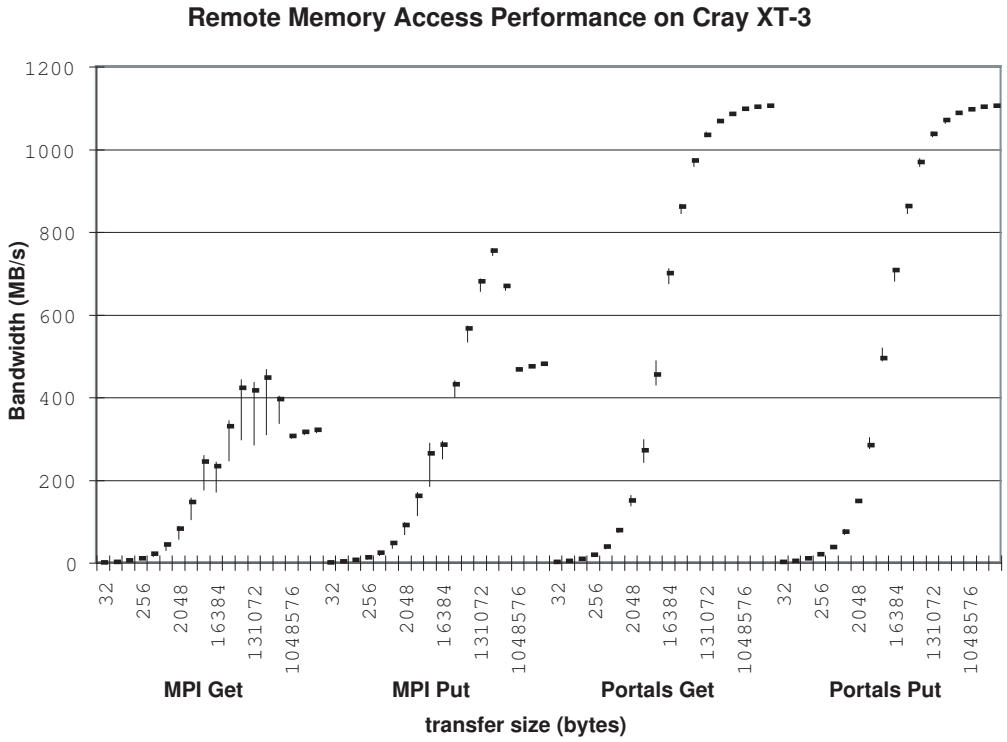

Figure 6.3. Predictably, the lower-level Portals library is more efficient, if less user-friendly, than MPI.

The locking data structures used for the Portals based locks are simply single integers. For MPI-2 based locks, the remote accessible array is integer based, though it could technically be byte based. As long as traffic remains in such small ranges, MPI-2 incurs very little penalty with respect to bandwidth for data movement relative to Portals. The more significant difference between the two interfaces at those data sizes is latency. Next, in Figure 6.5, aggregate lock throughput is analyzed in two contexts, one with full contention, and the other with no contention. In the full contention case, all processes try to obtain a single shared lock and release it after a set "computation" time; in this case 200 microseconds. In the contention free case, each process accesses a separate lock. Under full contention, lock

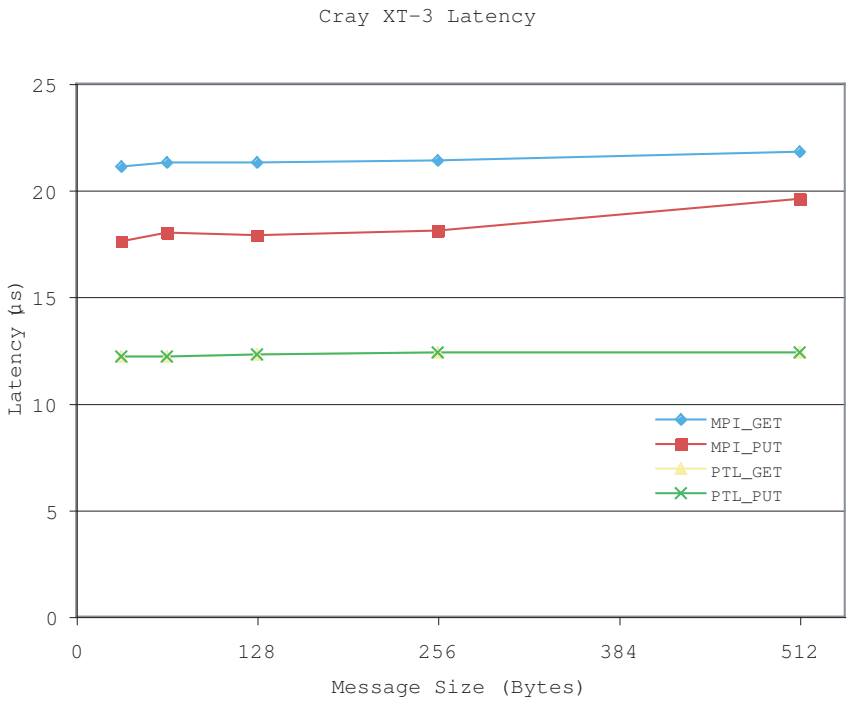

Figure 6.4. Given the small data amounts involved in locking latency is very important to lock performance. Portals Get and Put latencies are identical.

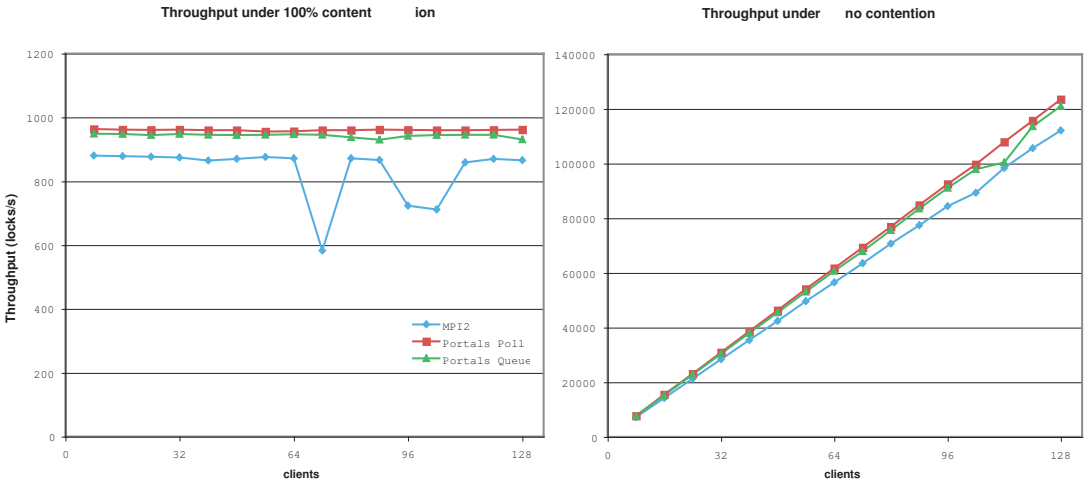

Figure 6.5. There is a definite performance advantage to using Portals over the MPI-2 when a native Portals implementation is available.

throughput stays relatively level with a slight downward trend in performance as lock clients are added. The Portals-based locking algorithms yield about a 10% improvement over the MPI-2 based locking algorithm. Under no contention, all algorithms scale linearly, but the MPI-2 based algorithm scales less quickly. These results were obtained by averaging the remaining values after removing the highest and lowest value out of five runs. Locks are hosted one per node on separate nodes that do not request locks. Hosting the locks on the client processes should make little difference. Without any contention, every lock client will successfully obtain its lock on the first try, so if they were to host the locks, themselves the interference between lock-hosting responsibilities and lock requests would be minimal. With total lock contention, only one process would have both lock-hosting responsibilities and generate lock requests, and the overall impact of these interactions is minimal. Furthermore, hosting the locks on separate nodes ensures that all lock requests travel over the network.

Another performance metric evaluated is lock propagation time. Essentially, this is the time it takes for a lock, once released, to propagate through all the waiting processes. This is similar to the full contention throughput experiment, but eliminates the inherent race between initial lock requests. One process is allowed to obtain the lock, and then it waits for all the other processes to submit an initial lock request before releasing the lock. Figure 6.6 shows that both the MPI-2 based locks and the Portals Distributed Queue locks pass locks along far more efficiently. While the first two methods appear to scale linearly, the polling methods grow quadratically. Tweaking the randomized waiting range affects polling performance, but one needs to have intimate knowledge of arrival rates and the nature of the computation time.

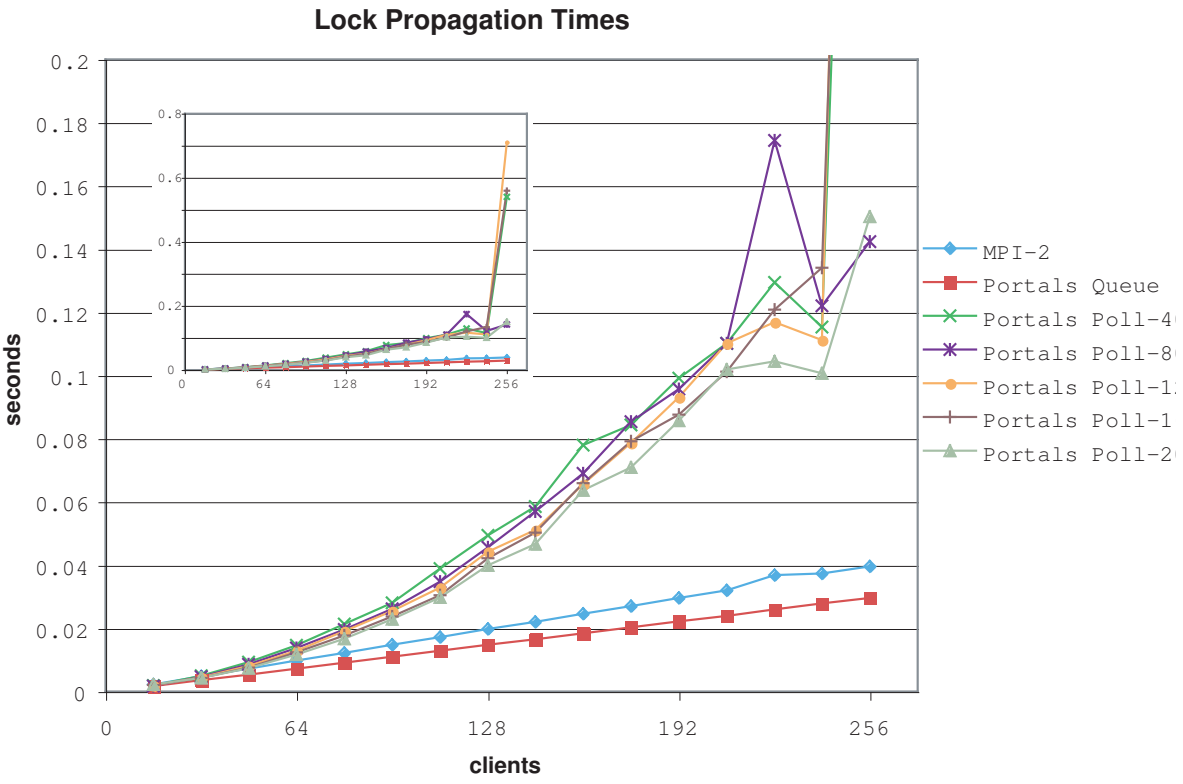

Figure 6.6. The MPI-2 based locking and Portals Distributed Queue locks clearly outperform Portals Polling both nominally and in run-time complexity. Several sleep ranges are used for polling: 40, 80, 120, 160, and 200 microseconds. The simulated compute time, or time the locks are held for, is 100 microseconds.

6.5. Conclusion

The availability of MPI-2's RMA interface or any other one-sided communication interface allows the implementation of a passive mutual exclusion algorithm. With the wide use of MPI-2, this is an ideal platform for deploying portable user-space passive locks. This fact, plus its comparable performance to using lower-level RMA APIs make MPI-2 a good choice. The lock tests used here really stress test the upper bound in performance, and while there was a certain amount of overhead associated with lock implementation in MPI-2, typical loads will narrow the realized performance differences.

Presumably, a comparison of MPI-2 lock performance versus other polling or distributed queues algorithms using other RMA APIs such as Infiniband should yield similar results. This is partially dependent on how efficiently the MPI-2 RMA interface is implemented for that particular network infrastructure. A direct comparison of MPI-2 base locks and Portals based locks can really only be made on the Cray XT3/4 architecture.

CHAPTER 7

DAChe Integration with ROMIO

With passive distributed lock management available, DAChe is readily integrated into ROMIO. By integrating DAChe into ROMIO, many of the already written MPI-IO codes may quickly take advantage of DAChe. The key design considerations in the integration effort are familiar: compartmentalization, transparency, portability, and performance.

7.1. ROMIO Design Overview

ROMIO is architected in such a way as to allow file system specific optimizations to be made. MPI-IO calls are passed to the Abstract Device I/O (ADIO) layer where the resident file system is determined, setting a function pointer to the appropriate set of ADIOI (Internal) file system functions. ADIOI functions are the internal ADIO functions not externally

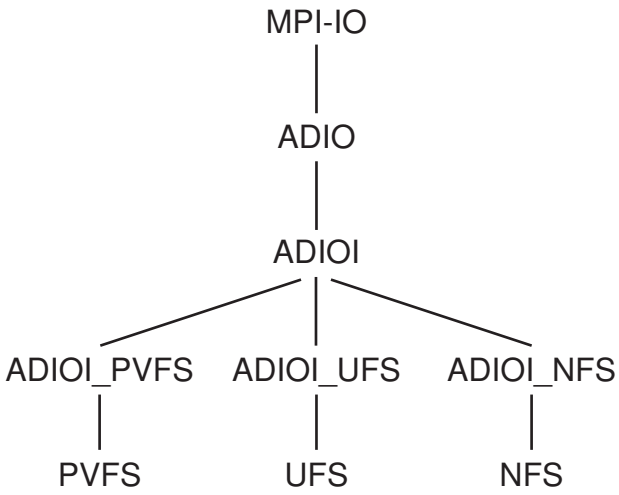

Figure 7.1. File system specific optimizations are implemented in the ADIOI_xxx functions of ROMIO.

exposed. The file system specific optimizations as well as optimizations further discussed in Section 7.5 are implemented here in the ADIOI functions. Figure 7.1 illustrates the basic ROMIO design.

File system resolution is handled during `MPI_File_open`. This will call the function `ADIO_ResolveFileType` to determine what file system the file resides on and set the file systems function pointer to the appropriate set of file system ADIOI functions. `ADIO_ResolveFileType` is found in `romio/adio/common/ad_fstype.c`, and calls either `ADIO_FileSysType_fncall` or `ADIO_FileSysType_prefix`. Both functions actually do the file system determination. The former uses system-dependent function calls, and the latter uses a user provided prefix (*eg.* `pvfs:`). Eventually, `MPI_File_open` passes the function pointer to `ADIO_open`, which in turn stores the function pointer in the ADIO file descriptor.

After an application makes a regular MPI-IO call, it first goes to the `romio/mpi-io` functions, which implement the MPI-IO interface. The MPI-IO functions then call an ADIO function that maps directly to file system specific `ADIOI_xxx` function.

This design also makes file system agnostic optimizations slightly more challenging to implement since changes may need to be propagated to each file system specific ADIOI implementation.

7.2. New DACche Features

In readying DACche for integration with ROMIO, there are several changes and additional features from the original DACche prototype. First and foremost is a new mutex system that uses RMA-based locks. For more flexible and dynamic load-balancing, a page-migration system is implemented that allows processes that frequently access a remote cache page to try to "steal" the page and make it local.

Also in the newer DACche is the ability to use either a standard POSIX I/O interface or MPI-IO to access files. Not only is this useful for testing, but it also addresses the issue of file system portability for DACche. The flush function for DACche is written to take advantage of the MPI-IO non-contiguous interface so each process may flush data to file in a single MPI-IO call. Since flushes are collective, the flush can also be done using MPI's collective I/O.

7.3. Hiding DACche

Allowing applications to transparently leverage the benefits of DACche is important to the overall utility of this research. The impact on current and future application development should be minimal. Achieving transparency is one of the main motivations for integrating DACche into ROMIO. By integrating DACche into ROMIO, taking advantage of DACche from the perspective of applications should be simple. The only potential change is the addition of and MPI hint to tell ROMIO whether to use DACche or not. As this is no more than many other optimizations in ROMIO require, this should be acceptable. The decision of where in ROMIO to put DACche really has no bearing on transparency.

7.4. DACche and the ROMIO Architecture

In order to maximize the utility of DACche and keeping in the spirit of ROMIO, DACche is integrated into ROMIO as portably as possible. Hopefully, if there are more people that can use DACche on their systems, more will. ROMIO intentionally has a solid infrastructure for maintaining portability. As noted in Section 7.1 however, the same features that make implementing file system specific optimizations easy, can make implementing universal optimizations harder.

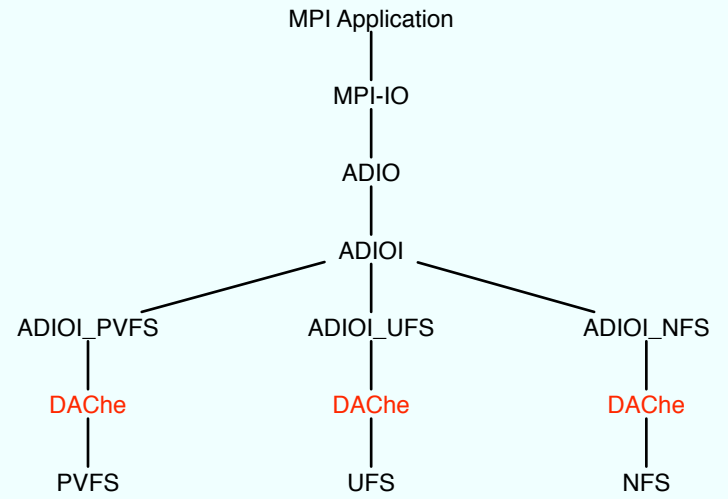

Figure 7.2. Adding DAChe below the ADIOI layer entails some replication of effort.

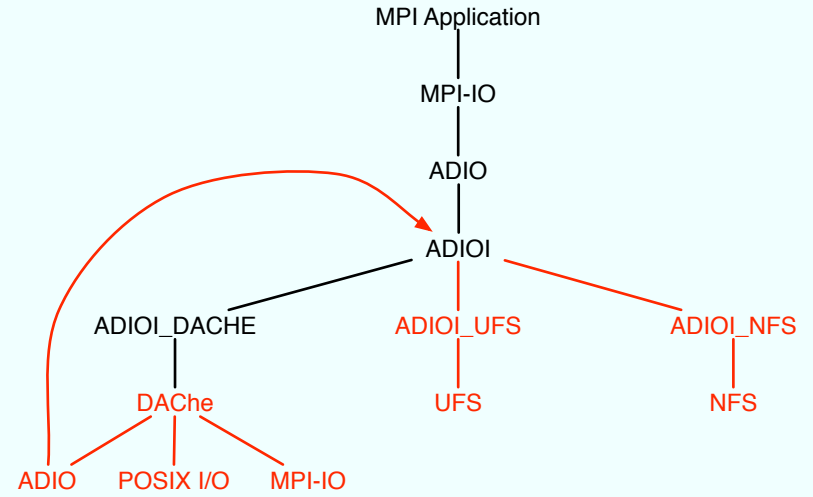

Figure 7.3. In this option, DAChe acts as a sort of Virtual ADIO device and calls ADIOI-level functions to access the file system appropriate ADIO driver.

From Figures 7.2, 7.3, and 7.4, the general effects on the ROMIO code base are apparent. In putting DAChe beneath the ADIOI file system specific functions, much of the work is replicated and affects each of the ADIOI implementations. The advantage at this level is the ability to incorporate file system specific DAChe optimizations. Implementing DAChe as an

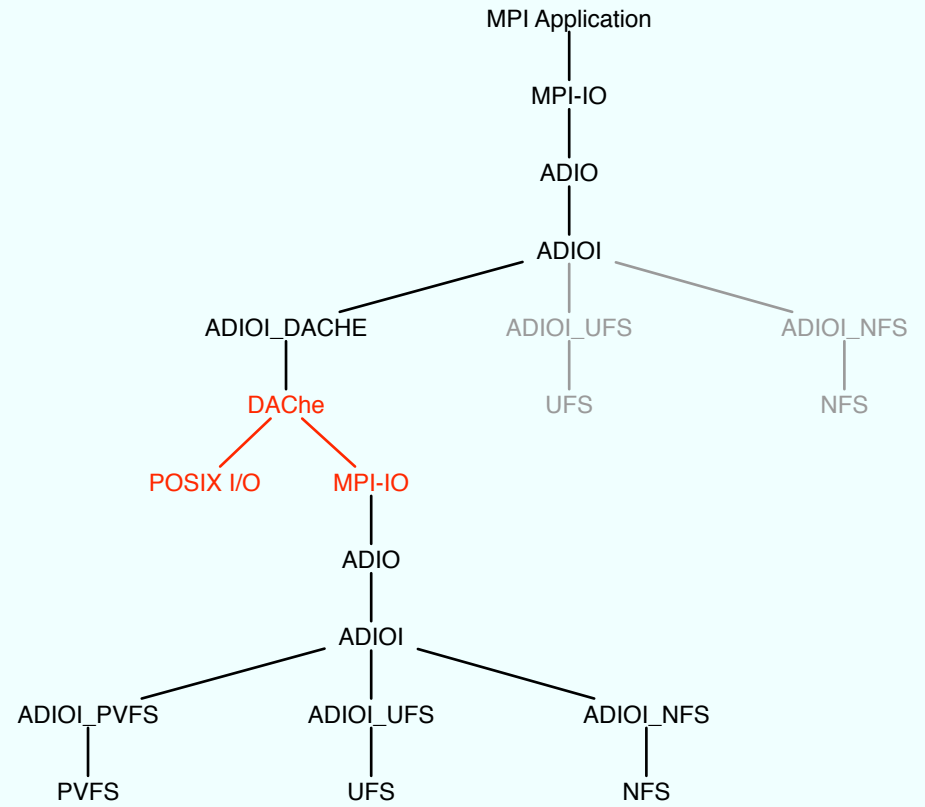

Figure 7.4. Again, DAChe acts as a Virtual ADIO device, but in this case DAChe itself does I/O using the MPI-IO interface itself on a new MPI file descriptor. This is the current implementation, and the easiest to test since DAChe alone can be tested easily using either POSIX I/O or MPI-IO for I/O.

independent ADIO driver compartmentalizes changes to the driver itself and some upstream code, but does not affect the other ADIO drivers. As an ADIO driver, there is a further choice of re-entering the MPI-IO stack using a new file descriptor, or using the already allocated ADIO file descriptor internally. From a testing and development perspective, re-entering the MPI-IO stack from the top is cleaner, but from an efficiency perspective, using the existing ADIO file descriptor avoids all the initialization and memory overhead associated with opening a new MPI file descriptor. In all these integration methods, the user need not modify application code except to optionally control DAChe through MPI hints.

DACche is largely file system independent. The one exception to this is in dealing with file system client-side caches. There is simply too much variation between file systems in this respect, and not all file systems even have caches. Fortunately, ADIO provides `ADIO_Flush` for syncing written data in a client cache back to disk. Methods for forcing file system reads to go all the way to disk rather than just cache (assuming there is a cache) are far less uniform across systems. On Linux machines, this can be done with `O_DIRECT` I/O calls. This may introduce memory alignment issues, but is definitely manageable. The problem with bypassing file system caches can be solved with the introduction of a new ADIOI function for that express purpose.

7.5. Interactions with Existing Optimizations

There are several I/O optimizations currently in use in ROMIO, the basic premises of which are reducing the number I/O calls to the file system. Since DACche can do I/O through MPI-IO as in Figure 7.3, DACche is able to leverage any of the optimizations an MPI library provides, including its powerful non-contiguous interfaces.

7.5.1. Data Sieving

The data sieving technique reads large contiguous pieces of file into a temporary buffer in order to satisfy multiple I/O requests at once. Calculating the correct DACche pages above data sieving would require some changes to generate noncontiguous requests for multiple DACche pages. Substituting simple contiguous DACche calls below the data sieving optimization would be much easier. Since DACche is page based, and performs I/O only on page

multiples, the reduction in I/O requests achieved by data sieving may be somewhat attenuated by DACHe underneath it. This is dependent on the size of the data sieve buffer and DACHe pages, so they may have to be tuned together.

7.5.2. Non-contiguous Interfaces

The list I/O and datatype I/O interfaces provide a means of succinctly describing multiple noncontiguous regions in a file. Since list I/O and datatype I/O are actually replacement calls for POSIX file I/O, DACHe can still take advantage of these interfaces when available. In order to take full advantage of these noncontiguous interfaces though, there needs to be some serious changes, or at least additions, in DACHe to generate I/O requests for multiple DACHe pages.

7.5.3. Two phase I/O

One of the biggest benefits of DACHe is its ability to maintain cache coherency for applications with a mix of independent and collective I/O without using locks in the file system. For example, an application that writes large amounts of data collectively, but also generates intermittent updates to metadata in the file independently. The only potential drawback is that two phase I/O with DACHe masks the user's access locality since one process may access a region of a file on behalf of another process that actually needs the data. Two-phase may lessen the likelihood that a process will cache data it will need itself (from the application's perspective).

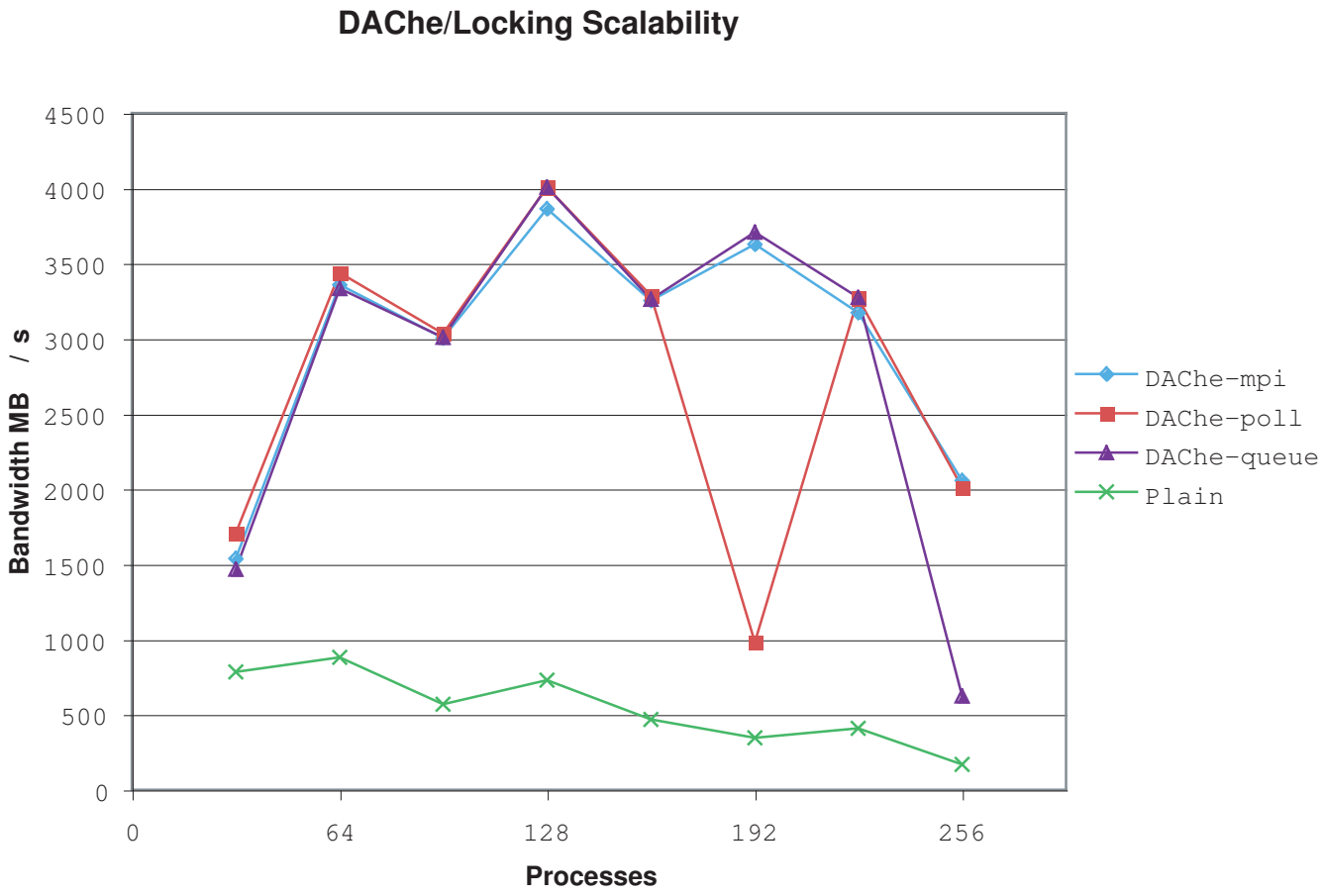

Figure 7.5. There is no real performance difference between DACHe and the different combinations of MPI-2, Portals-based Polling, and Portals-base Queuing. The application does perform significantly better with DACHe than without.

7.6. Performance Evaluation

These DACHe experiments were run on the Cray XT3/4 at Oakridge National Laboratories. The detailed machine configuration can be found in Chapter 6. I/O is done on a 72 host shared Lustre Partition.

Nodes	File Realm size (MB)
32	64
64	32
96	21.3
128	16
160	12.8
192	10.6
224	9.1
256	8

Table 7.1. Approximate file realm sizes as the number of clients increases.

7.6.1. Write-only Strided Test

The initial test of the DACche library in conjunction with MPI-IO is done using the Persistent File Realm benchmark described in Chapter 4. The DACche results were obtained by dropping the best and worst figures out of five runs and averaging. The plain collective I/O results were obtained by dropping the best and worst figures out of fifteen (this was unintentional) runs and averaging. DACche is configured to use 1 MB page sizes, and 32 MB of cache on each node. The benchmark writes an aggregate of 2 GB over the course of 16 collective writes. The element size (each contiguous piece of data) is 64 Bytes, there are 1024 elements per data point, and 2048 data points. Table 7.1 enumerates the approximate file realm sizes for the corresponding number of nodes. For 32 nodes, the file realm size on each process is about 64 MB. This is more than the cache size per node; so processes start evicting pages around the 8th collective write call. From Figure 7.5, it is apparent that the underlying locking method used does not make a big difference in the overall performance trends. The difference in performance of the various locking systems is overshadowed by other factors like I/O time and data transfer times. Looking back at Figure 6.3, there would likely be more of a performance difference if DACche were compared using MPI-2 RMA versus

Portals RMA for all its one-sided communication (remote cache hits). The across-the-board dips in performance at 64 and 160 clients can be explained by alignment.

7.7. Conclusion

For better organization and maintenance purposes, DACheis integrated into ROMIO through an ADIO driver. Keeping DACheis somewhat separate from the rest of the existing ADIO also reduces replicated code. In this integration route, DACheleverages the portability of ROMIO over many file systems, and allows many existing MPI-IO applications to easily enable and try DACheis.

While convenient, the existing method of re-entering ADIO from DACheis through the top of the MPI-IO stack is not the most efficient way of doing it. From purely an efficiency perspective, a driver in DACheis that will let DACheis call `ADIOI_xxx` functions directly would be better. This would cut initialization and memory costs.

The least recently used cache eviction policy is commonly accepted as the best general-purpose policy. In the current implementation, the LRU queue is stored locally, and is only affected by local accesses to cache pages. In addition to evaluation of the basic eviction performance of DACheis , further research is planned in how incorporating remote accesses in the eviction policy may affect overall cache performance. Also of interest is IBM's Adaptive Replacement Cache (ARC)[**27**] scheme. With relatively low overhead, ARC incorporates both recency and frequency in its replacement strategy. Like the page migration technique, the ARC scheme should be able to help accommodate applications that have changing I/O access patterns over time, but from a policy point.

While page migration is implemented, it has not been formally evaluated yet. Types of applications it may suit are AMR applications as well as any other application that does access files in several distinct patterns.

CHAPTER 8

Conclusions

The MPI-IO layer is ripe with opportunities for addressing I/O performance issues. Several optimizations demonstrate this, with caching being of particular promise.

High-performance caching is first addressed in the context of collective I/O using Persistent File Realms to reorganize I/O accesses to ensure coherent access to a file system client-side cache. The explicit and active coordination collective I/O allows is ideal for managing coherency. The file system no longer needs to worry about maintaining coherency with expensive control mechanisms since responsibility is moved up a layer where more information on a collection of accesses is available. Cache sensitive applications stand to make a substantial performance gain. PFRs allow users to safely leverage an incoherent client-side file system cache without using locks. PFRs also present I/O servers with better I/O locality.

Work on PFRs helped spur the development of a new collective I/O implementation for ROMIO. This implementation focuses on maintainability and flexibility while attempting to deliver comparable performance. Easier maintenance is achieved by a significant amount of code reuse, and flexibility is improved by using MPI datatypes to describe arbitrary file realms. The additional flexibility can be used to hone run time optimizations for a variety of access patterns as well as architectures. This new framework makes new experiments with PFRs, file realm alignment, and data sieving within collective I/O quick to set up. The new implementation will also handle custom or irregular file realm assignments tailored to the specific needs of applications, machine architectures, and system environments.

Extending cache coherence management to independent I/O operations in MPI requires setting up a file cache within MPI. Without explicit coordination beyond open and close, processes need to work independent of each other without interfering with each other. Using remote memory access, supported by many modern interconnects as well as MPI-2, DACche distributes cache data and maintains coherence passively and without the need for threads. Applications may be able to greatly reduce the number of I/O requests that go over the network to I/O servers, thereby improving performance and alleviating stress on I/O servers. Along these lines, the effect may be thought of as artificial route dispersion. Together with ROMIO, DACche has the potential to be run under a great number of MPI-IO applications. There are many combinations of DACche and other optimizations already in ROMIO that can be evaluated in the future.

References

[1] George Almasi, Ralph Bellofatto, Jose Brunheroto, Calin Cascaval, Jose G. Castanos, Luis Ceze, Paul Crumley, C. Christopher Erway, Joseph Gagliano, and et al. An overview of the Blue Gene/L system software organization. In *Proceedings of the Euro-Par '03 Conference on Parallel and Distributed Computing*, 2003.

[2] An introduction to GPFS 1.2. `http://www-1.ibm.com/servers/eserver /clusters/software/gpfs.html`, December 1998.

[3] Thomas Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. In *IEEE Transactions on Parallel and Distributed Systems*, volume 1, pages 6–16, 1990.

[4] Rajesh Bordawekar, Juan Miguel del Rosario, and Alok Choudhary. Design and evaluation of primitives for parallel I/O. In *Proceedings of Supercomputing '93*, pages 452–461, Portland, OR, 1993. IEEE Computer Society Press.

[5] Ron Brightwell, Bill Lawry, Arthur Maccabe, and Rolf Riesen. Portals 3.0: Protocol building blocks for low overhead communication. In *International Journal of High Performance Computing Applications*, volume 17, pages 7–19, 2003.

[6] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, October 2000. USENIX Association.

[7] Avery Ching, Alok Choudhary, Kenin Coloma, Wei Keng Liao, Robert Ross, and William Gropp. Noncontiguous access through MPI-IO. In *Proceedings of the IEEE/ACM International Symposium on Cluster Computing and the Grid*, Tokyo, Japan, May 2003. IEEE Computer Society Press.

[8] Avery Ching, Alok Choudhary, Wei Keng Liao, Robert Ross, and William Gropp. Noncontiguous I/O through PVFS. In *Proceedings of the IEEE International Conference on Cluster Computing*, Chicago, IL, September 2002. IEEE Computer Society Press.

[9] Avery Ching, Alok Choudhary, Wei Keng Liao, Robert Ross, and William Gropp. Efficient structured access in parallel file systems. In *Proceedings of the IEEE International Conference on Cluster Computing*, Hong Kong, December 2003. IEEE Computer Society Press.

[10] Avery Ching, Alok Choudhary, Wei Keng Liao, Robert Ross, and William Gropp. Evaluating structured I/O methods for parallel file systems. In *International Journal of High Performance Computing and Networking*, volume 2, pages 133–145, 2004.

[11] Avery Ching, Alok Choudhary, Wei Keng Liao, Lee Ward, and Neil Pundit. Evaluating I/O characteristics and methods for storing structured scientific data. In *Proceedings of the International Parallel & Distributed Processing Symposium*, Rhodes Island, Greece, April 2006. IEEE Computer Society Press.

[12] CPLANT: A commodity-based, large-scale computing resource. `http://www.cs.sandia.gov/cplant`.

[13] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *Proceedings of the IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 56–70, Newport Beach, CA, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 31–38.

[14] Ananth Devulapalli and Pete Wyckoff. Distributed queue-based locking using advanced network features. In *Proceedings of International Conference on Parallel Processing*, 2005.

[15] Bruce Fryxell, Kevin Olson, Paul Ricker, Frank Timmes, Michael Zingale, Donald Lamb, Peter MacNeice, Robert Rosner, and Henry Tufo. FLASH: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. *Astrophysical Journal Supplement*, 131:273, 2000.

[16] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.

[17] HDF5 home page. `http://hdf.ncsa.uiuc.edu/HDF5/`.

[18] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, 1988.

[19] IEEE/ANSI Std. 1003.1. Portable operating system interface (POSIX)–part 1: System application program interface (API) [C language], 1996 edition.

[20] Florin Isaila, Guido Malpohl, Vlad Olaru, Gabor Szeder, and Walter Tichy. Integrating Collective I/O and Cooperative Caching Into the "Clusterfile" Parallel File System. In *Proceedings of the 18th Annual International Conference on Supercomputing*, pages 58–67, Sain-Malo, France, July 2004. ACM Press.

[21] David Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74. USENIX Association, November 1994. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994.

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

[23] Robert Latham, William Gropp, Robert Ross, Rajeev Thakur, and Brian Toonen. Implementing MPI-IO atomic mode without file system support. In *Proceedings of the IEEE Conference on Cluster Computing Conference*, Boston, MA, September 2005. IEEE Computer Society Press.

[24] Wei Keng Liao, Alok Choudhary, Kenin Coloma, George K. Thiruvathakal, Lee Ward, Eric Russell, and Neil Pundit. Scalable implementations of MPI atomicity for concurrent overlapping I/O. In *Proceedings of the International Conference of Parallel Processing*, October 2003.

[25] Lustre. `http://www.lustre.org`.

[26] Xiaosong Ma, Marianne Winslett, Jonghyun Lee, and Shengke Yu. Faster collective output through active buffering. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, Fort Lauderdale, FL, April 2002. IEEE Computer Society Press.

[27] Nimrod Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proc. 2nd USENIX Conference on File and Storage Technologies (FAST 03), San Franciso, CA*, 2003.

[28] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. Technical Report TR 342, Computer Science Department, University of Rochester, April 1990. Also COMP TR90-114, Center for Research on Parallel Computation, Rice University, May 1990. Revised version published in *ACM Transactions on Computing Systems, February 1991.*

[29] David Nagle, Denis Serenyi, and Abbie Matthews. The Panasas ActiveScale storage cluster - delivering scalable high bandwidth storage. In *Proceedings of the 2004 ACM/IEEE Supercomputing Conference*, November 2004.

[30] NAS application I/O (BTIO) benchmark. `http://parallel.nas.nasa.gov/MPI-IO/btio`.

[31] Jonathan Nash. A scalable and starvation-free concurrent locking mechanism. *Concurrency: Practice and Experience*, 11(13):823–833, nov 1999.

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

[33] Ron A. Oldfield, Arthur B. Maccabe, Sarala Arunagiri, Todd Kordenbrock, Rolf Riesen, Lee Ward, and Patrick Widener. Lightweight i/o for scientific applications, 2006.

[34] Ron A. Oldfield, Patrick Widener, Arthur B. Maccabe, Lee Ward, and Todd Kordenbrock. Efficient data-movement for lightweight i/o, 2006.

[35] Panasas. `http://www.panasas.com`.

[36] Jean-Pierre Prost, Richard Treumann, Richard Hedges, Bin Jia, and Alice Koniges. MPI-IO/GPFS, an Optimized Implementation of MPI-IO on top of GPFS. In *Proceedings of Supercomputing*, Denver, CO, November 2001. ACM Press.

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

[38] Russ Rew and Glenn Davis. The unidata netcdf: Software for scientific data access. In *Proceedings of the 6th International Conference on Interactive Information and Processing Systems for Meteorology, Oceanography and Hydrology*, Anaheim, CA, February 1990. American Meteorology Society.

[39] ROMIO: A high-performance, portable MPI-IO implementation. `http://www.mcs.anl.gov/romio`.

[40] Mahadev Satyanarayanan. Scalable, secure, and highly available distributed file access. *Computer*, 23(5):9–18, 20–21, 1990.

[41] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the Conference on File and Storage Technologies*, Monterey, CA, January 2002.

[42] The BlueGene/L Team. An overview of the bluegene/l supercomputer. In *Proceedings of ACM Supercomputing Conference*, 2002.

[43] Rajeev Thakur and Alok Choudhary. An extended two-phase method for accessing sections of out-of-core arrays. *Scientific Programming*, 5(4):301–317, Winter 1996.

[44] Rajeev Thakur, Alok Choudhary, Rajesh Bordawekar, Sachin More, and Sivaramakrishna Kuditipudi. Passion: Optimized I/O for parallel applications. *IEEE Computer*, 29(6):70–78, June 1996.

[45] Rajeev Thakur, William Gropp, and Ewing Lusk. An abstract-device interface for implementing portable parallel-I/O interfaces. In *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 180–187, October 1996.

[46] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189, Annapolis, MD, February 1999. IEEE Computer Society Press.

[47] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 23–32, Atlanta, GA, May 1999. ACM Press.

[48] The parallel virtual file system 2 (PVFS2). `http://www.pvfs.org/pvfs2/`.

[49] Vinod Tipparaju, Andriy Kot, Jarek Nieplocha, Monika ten Bruggencate, and Nikos Chrisochoides. Evaluation of remote memory access communication on the cray XT3. In *IPDPS*, pages 1–7. IEEE, 2007.

[50] Jesper Larsson Traff, Rolf Hempel, Hubert Ritzdorf, and Falk Zimmermann. Flattening on the fly: Efficient handling of MPI derived datatypes. In *PVM/MPI 1999*, pages 109–116, 1999.

[51] Joachim Worringen, Jesper Larson Traff, and Hubert Ritzdorf. Improving generic non-contiguous file access for MPI-IO. In *Proceedings of the 10th EuroPVM/MPI Conference*, September 2003.

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1]The macros used in formatting this dissertation are based on those written by Miguel A. Lerma, (Mathematics, Northwestern University) which have been further modified by Debjit Sinha (EECS, Northwestern University) to accommodate electronic dissertation formatting guidelines.