Scalable High-level Caching for Parallel I/O

Kenin Coloma Alok Choudhary Wei-keng Liao
Center for Parallel and Distributed Computing
Northwestern University
{kcoloma, choudhar, wkliao}@ece.northwestern.edu

Lee Ward Eric Russell Neil Pundit
Scalable Computing Systems Department
Sandia National Laboratories
{lee, edrusse, pundit}@sandia.gov

Abstract

In order for I/O systems to achieve high performance in a parallel environment, they must either sacrifice client-side file caching, or keep caching and deal with complex coherency issues. The most 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.

The high-level abstraction layer of MPI allows us to 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 request. We can effectively exploit the benefits of a file system's client-side cache while minimizing its management costs.

1 Introduction

Caching is a good way of hiding the inherent mechanical limitations of disks, latencies, and bottlenecks, but when hurled into a parallel environment, client-side file caching brings its own rather complex problems. Presently, high performance I/O systems either completely avoid implementing client-side caches, or use locking systems to ensure correct file data is always manipulated. While not unique to parallel environments, client-side file cache coherency is a much more complex problem than in distributed and independent environments. In parallel environments, the performance boost that client-side file caches provide is more critical and concurrent file access is not unusual.

We investigate a new means of ensuring safe access to client-side file caches in parallel environments where multiple clients may be access the same file at the same time. By using the high level information that an API like MPI can provide, we can more efficiently manage cache coherency and hence lower the costs of caching. At the same time, all this should be possible with minimal user interaction or code modification.

In section 2 and 3 we describe the benefits and challenges of caching and the relevant I/O semantics. Then we will selectively describe the ROMIO implementation of MPI-IO as it pertains to the file consistency problem in section 4. In section 5 we describe a couple of simple intuitive solutions and move on to our idea of persistent file domains in section 6. In section 7 we present performance results gathered on ASCI Cplant at Sandia National Laboratory, and in section 8 we offer our conclusions and touch on areas for future research.

2 Cache Coherency

The basic cache coherency problem is ensuring globally accessible data is up-to-date when used. If there were no cache, data would always be up-to-date, but caching generally increases performance. In file systems, caching can mask the latency of accessing a local or even remote disk by allowing for quicker reads and writes. Another optimization made possible by client-side caching is write-behind in which write data is buffered in the cache while computation continues. 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 the one local process uses the system.

Distributed file systems use caching to hide the latency and bandwidth of the network. With caching in a distributed environment, there can be multiple cached copies of data,

Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)
IEEE COMPUTER SOCIETY

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

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 use 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. Expiring leases can also be used to ensure only one client at a time has file data cached. 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 the 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 will 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. Maintaining a client-side cache in a parallel environment is far more challenging than doing it in a distributed one. In a distributed environment, multiple copies of cached data on clients is somewhat avoidable, but in a parallel environment, there is no way around it. Ensuring that cached data on each node is safely accessed is not easy, and to date, there are no elegant solutions to this problem. IBM's General Parallel File System (GPFS) uses its distributed lock manager to enforce cache coherence [1]. Though overhead of the lock manager may be minimal depending on the computing environment, locking could potentially serialize what may otherwise be safe concurrent file access. On NFS, ROMIO uses byte range file locks to bypass the client-side cache. Not only does this introduce the potential for serializing file access, but the client-side cache is rendered useless along with all its performance benefits.

3 I/O Semantics

Consistency semantics dictate how strictly cache coherency should be enforced. Under POSIX specifications [6], newly written data should always be sequentially consistent and visible to all processes after a write routine has returned. Writes are also atomic, meaning a read should never return partially written data, a write has either completed or not started. MPI semantics are slightly more relaxed in that sequential consistency is guaranteed only under certain conditions and data needs to be visible to only the processes in the communicator group 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

Since POSIX consistency semantics are more stringent than MPI semantics, it would seem 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. This issue is further discussed by Liao *et al.* in [7]. MPI also requires that upon write completion, new data must be visible to all processes within the same communicator group.

4 ROMIO Collective I/O Implementation

ROMIO [9] is an implementation of the I/O functions of the MPI-2 standard, and is developed at Argonne National Laboratory. ROMIO will run on any file system for which there exists a working Abstract Device (ADIO) implementation. ROMIO presently supports a mix of commonly used local, distributed, and parallel file systems including SGI's XFS, IBM's PIOFS, Intel's PFS, NFS, and the Parallel Virtual File System. Since an ADIO layer is implemented for each individual file system, it can take advantage of any advanced features a specific file system may provide.

4.1 Data Sieving

The data sieving optimization, presented by Choudhary *et al.* [2, 11], 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. 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. If the aggregate access of a write is non-contiguous, a data sieving write must first read the contiguous section of the file to modify and then write back the entire section. The underlying file system is required to support locking to ensure correctness in a data sieving write.

4.2 Two-phase I/O

ROMIO uses a technique called two-phase I/O [10] to perform collective I/O operations. In two-phase I/O, a designated subset of processes partition I/O responsibilities for

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

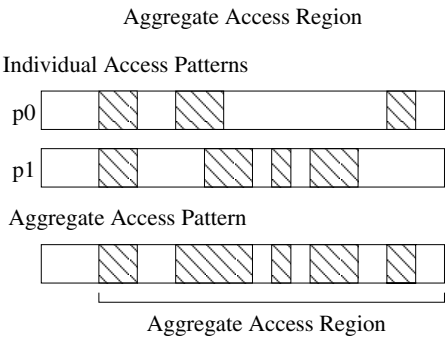

Figure 1. Aggregate access region illustrated.

a file among themselves. These processes are called aggregators. The ROMIO implementation bases this division on the very first file offset and very last file offset of the collective access shown in Figure 1. This aggregate access region of a collective I/O call is divided and I/O responsibility for each partition doled out to each process. This way, each of these file domains is $\lceil \frac{AAR}{N} \rceil$, where *AAR* is the size of the aggregate access region and *N* is the number of processes. By evenly splitting the aggregate access region, I/O is heuristically balanced over the different aggregating processes. The first thing a collective read or write routine does is partition the aggregate access region. Then in the case of a collective read, each process reads data within its assigned region (file domain) using the data sieving technique and distributes this data to the appropriate processes that actually requested the data. In ROMIO, the actual reads are done using the data sieving technique described in section 4.1 above. ROMIO's collective I/O routines use a default buffer size of 4 MB for buffering the underlying contiguous I/O. The data sieving buffer used in the collective routines is referred to as the collective buffer. ROMIO will try to read up to 4 MB of a file before trying to copy data between the collective I/O buffer and the user's memory. Under a collective write, each process will need to first exchange the data to be written before writing the data to their respective file domains. Without locking or any explicit cache management, the file state may not be consistent.

While the file consistency issue remains fundamentally the same with respect to two-phase I/O, the details are a little more complex since the process making a read request may not actually do its own I/O. Take this simple example in illustrated in Figure 2. Figure 2 presents two logical views of the file from each clients perspective. The upper file representation is the file data in the application's memory. The second view indicates what sections of the file are cached in the client-side file cache. Assuming there are two processes available, *p0* will be responsible for I/O in the the first half

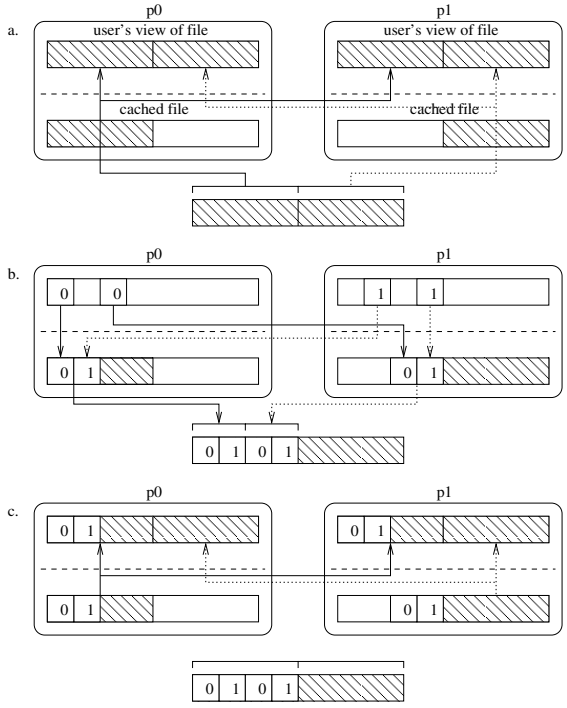

Figure 2. Example file consistency problem with collective I/O. In a., the entire file is collectively read. In b., the first half of the file is collectively written, and in c., the entire file is collectively read again.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

of the file, and *p1* for the second half, 2a. Note that *p0* has cached the first half of the file, and *p1* the second. Each process will then set a new interleaved view of and performa a collective write. In Figure 2b each process tries to write its rank in an interleaved pattern of two integers each to the first half of the file. The new set of file domains reflects the new access pattern, and assigns I/O for the first two integers to *p0*, and the second two integers 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 domains are assigned, *p0* read its file domain from its client-side cache where the last half is already stale, see Figure 2c. 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 illustrates the file consistency problem using collective I/O.

5 Intuitive Solutions

While locking is used by ROMIO and GPFS to side step cache coherency issues, a solution without locking is preferable. File locking itself incurs a high enough cost that some large scale systems sacrifice 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 provids. Reads and writes will always have to go over the network to the server for data. Write-behind, a file system optimization that will let several smaller write requests accumulate before actually sending the data to the network, cannot work without a cache to buffer the write data.

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 ENFS. Invalidating the client-side cache prior to every read guarantees that stale data is never retrieved by ensuring no data at all is 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 job 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.

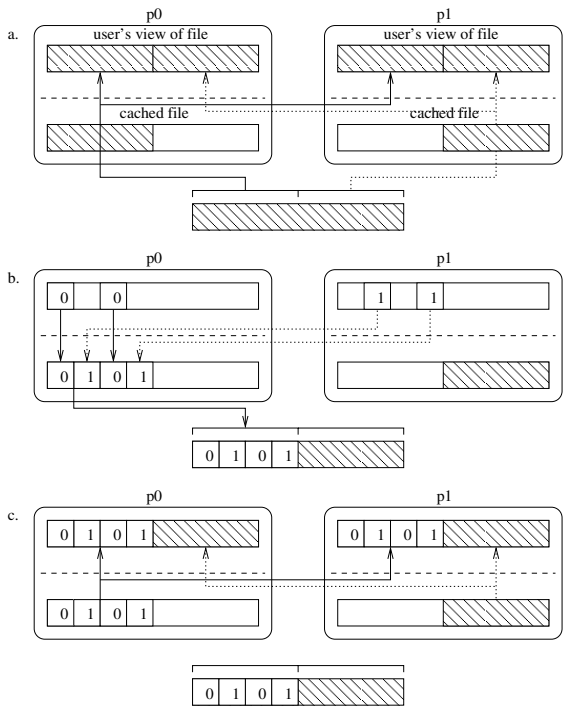

Figure 3. Example file consistency problem solved with persistent file domains.

6 Persistent File Domains

On a file system that does non-coherent client-side caching, file inconsistencies in ROMIO's collective I/O implementation stem from changes in the aggregate access regions and therefore file domains, between collective read and write calls. By making sure file domains 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 domain. Any data within the process's file domain that it has cached must be coherent since no other process is allowed to modify that data directly. In Figure 3 the same example as in Figure 2 is used, but this time using persistent file domains (PFDs). The file domains used in Figure 3a are used in the rest of the collective I/O operations, and the data read at the end of Figure 3 is now the same as the data residing on disk.

The original ROMIO two-phase implementation determines file domains by dividing the aggregate access region

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

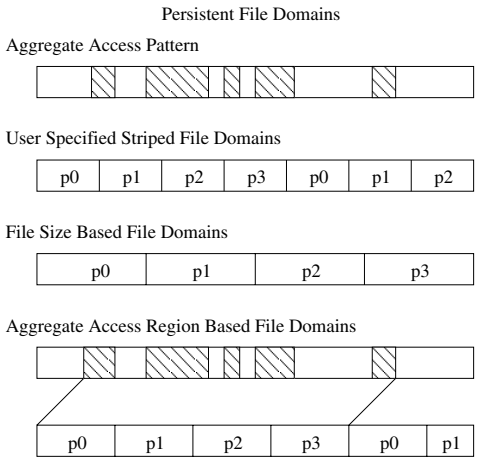

Figure 4. Assignment methods for persistent file domains given an aggregate access pattern and 4 processes.

by the number of aggregating processes with the intent of balancing I/O. Using this technique, one would hope that all available aggregators will be doing approximately the same amount of actual I/O. Given the noncontiguous and regular nature of the I/O access patterns scientific applications use [12], this heuristic should work well enough and will be considered the ideal I/O balancing scheme for the remainder of this paper. With this balancing scheme in mind, PFDs should not out perform file domains instantiated on a per collective I/O call with respect to actual I/O since they are not allowed to adapt to the access region of each individual I/O call. PFDs do allow MPI applications to safely leverage the performance benefits of the underlying file system's client-side caching. This is where the advantage of PFDs lies. So we are accepting a potentially less than optimal I/O balance for safe use of the cache. Mileage on this tradeoff is application dependent. This leaves the implementation choice of how to assign PFDs such that I/O responsibility is at least somewhat balanced.

6.1 User Specified Striped File Domains

By assigning PFDs based on a fixed PFD size, different access patterns can be specifically accommodated. The persistent file domain size passed down by the user through an MPI_File_Hint is used to cyclically assign PFDs to aggregating processes, figure 4. Using this technique, a user can optimize the persistent file domain size according to the application's file access pattern. To optimize the PFD 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 some idea of what kind of access pattern the application has, and what the appropriate PFD size is. A PFD 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 carry the burden. A PFD 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 PFD size should remain consistent as well.

6.2 File Size Based File Domains

Rather than evenly dividing the aggregate access region among processes, the entire file itself could be divided, Figure 4. File size based PFD sizes allows the user to concentrate more on the application task, but the file domain 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 domain sizes should perform well unless the file gets very large and the file domain size gets larger than the collective buffer size. A collective operation that spans a relatively small portion of the file will result in less available processes actually being used to do I/O.

A file size based file domain assignment scheme is the most natural extension of the original 2-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. How is the size of the file found at run time, considering a file may be newly created or appended to? Since the original file domains in 2 phase I/O did not exist outside of a single collective call, this was not an issue. Basically, 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 PFDs). Since these demands on the user were deemed unreasonable, this implementation was used as a proof-of-concept to show PFDs would indeed solve the file consistency problem. Instead we opted to use the implementation of striped file domains in section 6.1 to achieve file size based file domains.

By leveraging the striped file domains implementation, we can emulate the file size based file domains of the initial simplistic implementation, while retaining the more flexible attributes of the striped implementation. In this assignment strategy, a PFD size is determined based on the impending file size during the first collective I/O call. The impending file size is used because an collective write call could create or otherwise alter the size of the file. The PFD size is the impending file size divided by the number of aggregating processes. This way, the user does not need to know ex-

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

actly how large the file will be in advance since the striped assignment will allow for any later appending to the file.

6.3 Aggregate Access Region (AAR) Based File Domains

AAR based PFDs, like file size based PFDs relieve the application developer from having to manually calculate an optimal PFD size. Additionally, AAR based PFDs should accommodate a larger variety of access patterns than file size based PFDs. Both calculate a PFD size at run-time during the first collective I/O call. The difference is that in this assignment strategy, PFD size is size of the AAR of the first collective I/O call divided by the number of aggregating processes, Figure 4. 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 domains will have only been optimized for the initial access pattern. This last shortcoming leads to seemingly paradoxical dynamic PFDs.

6.4 Dynamic Persistent File Domains

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 PFDs, this event triggers a recalculation of PFD sizes based on the latest access pattern. Before these new file domains 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 domains. 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 just didn't change its access patterns much.

7 Performance Implications

We ran several applications to illustrate the impact of PFDs 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 64KB PFD size
- 4MB uses a 4MB PFD size
- fsize uses a calculated PFD size based on file size
- AAR uses a calculated PFD 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 PFD size aimed at matching the access pattern (sliding window application only)

7.1 Machine Configuration

All of our tests were run on ASCI Cplant at Sandia National Laboratory [4].

Cplant is an Alpha Linux cluster with each compute node configured with one 600 MHz EV-6, 512(Ross) 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, a variant of NFSv2 developed at Sandia National Laboratory, in a single global file space. Each I/O server in turn is an NFS client to a central XFS server. No caching is done on the I/O servers. Since we were not allowed the luxury of dedicated I/O servers, we present the best bandwidths from between three to five runs.

We chose to implement PFDs in ROMIO for portability and because of the existing infrastructure. Our changes have been submitted to the MPICH developers for consideration as user enabled optimizations. We developed and tested our software with MPICH 1.2.4.

7.2 Sliding Window Benchmark

To illustrate the potential benefits of PFDs, 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 each 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 5. 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 with possibly 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

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

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

Table 1. Library calculated persistent file domain sizes for the sliding window experiment.

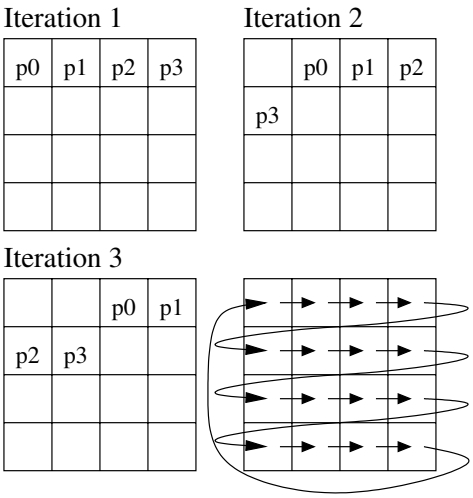

Figure 5. Several iterations of the sliding window I/O pattern for a 4x4 example. This continues until each process has read each tile.

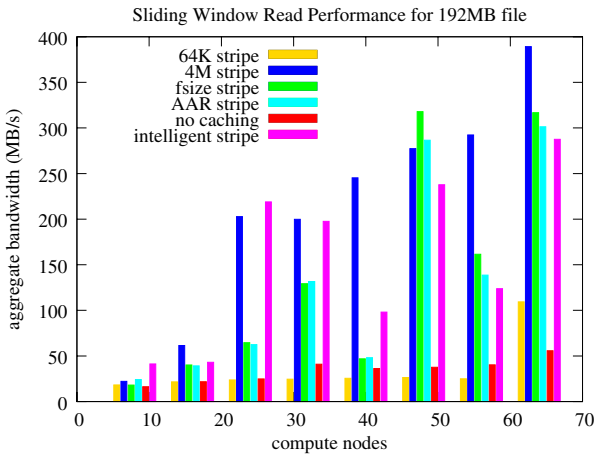

Figure 6. For most numbers of clients, the 4MB PFD allows the most pre-fetching.

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. The sliding window application was run with the default 4 MB collective buffer size. Table 1 lists the different PFD 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 PFD size. The "intelligent" stripe sizes are set by the sliding window application. This would represent a user specifying a PFDs 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:

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

where N is the number of processes and x is the number of tiles in a row. Since in theory, PFDs cannot achieve better actual I/O performance, the improvements over the

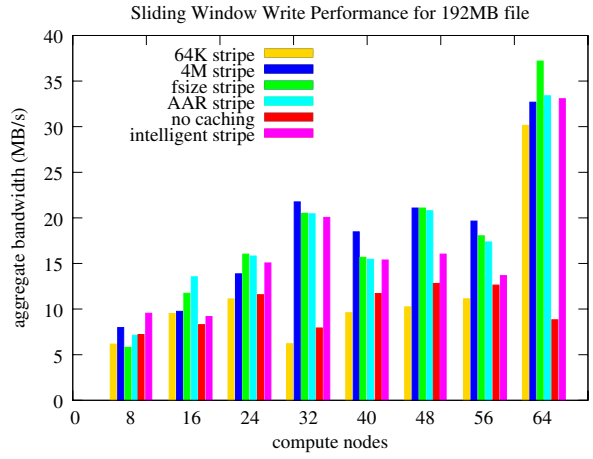

Figure 7. Write results are far more consistent between PFD sizes (except for 64 KB PFD size).

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

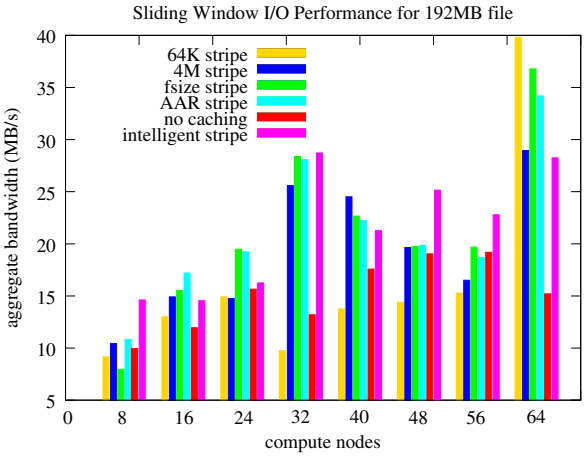

Figure 8. Write performance significantly brought down the overall I/O performance of the sliding window application.

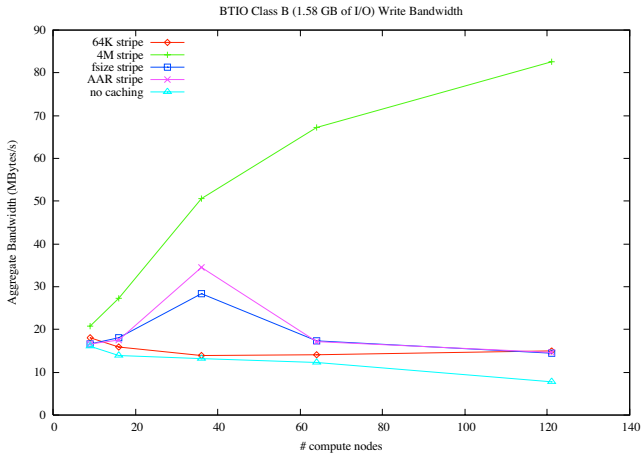

Figure 9. The larger 4MB PFD size allows for more effective use of the client-side cache.

non-caching implementation can be attributed to the effects of caching. From table 1, one would notice the PFD 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 PFD size. During the I/O phase, an aggregator can read no more than the size of the collective buffer or PFD size. The dynamics between the collective buffer, the PFD sizes, and the actual aggregate access regions of each collective I/O call introduce the large fluctuations exhibited by PFD sizes calculated during the initial I/O call in Figure 6. Other than the 64 KB PFD size, the PFD implementations consistently out-perform the no-caching case. The library calculated PFD sizes resulted in the best overall performances. Write performance in Figure 7 is much more stable than read performance. Relative features are about the same, but less pronounced.

7.3 BTIO Benchmark

BTIO [8] is a benchmark from NASA's Advanced Supercomputing (NAS) Division's parallel benchmark suite (NPB 2.4). The Block-Triagonal (BT) flow solver provides a representative means of measuring performance of I/O systems. BTIO uses diagonal multi-partitioning domain 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 mpi_full.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 PFDs end up using the same PFD size. Table 2 lists the dynamically calculated PFD sizes. The first row of data applies to the read and write cases for the aggregate access region based persistent file domains and the read case for the file size based persistent file domains. The write runs actually delete the file before writing out to a brand new file, so the file size based PFD sizes for the write case are the same as the aggregate access region based file domains 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 1.5GB, the resulting file size based file domains are rather large, ranging from 180 MB to 13 MB in Table 2 depending on the number of available processes. The dismal fsize results in Figure 10 can be explained by the excessively large PFDs. As the number of nodes increases, and the file domain sizes become more reasonable, performance begins to approach that of the other implementations. This is a good example of why using file size based PFDs can be bad.

If chosen well for BTIO, PFDs seem to offer a marginal

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

	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 2. Library calculated persistent file domain 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.

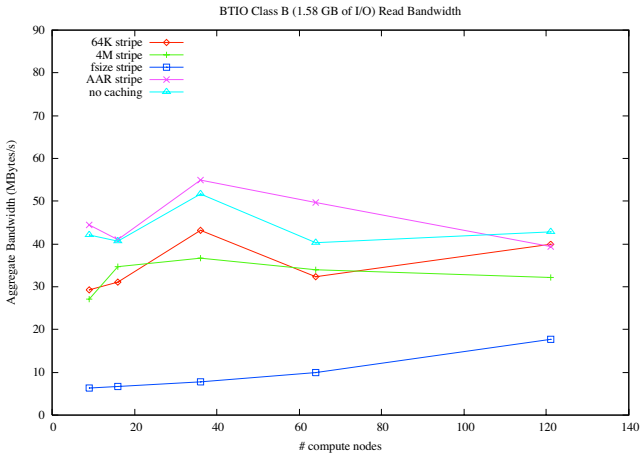

Figure 10. The large file domains calculated based on file size result in multiple I/O-communication phases, hurting performance.

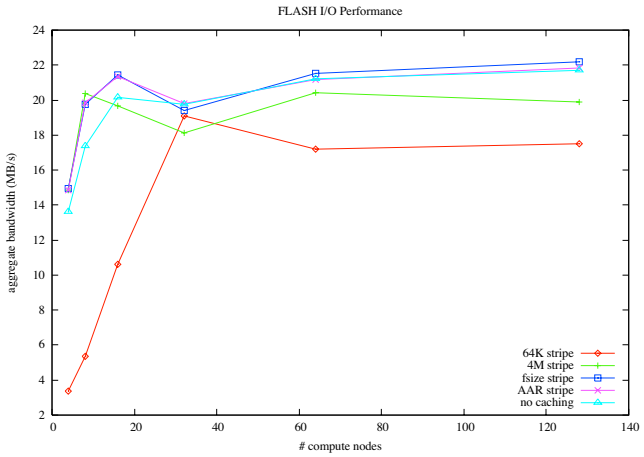

Figure 11. Results of the FLASH I/O benchmark with 4 - 128 processors.

amount of benefit over bypassing the cache completely for read performance. For the most part, BTIO's read performance does not benefit PFDs. As far as write performance is concerned, the constricting PFD sizes of the library calculated methods as the number of processes increases keeps performance well below that of the constant 4 MB PFD sizes, Figure 9.

7.4 FLASH I/O Simulation

The FLASH code [5] is used to model several types of thermonuclear flashes: hydrogen flashes on white dwarfs, helium flashes on neutron stars, and carbon flashed 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 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 domains are always 7680 KB. For every additional client, 80 FLASH (7MB) blocks are added to the file. So for client count that ranges between 4 and 128, the file sizes range between 28 MB and 896 MB. The PFD sizes for AAR and file size based PFDs are the same because the initial collective write is always the same. The 16 MB collective buffer size used easily accommodates the all the PFD sizes used. The FLASH benchmark and its I/O access patterns are further described by Ching *et al.* in [3].

Caching has a minimal impact on the write performance of FLASH except for runs with less than 32 clients, Figure 11. 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 PFD sizes for AAR and file size based PFDs is fairly good. The 4MB and 64KB PFD sizes are generating more I/O-communication phases than the other methods making their performances worse.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

8 Conclusions and Future Work

While useful, this work is only implemented in MPI's collective I/O routines. A future direction would be to extend persistent file domains to address non-collective independent I/O, possibly in MPI to keep things portable and since some parallel file systems do not perform client-side caching. In dealing only with the collective I/O routines, we were afforded some extra luxuries like explicit communication and synchronization. Eventually, we would like to see if we can eliminate the file locking bottleneck using knowledge of the application's access pattern and other relevant attributes.

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

Acknowledgements

This research was supported in part by the Department of Energy laboratories, SNL, LLNL, and LANL under subcontract No. P028264 and in part by the National Science Foundation under subcontract EIA-0103023.

References

[1] An introduction to GPFS 1.2. `http://www.ibm.com`, December 1998.

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

[3] A. Ching, A. Choudhary, W. Liao, R. Ross, and W. Gropp. Efficient structured data access in parallel file systems. In *Proceedings of the 2003 IEEE International Conference on Cluster Computing*, December 2003.

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

[5] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, and H. Tufo. FLASH: An adaptive mesh hydrodynamics code for modelling astrophysical thermonuclear flashes. *Astrophysical Journal Suppliment*, 131:273, 2000.

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

[7] W. Liao, A. Choudhary, K. Coloma, G. K. Thiruvathakal, L. Ward, E. Russell, and N. Pundit. Scalable implementations of MPI atomicity for concurrent overlapping I/O. In *Proceedings of the 2003 International Conference of Parallel Processing*, October 2003.

[8] NAS application I/O (BTIO) benchmark. `http://www.nas.nasa.gov`.

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

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

[11] R. Thakur, A. Choudhary, R. Bordawekar, S. More, and S. Kuditipudi. Passion: Optimized I/O for parallel applications. *IEEE Computer*, 29(6):70–78, June 1996.

[12] J. Worringen, J. L. Traff, and H. Ritzdorf. Improving generic non-contiguous file access for MPI-IO. In *Proceedings of the 10th EuroPVM/MPI Conference*, September 2003.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE