Appears in the Symposium on High Performance Distributed Computing 2005 (HPDC-14)

Collective Caching: Application-aware Client-side File Caching

Wei-keng Liao[†], Kenin Coloma[†], Alok Choudhary[†],
Lee Ward[‡], Eric Russell[‡], and Sonja Tideman[‡]

[†] Electrical and Computer Engineering Department
Northwestern University

[‡] Scalable Computing Systems Department
Sandia National Laboratories

Abstract

Parallel file subsystems in today's high-performance computers adopt many I/O optimization strategies that were designed for distributed systems. These strategies, for instance client-side file caching, treat each I/O request process independently, due to the consideration that clients are unlikely related with each other in a distributed environment. However, it is inadequate to apply such strategies directly in the high-performance computers where most of the I/O requests come from the processes that work on the same parallel applications. We believe that client-side caching could perform more effectively if the caching sub-system is aware of the process scope of an application and regards all the application processes as a single client. In this paper, we propose the idea of "collective caching" which coordinates the application processes to manage cache data and achieve cache coherence without involving the I/O servers. To demonstrate this idea, we implemented a collective caching sub-system at user space as a library, which can be incorporated into any Message Passing Interface implementation to increase its portability. The performance evaluation is presented with three I/O benchmarks on an IBM SP using it native parallel file system, GPFS. Our results show significant performance enhancement obtained by collective caching over the traditional approaches.

1. Introduction

In today's high-performance computers, many file subsystems are configured in a client-server model as illustrated in Figure 1. Usually the number of I/O servers is much less than the application nodes due to the system design being targeted mainly for computational intensive applications. Potential communication bottleneck can easily be formed when large groups of compute nodes make file requests simultaneously. Therefore, reducing the amount of data transfer between clients and file servers becomes important.

To achieve better I/O performance, client-side file caching is often considered as a scaling technique. It places a replica of repeated access data in the memory of the requesting processors such that successive I/O requests to the same data can be carried out locally without going to the file servers. However, storing multiple copies of the same data at different clients introduces the cache coherence problem [21]. Existing system-level solutions often involve the bookkeeping of the cached data at the I/O servers and require that I/O requests first consult the servers to see if it is safe to proceed. File locking is a common user-level solution for applications to obtain desired cache coherence because I/O can bypass the local system cache and directly access the servers when a file is locked. Since file locking is usually implemented in a centralized manner, it can easily limit the degree of I/O parallelism for concurrent file operations [12].

We propose a new design concept to perform client-side file caching for parallel applications, called *collective caching*. The motivation came from the inadequate use of client-side caching by the traditional approaches that consider each I/O request independently, assuming no correlation between the requests from different clients. While this strategy may be suitable for distributed file systems, for parallel applications that often work on the same data structures and perform concurrent I/O to a shared file, it can aggravate the cache coherence problem. Collective caching, on the contrary, considers all processes that run the same application as a single client and coordinates the processes to perform file caching. The design concept consists of the followings: 1) the caching sub-system knows the scope of processes that run the same application; 2) a global cache pool is logical constructed from the local memory buffers of all the processes; 3) cache metadata is distributed across the processes; and 4) both global cache pool and metadata are accessible to every process. Since

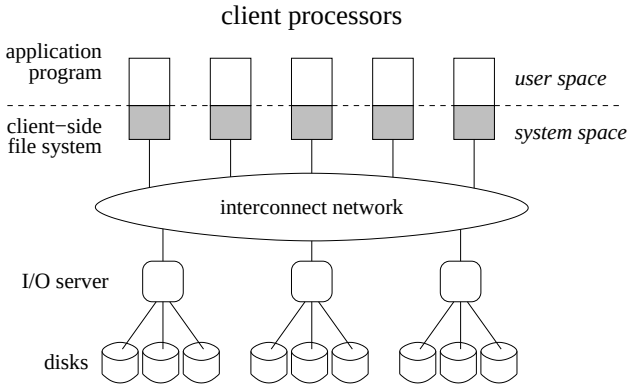

Figure 1. Typical configuration of a parallel file system in today's high-performance computers.

collective caching manages cached data among application clients, coherence problem can be solved without the involvement of I/O servers.

Collective caching can be implemented either at user space as an I/O library or at system space by incorporating into client-side file systems. Placing collective caching in a file system requires a new set of programming interfaces to pass the information of processes scope from an application to the kernel, which is not currently defined in POSIX standard. However, such information can be easy to obtain if the collective caching is implemented at user space. To demonstrate the idea of the collective caching, we implement a thread based caching sub-system and embedded it into the Message Passing Interface (MPI) library. The I/O thread is responsible for processing I/O requests, remote cache data access, and the cache metadata management. The thread catches `read()/write()` system calls from applications and determines whether the I/O request should go to the file servers or the caching sub-system. Experimental results presented in this paper were obtained from the IBM SP at San Diego supercomputing center using its GPFS file system. Three sets of I/O benchmarks are provided: sliding-window I/O, BTIO, and FLASH I/O. Compared with the traditional approach that uses either byte-range file locking to enforce the cache coherence or simply native UNIX read/write calls, collective caching shows a significant performance enhancement.

The rest of the paper is organized as follows. Section 2 discusses related works and background. Section 3 describes the idea and principles of collective caching. We present our implementation for collective caching in section 4. The performance results are given in section 5 and the paper is concluded in section 6.

2. Related works

Client-side file caching is often used in a distributed environment in order to reduce data transfer between file servers and clients. Although the I/O performance is enhanced, caching introduces the coherence problem. System-level solutions for cache coherence problem usually involve the bookkeeping of caching status at the servers and a client must consult with the servers before proceed with its I/O request. An example is Network File System v.4 [20]. Client-side caching is also used in some parallel file systems, for example, GPFS [17, 19], Lustre [13], and Panasas [16]. Similar strategies are adopted in these file systems to deal with the cache coherence problem.

GPFS employs a distributed lock management to enforce coherent client-side cache data, in which lock tokens must be granted before any I/O operation can be performed on the cache data. To avoid centralized lock management, a token holder becomes a local lock manager which is responsible for granting tokens for any further requests to the byte ranges it locks. Lustre file system uses dedicated cache servers to manage the cache metadata and read requests for a file are serviced in two phases. A lock request precedes the actual read request and the cache servers can assess where in the cluster the data has already been cached to include a referral to that node for reading. Although cache data referral means direct data communication between clients without going through the servers, only read operations can benefit from it. Nevertheless, cache metadata management is performed at the servers which are resources to be shared all applications and can potentially result in communication contention. As will be described later, the caching scheme proposed in this work puts the overhead of any effort for maintaining cache coherence on the clients where all the trouble is created. Panasas file system also employs metadata servers to maintain client-side cache coherence through using a callback. A client's read/write request is first registered with the servers with a callback. If conflicted cache data access occurs, then all clients that have callbacks are contacted and appropriate write-back or cache invalidation is performed. Therefore, data are transferred back/forth to the servers each time the coherence control occurs. Some parallel file systems leave the responsibility to users for coherent results. For instance, ENFS [3] does not enforce any coherence semantics and PVFS [2] provides no client-side caching at all.

Cooperative caching was proposed to deal with the coherence problem by coordinating the client's cache and allowing requests not satisfied by client's local cache be satisfied by the cache of another client [5]. Systems that use cooperative caching are PGMS [23], PPFS [8], and PACA [4]. However, cooperative caching also considers the system-level solutions without the concept of regarding applica-

tions as individual clients. Clusterfile parallel file system integrates cooperative caching into MPI collective I/O operations by using cache managers to manage a global cache consisting of memory buffers from both clients and servers [10].

A common user-level solution is to use the byte-range file locking facility for applications to achieve the desired coherence. When a file range is locked, all I/O operations within the range will go directly to the I/O servers bypassing the client's file system cache. This approach is adopted by ROMIO, a popular MPI I/O implementation developed by Argonne National Laboratories [22].

2.1. MPI I/O

Message Passing Interface (MPI) standard defines the programming interface and functionality for developing parallel programs that explicitly use message passing to perform the inter-process communication [14]. MPI standard version 2 extends the interface for the file I/O operations [15]. MPI I/O inherits two important MPI features: the ability to define a set of processes for group operations using an MPI communicator and the ability to describe complex memory layouts using MPI derived data types. A communicator specifies the processes that participate in an MPI operation, either an inter-process communication or an I/O request to a shared file. For the file operations, it is required to supply an MPI communicator when opening a file in parallel to indicate the processes that will later access the file.

In general, there are two types of MPI I/O operations: collective I/O and independent I/O (or non-collective I/O). The collective operations require all the processes that synchronously open the file participate the calls. During the synchronization, many collective I/O implementations take such opportunity to exchange the access information among all the processes in order to generate a better I/O strategy. Examples are the two-phase I/O and disk-directed I/O [6, 11]. The independent I/O, on the other hand, does not require the process synchronization, which makes any optimization practically difficult.

3. Collective caching

As used in MPI I/O, the term of *collective* means that the application processes work together to complete an operation. The key idea of collective caching is to consider all the processes as a single client to the file servers, so that cache data can be managed collectively within the client processes. This strategy contrasts with treating each application process independently as adopted in most of the existing parallel file systems. In the distributed environment, such as LAN, WAN, and Internet, the requirement

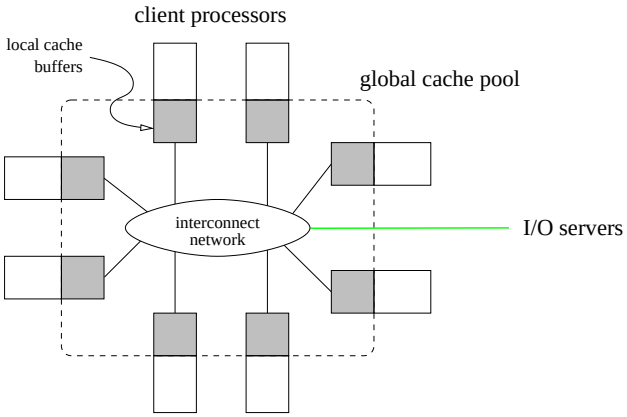

Figure 2. The alternative view of client-server model for collective caching, where the application processes form a single client. A global cache pool comprises the cache buffers from all the processes. Caching is performed by collaborating the client processes.

for cache coherence is usually relaxed since the requesting clients are often not related to each other. However, on parallel computers in which application processes work together to solve a single problem, the I/O requests are usually correlated. Therefore, the requirement of cache coherence can be stringent for parallel applications. If client-side caching is performed independently, the cost of maintaining cache coherence may easily overwhelm the I/O costs.

Collective caching relieves I/O servers the workload of maintaining coherent client-side cache to the application clients. Once the data reaches the clients, any effort to keep the cached data coherent will be performed by the client processes collectively. As a result, the I/O servers are solely responsible for transferring data from/to the clients. The I/O servers need not know the process scope of the application clients. At the client side, a file data can be cached locally at any process in the same application process group. Cached data and its metadata must be accessible to all the processes at any time. This can be achieved either by actively informing the status of one modified cache to all processes or by passively letting a process to query the metadata when needed. From a logical point of view, data is cached in a global cache pool that comprises the memory buffers from all the processes, as illustrated in Figure 2. To avoid cache coherence problem, a read/write request must first obtain the cache metadata and, then, determine whether the request should go to either the cache pool or the I/O servers. The management of cache metadata is distributed among the application processes, so that the communication contention for accessing metadata can be minimized.

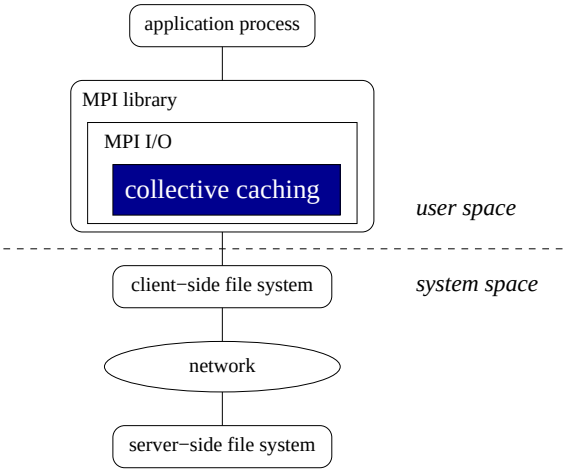

Figure 3. Software layout of our collective caching implementation which resides in MPI I/O library sitting on top of the file system.

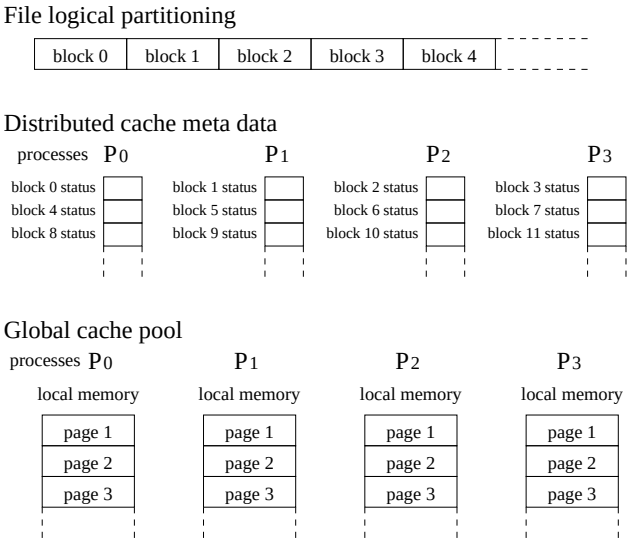

Figure 4. The organization of cache metadata and cache pages. Cache metadata are distributed among the processes in a round-robin fashion.

4. Design and implementation

For demonstration, we choose to implement the collective caching at user space which can be incorporated into the MPI library. One advantage by doing so is to increase the portability that makes our implementation independently sit on top of the file systems. Another advantage is being able to use MPI communicators directly to define the process scope of a parallel application, which is essential for collective caching. In MPI I/O, the handler of an opened file must be associated with a communicator provided at the time the file is opened. Figure 3 shows the software design layout of our implementation relative to the MPI I/O library and file systems. Our design principles include the following.

- The processes specified in the MPI communicator of an opened file are considered as a single client.
- Similar to the traditional caching policies, cache data is placed as close to the requesting processes as possible.
- Once a data is cached by a process, its status is made available to all the processes. Any query to the caching status and cache data shall not stop the execution of any process.
- To keep all the cache data coherent, we construct a *global cache pool* from memory buffers of all the processes.
- At most one copy of the file data can be cached in the pool at any moment. Although this condition can be relaxed for a more aggressive caching, we choose this simple approach for demonstration purpose.

The following sections describes the building blocks of our caching sub-system: management of cache data and metadata, the I/O thread handling requests to cache data, and caching policies.

4.1. Management of cache pages and metadata

We first logically divide a file into blocks of the same size in which each block represents an indivisible page that can be cached in a process's local memory. Cache metadata describing the caching status of these file blocks is distributed in a round-robin fashion among the processes that together open the file. The metadata for block i is held by the process of rank (i mod $nproc$), where $nproc$ is the number of processes defined in the MPI communicator. Note that the location of cache metadata is fixed (cyclically assigned among processes) but a file block can be cached at any process. Figure 4 illustrates the organization of cache pages and their metadata. A global cache pool consists of local cache buffers from all processors. Accessing to a file blocks cached in the global pool will be through either local memory copy if the block is cached locally or remote memory access otherwise. Note that the cache pool is shared by all the opened files, but cache metadata is unique to each file.

The contents of a cache metadata includes the file descriptor, file offset, current owner process id, a dirty flag, byte range of the dirty data, and the locking status. Prior to performing an I/O request, a process must first check the

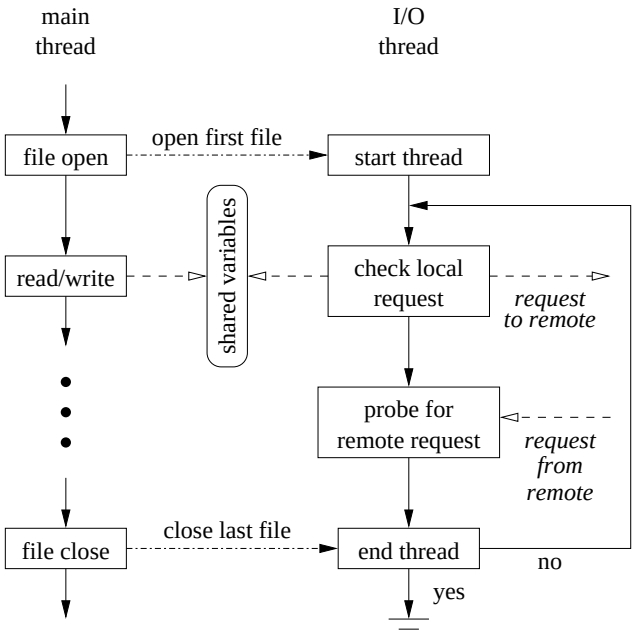

Figure 5. The design of the I/O thread.

caching status of the blocks to be accessed in the request. If the requested blocks have not been cached by any process, the requesting process will cache them locally by reading them from the file servers. Otherwise, the request will be forwarded to the owner(s) that cache the blocks locally. To simplify the cache coherence control, we allow at most one copy of the file data to store in the global cache pool at any time.

4.2. I/O thread

Since cache data and metadata are distributed among processes, each process must be able to response to remote requests for accessing to data stored locally. For MPI collective I/O where all processes must be synchronized, fulfilling remote requests can be achieved by first making each request known to all processes and, then, using interprocess communication to deliver data to the requesting processes. On the contrary, MPI independent I/O is asynchronous which makes it difficult for one process to explicitly receive remote requests. Therefore, our design needs a mechanism to allow a process to access to remote memory without interrupting the execution of the remote processes. There are two possibilities to achieve this goal. One is to use *remote memory access* (RMA) functions provided by the MPI version 2 and the other is to create a separate thread in each process to handle the remote requests. The RMA approach allows a process to perform one-side communications to a remote process once the memory buffers declared as RMA accessible. Our implementation using RMA approach is currently under development.

We choose the I/O thread approach. To increase the portability, our thread programming uses the POSIX standard thread library [9]. The I/O thread can be embedded inside the MPI I/O calls so that caching can be performed implicitly and let main thread continue its task without interruption. Figure 5 illustrates the I/O thread design from the viewpoint of a single process. Details of our design are described as follows.

- The I/O thread is created when the application opens the first file and destroyed when the last file is closed. Each process can have multiple files opened, but only one thread is created.
- Once the I/O thread is created, it enters an infinite loop to serve the local and remote I/O requests until it is signaled by the main thread for its termination.
- All I/O and communication operations are carried out by the I/O thread only. For blocking I/O operations, once the main thread signals the I/O thread, it waits for the I/O thread to complete the request. For non-blocking I/O, the main thread can continue its task but must explicitly call a wait function to ensure the completion of the request. This design conforms to the MPI blocking and non-blocking I/O semantics.
- A conditional variable protected by a mutual exclusion lock is used to indicate if an I/O request has been issued by the main thread or if the I/O thread has completed the request. The communication between the two threads is also through a few shared variables that store the file access information, such as file handler, offset, memory buffer, etc.
- To serve remote requests, the I/O thread keeps probing for incoming I/O requests from any process in the MPI communicator group. Since each opened file is associated with a communicator, the probe will check for all the opened files.
- The types of local requests are: file open, close, read, write, synchronization, and thread termination.
- The remote I/O requests include get/put data from/to cache pages, inquiry, lock/unlock, and reset the metadata.

4.3. Lock management and caching policy

To explain the execution flow and the caching policy implemented in our design, we use an MPI independent I/O operation as an example. The execution flow of an example read operation is given in Figure 6. When a process issues an independent I/O call, it's main thread first sets the

Local request to block *a*

1. send lock request to process ***(a* mod *nproc)***
2. wait for lock to be granted
3. if block ***a*** is not cached anywhere
4. read block ***a*** from file system
5. copy data to/from the request buffer
6. if block ***a*** is cached locally
7. copy data to/from the request buffer
8. if block ***a*** is cached in process ***p***
9. send request to process ***p***
10. send/receive data to/from process ***p***
11. send unlock request to process ***(a* mod *nproc)***

Figure 6. The execution flow for a read request to file block *a*.

access information into the thread-shared variables and signals the I/O thread. Once signaled, the I/O thread uses the current file pointer position and the request length to identify the file blocks covered by the request. For each file block, the I/O thread sends a lock request to the process that holds the block's metadata. The lock to the metadata must be granted before any read/write can be performed on the block. The locks are only applicable to the metadata rather than the cache pages.

If the metadata is currently locked, the request will be added into a queue in the process that holds the metadata and the requesting process must wait for the lock to be granted. For the I/O request that covers multiple blocks, we enforce the locks to be granted in an increasing order. For example, an I/O request covers file blocks i to j, where $i \leq j$. Lock request for block k, $i \leq k \leq j$, will not be issued until lock to block $(k-1)$ is granted. In addition, all locks must be granted before any I/O operations can be performed on any of the blocks. This design is similar to the two-phase locking method [1] used to serialize multiple overlapping I/O requests to guarantee the I/O atomicity.

Once a lock to a file block is granted, the caching status of the requested block is retrieved. If the status indicates the block is not cached anywhere, the requesting process will cache the block in its own local memory and update the metadata accordingly. If the block is already cached locally, then a local memory copy can fulfill the request. If the block is cached by a remote process, the request will be forwarded to the process that holds the cache page. Note that file caching is performed in the granularity of file blocks when moving data from/to the file system. Unlike caching, when accessing to a remote cache page, only the requested data is transferred between processes, not necessarily the whole blocks. When the request completes, the lock to the metadata is released.

The global cache pool consists of local memory buffers from the processes in the same communicator group, which are reserved from the local memory when the I/O thread is created. Similar to the file system, cache pool is shared by all opened files. We let each process manage its local cache pages independently based only on the local usage. When the reserved memory is full, we choose the least-recent-used eviction policy in our implementation. For the machine platforms of which file system itself performs client-side caching, we wrap each read/write call with a byte-range locking to bypass the system's cache in order to prevent the possible cache incoherence.

5. Experimental results

The evaluation of our implementation for collective caching was performed on the IBM SP machine at San Diego Supercomputing Center. The IBM SP contains 144 Symmetric Multiprocessing (SMP) compute nodes and each node is an eight-processor shared-memory machine. We use the IBM GPFS file system to store the files. The peak performance of the GPFS is 2.1 GBytes per second for reads and 1 GBytes per second for writes. The I/O will approximately max out at about 20 compute nodes. In order to simulate a distributed-memory environment, we ran the tests using one processor per compute node. We present three benchmarks: a sliding-window access pattern, NASA's BTIO benchmark, and FLASH I/O benchmark.

5.1. Sliding-window benchmark

In order to simulate a repeated file access pattern that potentially can cause cache coherence problem, we constructed a sliding-window I/O test code, as depicted in Figure 7. The inner loop, j, is the core of sliding-window operation and the outer loop, i, indicates the number of iterations. In each inner loop, a read-modify-write operation on two file blocks is performed. A different file segment is accessed at a different outer loop. In the sliding-window access pattern, every process will be able to read and write the data modified by all other processes.

We compare the collective caching with the method of using byte-range file locking. This comparison is under the assumption that cache coherence must be enforced at any time. We wrap each read/write call with a byte-range locking call in the sliding-window test code. The performance results are presented in Figure 8. File sizes used in our tests range from 32 MBytes to 1 GBytes. We used the file block size of 256 Kbytes. The bandwidth numbers are obtained

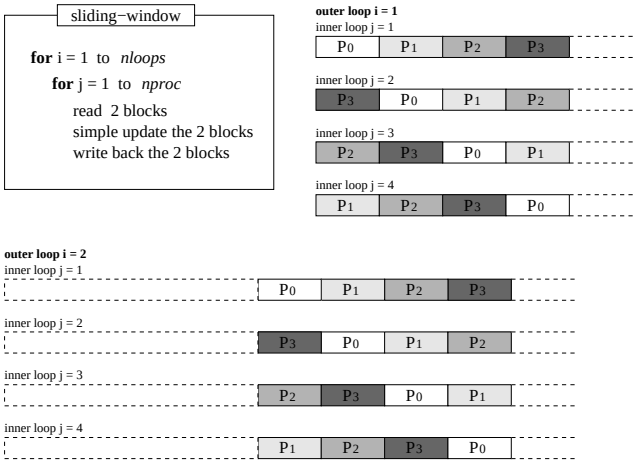

Figure 7. The sliding window access patterns.

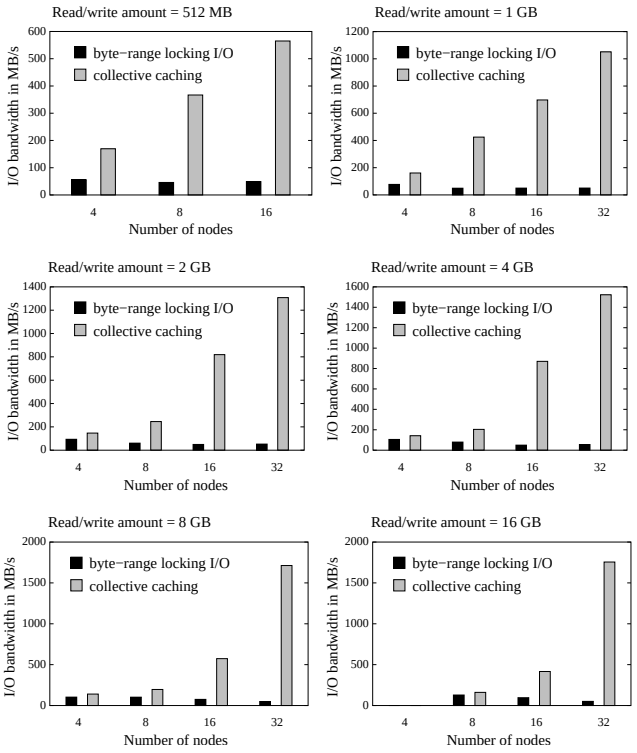

Figure 8. I/O bandwidth for running the sliding-window access pattern.

by dividing the I/O amount by the execution time measured from file open to close.

The experimental results clearly show a much better performance obtained by using collective caching. Especially, good speedups are observed for the case when using 32 compute nodes. In this case, the byte-range file locking approach suffers from the serious contention of lock requests to the common file regions from as many as 32 processes. The overhead of using the byte-range locking approach also includes the communication cost for accessing data from/to the servers each time an I/O operation is performed. In principle, I/O would perform better if clients access the data from each other's memory than accessing to the file servers. Especially when running a large number of processes, reducing the involvement of the file servers can significantly improve the I/O performance.

5.2. BTIO benchmark

BTIO is the I/O benchmark from NASA Advanced Supercomputing (NAS) parallel benchmark suite (NPB 2.4) [24]. BTIO uses a block-tridiagonal (BT) partitioning pattern on a three-dimensional array across a square number of compute nodes. Each processor is responsible for multiple Cartesian subsets of the entire data set, whose number increases as the square root of the number of processors participating in the computation. The benchmark performs 40 collective MPI writes followed by 40 collective reads. BTIO provides four types of evaluations, each with different I/O implementations, including MPI collective I/O, MPI independent I/O, Fortran I/O, and separate-file I/O. In this paper, we only present the performance results for the type of using MPI collective I/O, since collective I/O generally results in the best performance. We evaluated two I/O sizes: classes A and B, which generate I/O amount of 800 MBytes and 3.16 GBytes, respectively.

Figure 9 compares the bandwidth results for the implementation with collective caching and the native approach (without collective caching.) Note that IBM MPI collective read/write calls are performed for both with and without collective caching. Since MPI collective I/O calls only generate non-overlapping I/O requests, byte-range locking is not needed in the native approach. Even though there is no lock contention in the native approach, we still can see that collective caching out-perform the native approach in most of the cases. Especially, when the number of compute nodes becomes large, collective caching can achieve sufficient bandwidth near to the system peak performance. The main contribution to the performance improvement is due to the user-level caching acting as a large write-back cache. Many small write data are aggregated and later written by a large write request.

5.3. FLASH I/O benchmark

FLASH is an adaptive mesh refinement application that solves fully compressible, reactive hydrodynamic equations, developed mainly for the study of nuclear flashes on

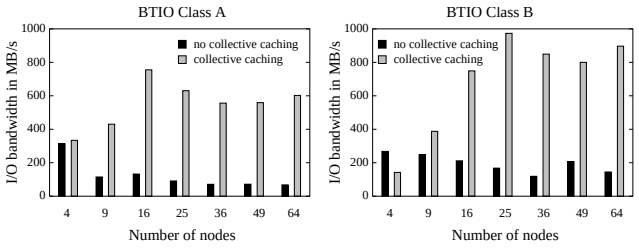

Figure 9. Bandwidth results for BTIO benchmark. BTIO contains 40 collective MPI collective writes followed by 40 collective reads. The aggregated I/O amount for classes A and B are 800 MBytes and 3.16 GBytes, respectively.

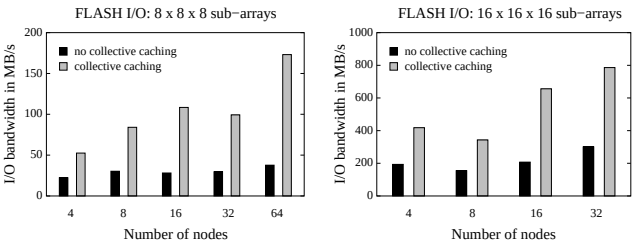

Figure 10. Bandwidth results for FLASH I/O benchmark. The I/O amount is proportional to the number of compute nodes, ranging from 72.94 MBytes to 1.14 GBytes for the case of $8 \times 8 \times 8$ arrays and from 573.45 MBytes to 4.49 GBytes for the case of $16 \times 16 \times 16$ arrays.

neutron stars and white dwarfs [7]. The FLASH I/O benchmark [25] simulates the I/O pattern of FLASH, which uses HDF5 for writing checkpoints, but underneath is using MPI I/O for performing parallel reads and writes. FLASH I/O produces a checkpoint file, a plot file with centered data, and a plot file with corner data. The in-memory data structures are 3D sub-arrays of size $8 \times 8 \times 8$ or $16 \times 16 \times 16$ with a perimeter of four guard cells that are left out of the data written to files. In the simulation, 80 of these blocks are held by each processor. Each of these data elements has 24 variables associated with it. Within each file, the data for the same variable must stored contiguously. The access pattern is non-contiguous both in memory and in file, making it a challenging application for parallel I/O systems. Since every processor writes 80 FLASH blocks to file, as we increase the number of clients, the dataset size increases linearly as well.

In order to focus on the data I/O performance, we only measured the time for read/write requests. Figure 10 shows the bandwidth results for comparing the I/O implementation with and without the collective caching. In the case of using $8 \times 8 \times 8$ array size, we can see that the I/O bandwidth is far from the system peak performance. This is because FLASH I/O generates many non-contiguous and small I/O requests and the system peak performance can only be achieved by large contiguous I/O requests. The bandwidth improves significantly when we increase the array size to $16 \times 16 \times 16$. Nevertheless, collective caching performs better than the native approach for both array sizes. Similar to the BTIO benchmark, collective caching also attains the effect of write-back caching which improves performance by aggregating multiple I/O requests for later a large write.

5.4. Performance Implication

As observed from the I/O bandwidth numbers from our performance evaluation, the results of using native IBM MPI collective calls are far from the system peak performance. The reasons can be the followings. First, an application I/O performance heavily dependents on its file access patterns. As mentioned earlier, system peak I/O bandwidth is achieved when large read/write calls are performed, which is not always the case for parallel applications. The use of file locking that serializes the I/O can be one of the reasons and patterns with many small read/write calls can also result in worse performance than large and fewer requests. As already known to operating system designers, write-back caching often can enhance the I/O performance significantly.

GPFS provides an option to enable data shipping mode for MPI I/O [18]. To prevent concurrent access of file blocks by multiple tasks, data shipping binds each GPFS file block to a unique I/O agent which is responsible for all the accesses to this block. Any I/O operations on GPFS must go through the I/O agents which will ship the requested data to appropriate processes. As indicated in [17], certain performance enhancement can be obtained for read operations, but the write performance degrades when data shipping is enabled. Due to this concern, all our benchmarks did not perform with the data shipping mode enabled. Although the I/O bandwidth may be enhanced if the I/O is fine-tuned with specific GPFS parameters, we expect limited performance improvement. This is because half of the I/O operations are writes in sliding-window and BTIO benchmarks and only writes in FLASH I/O. On the other hand, the proposed collective caching scheme is a portable user-level solution that has been proved to be able to achieve significant percentage of the system peak bandwidth. Comparing with platform-specific approaches, collective caching definitely

has demonstrated its advantage.

6. Conclusions

Collective caching is a new application-level client-side file caching design. The motivation came from the fact that directly applying existing caching strategies developed for distributed systems is inadequate for the parallel file system in today's high-performance computers. Instead of treating each process client independently as in traditional file systems, collective caching considers all the application processes as a single client and makes every caching status known to all the processes. Therefore, it is more efficient for collective caching to maintain the coherent state of the client-side cache. In this paper, we presented a user-space implementation that uses an I/O thread in each client process to handle the local and remote access to the cache data. The experimental results have shown a great improvement for MPI independent I/O operations over the traditional approach that uses the byte-range file locking. For collective I/O operations, we also demonstrated better performance results obtained by collective caching when evaluating BTIO and FLASH I/O benchmarks. In the future, we plan to investigate in depth the effect of the file block size and explore the possible I/O modes that can further help collective caching to deal with variety of access patterns.

7. Acknowledgments

This work was supported in part by Sandia National Laboratories and DOE under Contract No. 28264, DOE's SCiDAC program (Scientific Data Management Center), award No. DE-FC02-01ER25485, and NSF through the SDSC under grant ASC980038 using IBM DataStar.

References

[1] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[2] P. Carns, W. Ligon, R. Ross, and R. Thakur. PVFS: A Parallel File System for Linux Clusters. In *the Third Annual Linux Showcase and Conference*, pages 317–327, Oct. 2000.

[3] Computational Plant, Sandia National Laboratories. *ENFS - Extended NFS.* http :// www.cs.sandia.gov / cplant / doc / io / ENFS_User_Doc.html.

[4] T. Corts, S. Girona, and J. Labarta. PACA: A Cooperative File System Cache for Parallel Machines. In *the 2nd International Euro-Par Conference*, pages 477–486, Aug. 1996.

[5] M. Dahlin, R. Wang, and T. A. adn D. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *the First Symposium on Operating System Design and Implementation*, Nov. 1994.

[6] J. del Rosario, R. Brodawekar, and A. Choudhary. Improved Parallel I/O via a Two-Phase Run-time Access Strategy. In *the Workshop on I/O in Parallel Computer Systems at IPPS '93*, pages 56–70, Apr. 1993.

[7] 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*, pages 131–273, 2000.

[8] J. Huber, C. Elford, D. Reed, A. Chien, and D. Blumenthal. PPFS: A High Performance Portable File System. In *the 9th ACM International Conference on Supercomputing*, 1995.

[9] IEEE/ANSI Std. 1003.1. *Portable Operating System Interface (POSIX)-Part 1: System Application Program Interface (API) [C Language]*, 1996.

[10] F. Isaila, G. Malpohl, V. Olaru, G. Szeder, and W. Tichy. Integrating Collective I/O and Cooperative Caching into the "Clusterfile" Parallel File System. In *the 18th annual international conference on Supercomputing*, pages 58–67, June 2004.

[11] D. Kotz. Disk-directed I/O for MIMD Multiprocessors. *ACM Transactions on Computer Systems*, 15(1):41–74, Feb. 1997.

[12] W. Liao, A. Choudhary, K. Coloma, G. Thiruvathukal, W. Lee, E. Russell, and N. Pundit. Scalable Implementations of MPI Atomicity for Concurrent Overlapping I/O. In *the International Conference on Parallel Processing*, Oct. 2003.

[13] Lustre: A Scalable, High-Performance File System. *Whitepaper.* Cluster File Systems, Inc., 2003.

[14] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard, Version 1.1*, June 1995. http :// www.mpi-forum.org / docs / docs.html.

[15] Message Passing Interface Forum. *MPI-2: Extensions to the Message Passing Interface*, July 1997. http :// www.mpi-forum.org / docs / docs.html.

[16] D. Nagle, D. Serenyi, and A. Matthews. The Panasas ActiveScale Storage Cluster - Delivering Scalable High Bandwidth Storage. In *the ACM/IEEE Supercomputing Conference*, Nov. 2004.

[17] J. Prost, R. Treumann, R. Hedges, B. Jia, and A. Koniges. MPI-IO/GPFS, an Optimized Implementation of MPI-IO on top of GPFS. In *Supercomputing*, Nov. 2001.

[18] J. Prost, R. Treumann, R. Hedges, A. Koniges, and A. White. Towards a High-Performance Implementation of MPI-IO on top of GPFS. In *the Sixth International Euro-Par Conference on Parallel Processing*, Aug. 2000.

[19] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *the Conference on File and Storage Technologies (FAST'02)*, pages 231–244, Jan. 2002.

[20] S. Shepler, C. Beame, R. Callaghan, M. Eisler, D. Noveck, D. Robinson, and R. Thurlow. *NFS version 4 Protocol.* RFC 3010, Dec. 2000.

[21] A. Tanenbaum and M. van Steen. *Distributed Systems - Principles and Paradigms.* Prentice Hall, 2002.

[22] R. Thakur, W. Gropp, and E. Lusk. *Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation.* Technical Report ANL/MCS-TM-234, Mathematics

and Computer Science Division, Argonne National Laboratory, Oct. 1997.

[23] G. Voelker, E. Anderson, T. Kimbrel, M. Feeley, J. Chase, A. Karlin, and H. Levy. Implementing Cooperative Prefetching and Caching in a Globally-managed Memory System. In *the Joint International Conference on Measurement and Modeling of Computing Systems*, pages 33–43, 1998.

[24] P. Wong and R. der Wijngaart. NAS Parallel Benchmarks I/O Version 2.4. Technical Report NAS-03-002, NASA Ames Research Center, Moffet Field, CA, Jan. 2003.

[25] M. Zingale. FLASH I/O Benchmark Routine – Parallel HDF 5. http://flash.uchicago.edu/~zingale/flash_benchmark_io.