Revealing Applications' Access Pattern in Collective I/O for Cache Management

Yin Lu, Yong Chen
Computer Science
Texas Tech University
yin.lu@ttu.edu,
yong.chen@ttu.edu

Rob Latham
Mathematics and Computer Science Division
Argonne National Laboratory
robl@mcs.anl.gov

Yu Zhuang
Computer Science
Texas Tech University
yu.zhuang@ttu.edu

ABSTRACT
Collective I/O is a critical I/O strategy on high-performance parallel computing systems that enables programmers to reveal parallel processes' I/O accesses collectively and makes possible for the parallel I/O middleware to carry out I/O requests in a highly efficient manner. Collective I/O has been proven as a core parallel I/O optimization technique. However, due to the collective nature of collective I/O, the access pattern of each individual process can be lost after I/O requests are aggregated at the parallel I/O middleware layer. In this study, we analyze this issue in detail. We show that such lost access pattern can have a negative impact on underlying caching algorithms' view of locality and can result in many unnecessary cache misses in low level buffer caches and additional disk accesses. To address this issue, we propose to reveal unseen access patterns - performing collective I/O but more importantly retaining applications' access patterns to underlying cache management. With such an idea, we have prototyped a new collective I/O aware cache management methodology. The evaluations with various cache management algorithms have confirmed clear advantages over the existing collective I/O strategy that throws away applications' original access pattern.

Categories and Subject Descriptors
B.4.3 [**Input/Output and Data communications**]: Parallel I/O; D.4.3 [**File Systems Management**]: Access methods

General Terms
Algorithms, Design, Performance

Keywords
Parallel I/O, collective I/O, high performance computing

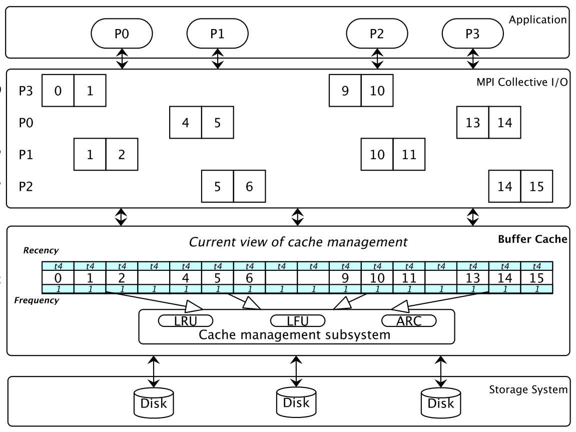
1. INTRODUCTION
Scientific applications, simulations, and visualizations running on high-performance computing clusters produce and consume growing massive amounts of data. For example, The EarthScience project hosted on Intrepid system accesses a total of 3.5 PiB of data volumes within two months [3]. Similarly, remarkable data volumes moved by accurate climate modeling expect to reach hundreds of exabytes by 2020 [8]. Such massive data sets require extreme amounts of I/O to store and retrieve results for later use and analysis. The disk access latencies of these data-intensive applications have resulted in I/O becoming a significant performance bottleneck.

Many efforts have been taken to tackle the I/O bottleneck issue from different angles. From the system architecture point of view, buffer caches are widely used in high performance storage systems to alleviate disk access latencies for data-intensive applications. Large-scale high performance computing platforms typically are hierarchically organized and can employ buffer caches in multiple layers. Such architectures can significantly enhance the scalability and availability of the systems and reduce I/O operation costs. Clearly, how to take advantage of such buffer cache hierarchy in high performance computing platforms for data-intensive scientific applications is critical from the performance point of view.

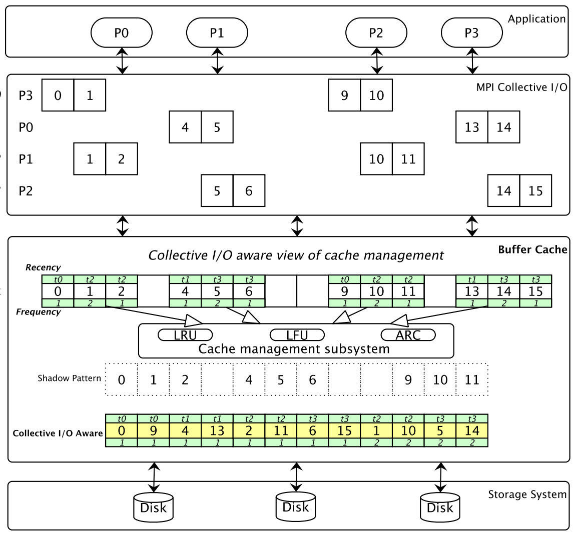
From the software perspective, a large-scale data-intensive application may use several layers of software for I/O optimizations. For example, I/O middleware such as an MPI-IO implementation organizes and coordinates I/O within applications using their access patterns. Collective I/O [18] is one of the most important I/O access optimizations in MPI-IO. MPI collective I/O layer in Figure 1a illustrates how a group of read operations can benefit from a collective routine. Four processes P_0, P_1, P_2 and P_3 from application layer request four data blocks at four times respectively. Collective I/O aggregates and services requests from all processes together instead of making several read calls separately. As shown in Figure 1a, the implementation of collective I/O aggregates requests from all those processes and exchanges their access offsets at time t_4. After analyzing the access requests of different processes, collective I/O filters overlapping requests, combines the interleaved noncontiguous requests, and carries out a large contiguous data access.

However, from this example we can observe that, with collective I/O the detailed access patterns available at the application level are changed when the aggregated I/O request reaches the low level buffer cache. How many times a block

ICS '14 June 10 - 13 2014, Muenchen, Germany
Copyright 2014 ACM 978-1-4503-2642-1/14/06 ...$15.00.
http://dx.doi.org/10.1145/2597652.2597686



(a) Collective I/O and current cache managment



(b) New cache managment

Figure 1: Collective I/O Hides Applications' Original Access Pattern (Recency and Frequency of Requested Blocks) Away

is requested and the original temporal information are all thrown away and not known to the low level buffer cache after being aggregated by collective I/O. Therefore data blocks could reside in low level buffer caches undiscerningly for a long period of time before they become cold enough to be replaced by a local replacement algorithm. Furthermore, collective I/O can potentially bring to low level cache more data elements than it needs. For instance, in Figure 1a, at t_4 four blocks in the combined large data chunk are extra data not truly required by processes. These extra data blocks increases the pressure on the low level buffer caches. The effective cache capacity is reduced, which in turn affects application performance. Without a proper coordination between I/O middleware and the low level buffer cache, the *shadow pattern* caused by collective I/O can lead to the buffer cache seriously under-utilized.

To address these limitations, in this paper we propose a collective I/O aware (CIO-aware in short) buffer cache management scheme, in which the buffer cache is exposed with the original pattern of access stream and has the better potential to exploit it. The buffer cache layer in Figure 1a versus the buffer cache layer in Figure 1b demonstrate the comparison of two cache layouts with the same application data accesses. One layout is the effect of the hidden access pattern stemming from current collective I/O. Another cache layout is optimized with our proposed CIO-aware approach. With the proposed strategy, the data elements in buffer cache are organized based on the actual pattern from application level (please note the difference of both recency and frequency between Figure 1a and Figure 1b) and stored in consecutive locations, which helps minimize the number of data blocks occupied in buffer cache and makes the cache management much more efficient than the existing strategy.

The primary contributions of the study are as follows:

- We investigate the impact of collective I/O on the low level buffer cache management and analyze the potential limitation and improvement.
- We propose a CIO-aware cache management scheme which integrates enhanced collective I/O module with pattern detection threads to improve the performance of underlying buffer cache without dedicating extra resources.
- Compared to current scheme, the beauty of the CIO-aware cache management is that, collective I/O is still performed, but more importantly the original true access patterns are revealed to low level buffer caches.
- We implemented CIO-aware cache management scheme within ROMIO [17], the most popular implementation of the MPI-IO middleware. Both pattern detection and cache management are transparent to the users and collective I/O interfaces remain unchanged. Furthermore, CIO-aware cache management is implemented in the file-system-independent layer of ROMIO, allowing it to be easily ported.
- We evaluated CIO-aware buffer cache management with three widely-used parallel I/O benchmarks. Our results show that CIO-aware buffer cache can significantly reduce the total run time and improve the applications' overall performance. Through our experiments, we also found that CIO-aware buffer cache management can help I/O middleware to reduce the actual I/O bandwidth usage, by reducing the data movement between compute and storage nodes.

The rest of this paper is organized as follows. Section II briefly discusses collective I/O and middleware caching as the related work of this study. The design and implementation of collective I/O aware cache management strategy are presented in Section III, and the evaluation methodology and experimental results with analysis are given in Section IV. We conclude this study in Section V.

2. RELATED WORK

Extensive studies have focused on improving the I/O performance of high performance computing systems. We briefly review closely related work with this study along three lines:

parallel I/O and collective I/O, cache management at storage and file systems level, and cache management at middleware and library level.

2.1 Parallel I/O and Collective I/O

There have been significant amount of research efforts in optimizing parallel I/O performance, such as collective I/O [4, 11, 18], data sieving, server-direct I/O, disk-directed I/O, lightweight I/O [15], partitioned collective I/O [21], layout-aware collective I/O [4], ADIOS library [12], and resonant I/O [22]. These strategies collect and aggregate small requests into larger ones at the I/O client/middleware/server level. Abbasi et. al. recently proposed a DataStager framework with data staging services that move output data to dedicated staging or I/O nodes prior to storage, which has been proven effective in reducing the I/O overheads and interferences on compute nodes [1]. Zheng et. al. proposed a preparatory data analytics (PreDatA) approach to preparing and characterizing scientific data when generated (e.g. data reorganization and metadata annotation) to speedup subsequent data access [23]. These approaches have shown considerable performance improvement with dedicated output staging services and preparatory analysis. Advanced I/O libraries, such as Hierarchical Data Format (HDF), Parallel netCDF (PnetCDF) [9], and Adaptable IO System (ADIOS) [12], provide high-level abstractions, map the abstractions onto I/O in one way or another, and complement parallel programming models in managing data access activities.

2.2 Cache Management at Storage and File Systems Level

Numerous prior work focus on improving the behavior of storage (or second-level) cache management because the behavior of the second-level cache is often hard to characterize, making cache management schemes inadequate. Particularly, Zhou et al. investigated multi-queue, eviction-based, and CLOCK replacement policies [24]. Choi et al. proposed a fine-grained file-level characterization of chunk references in buffer management [5]. Vilayannur et al. introduced selective caching because caching of certain blocks is not always beneficial [19]. Sarhan and Das proposed to use the on-disk buffers for caching intervals between successive streams, while multimedia-on-demand servers improve resource sharing by intelligent request schedulers [16]. Our approach complements these existing caching policies with improved cache locality view via revealing the original access pattern that is hidden by collective I/O.

Recently, several studies looked into cache management for multi-level storage hierarchies. The main motivation for these studies is that the modern networked storage systems have a hierarchy of caches, and special care needs to be taken in order to manage those cache hierarchies efficiently. A key idea is how to reduce negative interference while keeping most valuable blocks in shared cache. Techniques to extract and predict the most valuable blocks include transforming application-level requirement into I/O reservations, correlating program counters with program context, exploiting reference regularities, locality of file chunks of non-uniform strength, and automatic application reference pattern detection. For example, Wong and Wilkes explored the exclusive cache policies against the prevalent inclusive ones [20]. These studies are system-level approaches and are therefore orthogonal to our approach. Our approach is also along this direction but is unique because it specifically addresses hidden pattern and locality issues to low level buffer cache management when collective I/O is heavily used in parallel computing systems.

2.3 Cache Management at Middleware and Library Level

Cooperative caching [6] seeks to improve network file system performance by mutually sharing the contents of client data caches. In cluster environments where high performance, low latency message passing networks are frequently available, accessing remote clients to retrieve cached data may result in improved file system throughput. Cooperative caching offers the most opportunity for performance improvements when the client exhibits a large degree of inter-client sharing. Many projects have explored the use of cooperative caching within the file system as an effective means for improving file system performance. The Center for Ultra-Scale Computing and Information Security at Northwestern University has prototyped several file cache designs [2] with ROMIO [17], an open source implementation of the MPI-IO standard. The basic approach involves partitioning the file into a set of fixed size pages. Pages are then assigned to a single computation node by taking the modulo of the page number. Clients processes access file data by requesting it from the client responsible for the cache page rather by accessing the file system, a cooperative caching approach. In one scheme the file data may only be cached at nodes responsible for the cached page. Another scheme implements directories at the responsible node so that another node may cache the page. All of these schemes require that file data is cached at only one node and that all file accesses occur on page aligned boundaries. Our study leverages these existing work, identifies, and addresses the issue of hidden access pattern to low level cache management due to collective I/O.

3. COLLECTIVE I/O AWARE CACHE MANAGEMENT

Each I/O request from applications represents a caching opportunity for the lower level storage systems. In this section, we introduce the proposed collective I/O aware cache management framework to make applications' access pattern available to low level cache. We also present methods for exploiting this knowledge to improve the overall caching performance.

3.1 CIO-Aware Cache Management Framework

Figure 2 illustrates the high-level view of the proposed CIO-aware cache framework. As shown in the figure, the computations of parallel scientific applications are carried out on compute processes, which generate a number of I/O requests for underlying parallel file system. Each parallel process launches a main thread to perform I/O related operations. The caching helper thread is attached to each main thread for cache management. It delivers the original accesses of each parallel process to the pattern detection module.

A pattern detection module is embedded inside the MPI I/O library. It collects and processes the stream of access requests dispatched from caching helper threads. The current

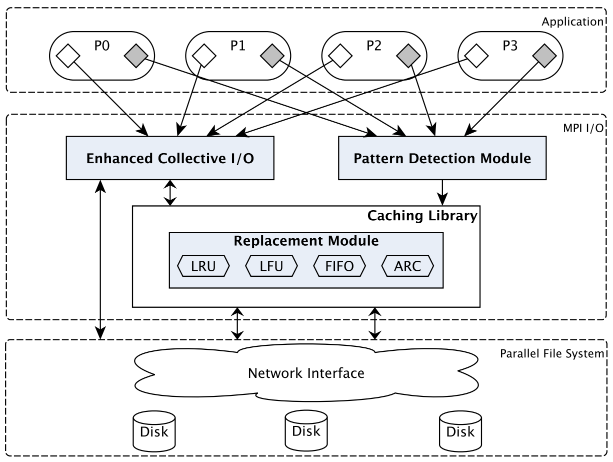


Figure 2: Collective I/O Aware Cache Management

file pointer offsets from caching helper threads are maintained in an implicit file table. The pattern detection module explores the access information from this file table to build pattern views and pass them to the underlying MPI I/O caching library. The pattern detection module also tracks function-call identifiers to synchronize the caching helper thread and the main thread collective I/O calls.

The cache library maintains a global buffer cache among multiple processes. This library is implemented at user space and integrated in the MPI I/O library. It captures the access patterns transferred from the pattern detection module and manages the actual data fetching to the buffer cache. The regular collective I/O is enhanced to take advantages of the cached data residing in the buffer cache. An I/O requesting process must first check the caching status of the requested blocks before exchanging I/O accesses with other processes. If the requested blocks are cached, the requesting process will fetch data from buffer cache directly.

The proposed CIO-aware cache management can work with any cache replacement algorithm. In this study, we focus on four typical replacement algorithms, LRU, LFU, FIFO, and ARC, and study the impact of CIO-aware cache management with any one of these cache replacement algorithms. Such a functionality is implemented in the replacement module.

3.2 MPI I/O Access Pattern Detection

The success of the proposed CIO-aware buffer cache management relies on extracting and utilizing original I/O access information before the collective I/O aggregation. We choose a multi-threading approach to obtain the actual access information of each parallel process. A caching helper thread is constructed in each MPI process when opening the file and destroyed when closing the file. Figure 3 shows several key lines in our prototype to illustrate the thread execution. As shown in Figure 3, the caching helper thread shares certain resources with the main thread, such as process rank, MPI file handles, and file views. While the main thread performs enhanced collective I/O, the caching helper only performs essential computation for data address calculation. A list of offsets and request sizes are created and maintained in a pattern record corresponding to one process.

Main Thread

```
...
MPI_File_open(comm,fname,mode,
info,&mpi_fh);
...
MPI_File_read_all(mpi_fh,buf,
count,dttype,status);
...
MPI_File_close(&mpi_fh);
...
```

Cache Helper Thread

```
...
/*process rank*/
rec->rank=thisrank;
/*file descriptor*/
rec->filedes=mpi_fh->fd_sys;
...
/*individual file pointer*/
rec->file_pos=mpi_fh->fp_ind;
...
```

Figure 3: Collective I/O Aware Access Pattern Detection

One caching helper thread only evaluates how a file is accessed by a local process and transfers the records to the pattern detection module. The pattern detection module receives local patterns from all processes involved in collective I/O operations. It analyzes these local patterns and combines them into a global pattern. The pattern detection module considers the following four factors when producing the global pattern: I/O operation type, spatial locality, temporal pattern, and iterative behavior.

The I/O operation type is classified as read, write, or read-/write. The spatial locality can be contiguous, noncontiguous, and the combinations of contiguous and noncontiguous patterns. When the application conducts one collective I/O operation, each process may access several noncontiguous portions of a file while the requests of multiple processes are often overlapped. These gaps and overlaps can help the caching library identify the potential candidate data blocks to be placed into buffer caches. Capturing temporal patterns is also helpful for organizing the cache blocks. If at one point a particular data block is requested by one process, it is likely that the same block will be requested again in the near future. The replacement module in our proposed caching library manage the cache blocks by using the temporal information obtained from the previous I/O accesses. Scientific parallel programs using MPI I/O usually issue data requests with a few loops. This I/O access pattern can be described as iterative behavior. When repetitive I/O access patterns are captured, identified data blocks can be effectively kept longer in the cache. The cached data can be completely used before evicting them to make room for the new blocks.

Taking the factors mentioned above into account, the global pattern stores information of the file descriptor, process id, I/O operation, time stamp, dimension, starting offset, request sizes and number of repetitions. Consider, as an example, a global pattern value with parameters {[3],READ, 0.023184, 1, [(2622716, 510080), (1573632, 510080)], 64} indicates a one dimensional read access pattern. At time 0.023184, the third MPI process accesses a region whose starting offsets are 2622716 and 1573632 respectively. The request size is 510080 bytes for both accesses. This one dimensional pattern repeats 64 times. Using the pattern value and data block sizes, caching library can identify the set of data blocks captured in the buffer cache.

3.3 MPI IO Caching Library

MPI-IO based data cache can leverage other MPI library components to take advantage of the collective nature of parallel I/O. Incorporating the caching into the MPI library also increases the implementation portability. MPI-IO based caching can easily interface with different underlying file systems. Several research projects have been working on MPI-IO caching libraries. Liao et al. developed a collective caching library implemented at the MPI-IO level[10, 14]. Collective caching maintains a global buffer cache among multiple processes in the client side. We use this library as the starting point for our study. Each client contributes part of its memory to construct the global cache pool. The cached data is transfered among clients through the high-speed interconnect network. Metadata of cached blocks is maintained to locate data quickly. A simplified cache-coherency protocol is used to maintain consistency among cache copies in the cache pool. At most a single copy of file data is allowed to be cached among all MPI processes. Since the read/write mix varies considerably by application domain and read workloads are as prevalent as writes on leadership platforms [3], in this study we customize the collective caching prototype implementation by enabling read caching only. In addition, we utilize a replacement module in conjunction with pattern detection results to direct caching policy. The details will be discussed in the next subsection.

3.4 CIO-Aware Cache Management

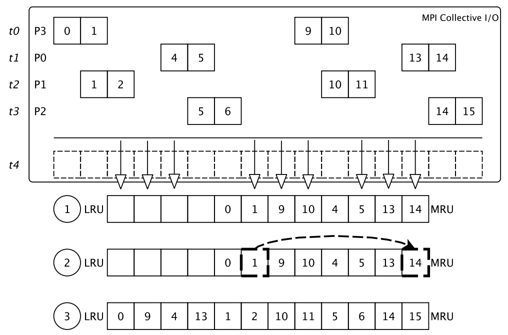
The replacement module in the MPI-IO caching library manages the cache by applying specific replacement policies that best utilize the cache under that access pattern. By taking full benefits of original access patterns delivered from the pattern detection module and used for making the block replacement decisions, caching performance can be enhanced. There has been an extensive research on designing cache replacement algorithms, e.g. LRU [7], LFU, FIFO and ARC [13], etc. In this subsection, we illustrate how cache replacement policies are extended to take advantage of original access pattern from MPI-IO processes.

3.4.1 CIO-Aware LRU

We extend the Least Recently Used (LRU) cache replacement policy and exploit original access temporal locality filtered by collective I/O to manage the LRU list and to decide whether or not to cache accessed blocks.

The new replacement policy of *CIO-aware LRU* first extracts the values of starting offset and request size from each global pattern value. The request is divided into blocks of size equal to the buffer cache block size. We check whether each block is already in the buffer cache or not. If the block is cached, the block is directly copied from buffer cache to user's buffer by using *memcpy()* function call. The exact location where the buffer cache should be copied to is decided by the index of the requested block in user's buffer. Meanwhile, the last access time of this block is updated with its original temporal information and this block is moved to the most-recently-used position. For blocks not placed in the cache buffer, collective reads are first performed directly from the underlying file system. Then these blocks are fetched into the buffer frame held by LRU victims. The general design of CIO-aware LRU is summarized in Algorithm 1.

Figure 4 demonstrates how data blocks are arranged by LRU



MPI Collective I/O
t0 P3 0 1 9 10
t1 P0 4 5 13 14
t2 P1 1 2 10 11
t3 P2 5 6 14 15
t4
1 LRU 0 1 9 10 4 5 13 14 MRU
2 LRU 0 1 9 10 4 5 13 14 MRU
3 LRU 0 9 4 13 1 2 10 11 5 6 14 15 MRU

Figure 4: LRU with Collective I/O Awareness

by exploiting their original access temporal locality filtered by collective I/O. We assume the buffer cache is clean at the beginning with twelve frames/slots. Blocks 0, 1, 9 and 10 are referenced by P_3 at t_0 and blocks 4, 5, 13 and 14 are referenced by P_0 at t_1 respectively. Each data block is copied from the data file on the file system into a buffer in the cache. The LRU list holds all these blocks as shown in the first status. The second status demonstrates block 1 referenced by P_1 at t_2 is moved to the most recently used (MRU) position of the LRU list. Other buffers age toward the LRU position of the LRU list. The third status shows the cache content after all data blocks required through one collective I/O have been organized by LRU with their actual timestamps. By leveraging the virtue of the original access pattern, buffer cache avoid copying extra data blocks which are not requested by processes.

Algorithm 1: CIO-Aware LRU
```
input : A sequence of global pattern values $S_v$ from
        pattern detection module
output: The contents of buffer cache

foreach global pattern value $g_v \in S_v$ do
    split data requests with $g_v$ into blocks $B_s$ ;
    uncatched data blocks set $U_s \leftarrow \emptyset$ ;
    foreach block $b_i \in B_s$ do
        if $b_i \in$ buffer cache then                    // cache hit
            hits++;
            // copy data $b_i$ to user using memcpy()
            user specified buffer $\leftarrow b_i$ in buffer cache;
            // update $b_i$ last access time
            $Last(b_i) \leftarrow b_i$ time stamp;
        else                                             // cache miss
            // perform I/O from disk
            user specified buffer $\leftarrow b_i$ in file system ;
            // evicting the LRU block
            min $\leftarrow$ current time;
            foreach block $b_j \in$ buffer cache do
                if $Last(b_j) < min$ then
                    victim $\leftarrow b_j$ ;
                    min $\leftarrow Last(b_j)$ ;
            if victim == dirty then
                flush the victim to the disk;
            fetch $b_i$ into the buffer frame held by victim;
            $Last(b_i) \leftarrow b_i$ time stamp;
```

To interact with the replacement module and benefit from caching, the current collective I/O implementation is modified to be able to access the buffer cache for requested data. When I/O requests are issued, the replacement module extracts the global pattern values from the pattern detection module. The requests are divided into blocks of size equal to the buffer cache block size. The enhanced collective I/O module first checks whether each block is already in the buffer cache or not. If the block is cached, the block is directly copied from buffer cache to user's buffer by using the *memcpy()* function call as discussed earlier. The general design of cooperation mechanism between enhanced collective I/O and CIO-aware LRU module follows Algorithm 1.

3.4.2 CIO-Aware LFU

A potential problem with LRU is that it may quickly replace some data blocks that do not provide hits for a short period of time, although they are beneficial in the long run. In addition, LRU might also fail when the access pattern is such that all requested data blocks can not fit into the buffer cache and the data blocks are requested in a round robin fashion. What will happen in case of LRU is that data blocks will constantly enter and leave the cache, with no client request ever hitting the cache. Under the same condition however, the Least Frequently Used (LFU) will perform much better, with most of the cached items resulting in a cache hit. In I/O intensive scientific applications, a large amount of overlaps exist among the file regions required by multiple processes. Obviously, the overlapped data are referenced more than other data blocks. Under such a circumstance, we anticipate that LFU can better identify these blocks and they can have higher priorities to stay in cache.

The pseudo code for collective I/O aware LFU is similar to that of Algorithm 1 and thus is not included here. Instead of utilizing temporal information, the algorithm keeps the hit count for each data block in the cache. This is achieved by maintaining two double linked list. One is the access frequency list which is used to link together rectangular hit counters. Each hit counter has a frequency value and connects with a set of circular data blocks that have the same access frequency. All the data blocks with the same access frequency are connected using another doubly linked list.

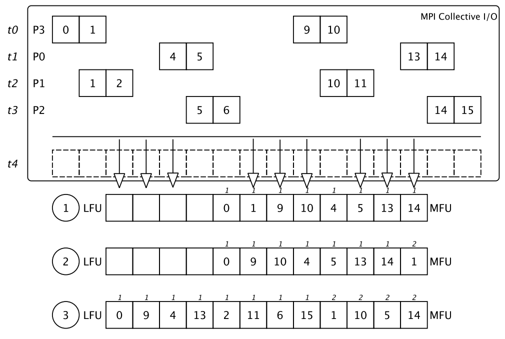


Figure 5: LFU with CIO aware

Figure 5 demonstrates how CIO-aware LFU organizes data blocks using reference frequency hidden by collective I/O. LFU keeps track of the number of times a block is referenced. Status one shows that blocks 0, 1, 9, 10, 4, 5, 13 and 14 are referenced once after t_2. In the second status, block 1 is moved to frequency list with value 2 since it is referenced by P_1 at t_2. The third status illustrates all the data blocks are arranged with their actual reference frequency at t_3. Without revealing the original access pattern that is hidden away by collective I/O, the cache management will not be able to tell the correct request frequency of blocks. For instance, blocks 1, 5, 10, and 14 are all requested twice, whereas after being aggregated by collective I/O, the requested frequency of these blocks becomes once only to the underlying cache.

3.4.3 CIO-Aware ARC

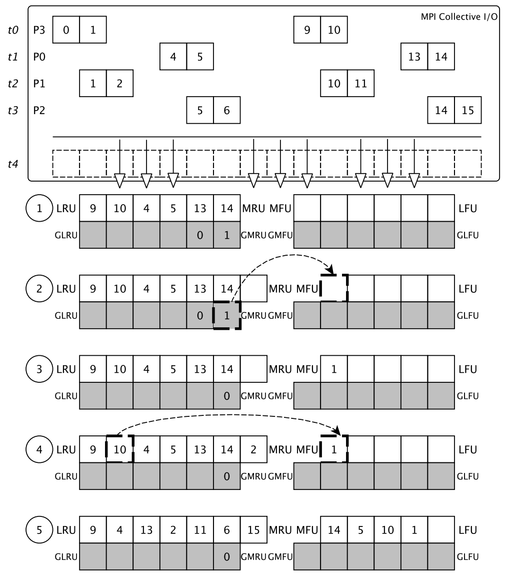


Figure 6: ARC with CIO aware

The standard LFU policy has shortfalls as well. The most significant drawback is that LFU pays no attention to temporal history. Potentially it may accumulate stale pages with high frequency counts that may no longer be useful. Thus LFU does not adapt well to the changing access patterns. Adaptive Replacement Cache (ARC) [13] bridges the gap between LRU and LFU by capturing both recency and frequency. ARC maintains two lists of cache pages, one list for the most recently used pages and another list for the most frequently used pages. In addition, ARC maintains two ghost cache directories that remembers twice as many pages as in the cache memory. Both ghost lists do not cache data, but track the recently evicted pages from the list of most recently used pages and the most frequently used pages respectively. A hit on them affects the behavior of the cache.

The key idea of ARC is to adaptively decide how many top pages from each list to maintain in the cache. Figure 6 shows how CIO-aware ARC performs in response to the same workload we have demonstrated in the previous subsections. GLRU and GLFU represent most recently evicted pages from LRU list and LFU list respectively. When a block is referenced by any processes at first time, it is placed in the recently used list. Status one indicates that blocks 0 and 1 are evicted from the LRU list at t_1 when the LRU list is filled up. These two blocks are put onto the list of recently evicted pages. When block 1 is referenced by P_1 at t_2, this block is on the list of already evicted pages. Such an attempt to read leads to a phantom cache hit. As the block 1 has already been evicted from cache, the system has to read it from un-

Algorithm 2: CIO-Aware ARC

```
input  : A sequence of global pattern values $S_v$ from
         pattern detection module; $T_1$ and $T_2$ hold pages
         metadata in the cache; $B_1$ and $B_2$ are ghost
         caches; $c$ is the cache size; $p$ is a tunable parameter
output: The contents of buffer cache
Set $p = 0$ and Set $T_1$, $T_2$, $B_1$ and $B_2 \leftarrow \emptyset$ ;
foreach global pattern value $g_v \in S_v$ do
    split data requests with $g_v$ into blocks $B_s$ ;
    foreach block $b_i \in B_s$ do
        if $b_i \in T_1$ or $T_2$ then
            Move $b_i$ to top of $T_2$;
        else if $b_i \in B_1$ then
            $p$=min $\left(c, p + \max\left(\frac{|B_2|}{|B_1|}, 1\right)\right)$;
            replace($b_i$, $p$);
            Move $b_i$ to top of $T_2$;
        else if $b_i \in B_2$ then
            $p$=max $\left(0, p - \max\left(\frac{|B_1|}{|B_2|}, 1\right)\right)$;
        else                // page not in $T_1, T_2, B_1, or B_2$
            if $(|T_1| + |B_1|) == CacheSize$ then
                if $T_1 < CacheSize$ then
                    Remove LRU page in $B_1$;
                    replace($b_i$, $p$);
                else
                    Remove LRU page in $T_1$;
            else if $(|T_1| + |B_1| + |T_2| + |B_2|) >= CacheSize$
            then
                if
                $(|T_1| + |B_1| + |T_2| + |B_2|) >= 2 \times CacheSize$
                then
                    Remove LRU page in $B_2$;
                else
                    replace($b_i$, $p$);
            Put $b_i$ at the top of $T_1$;
Replace (page, p)
    if $|T_1| \geqslant 1 \wedge (|T_1| \geqslant p \vee (|T_1| == p \wedge page \in B_2))$ then
        Move LRU page in $T_1$ to top of $B_1$
    else
        Move LRU page in $T_2$ to top of $B_2$
```

derlying file system. Since this was a recently evicted page and not a page referenced just the first time, as shown in status two and three, ARC first places this block in the LFU list. This phantom hit also indicates the capacity of LRU list is not enough. In this case the length of LRU list in cache is increased by one. Obviously this reduces the place for LFU list by one. The same mechanism is applied on the other side. If we get a hit on the list of recently evicted pages of LFU list, the available space for frequently used pages will be increased by one. Obviously the list for currently cached recently used pages will be decreased by one. The status five exhibits the cache contents and the adapted lists' size after collective I/O performed. Algorithm 2 demonstrates a high-level description of CIO-aware ARC.

4. EVALUATIONS

In this section, we present the evaluation results of the CIO-aware cache management prototype tested with a variety of benchmarks. We present results that quantitatively demonstrate the benefits of revealing unseen access pattern in collective I/O and confirm the feasibility of our design.

4.1 Methodology

We quantify the extent to which the CIO-aware cache management scheme improves upon the traditional cache management schemes with respect to two key metrics: I/O throughput and buffer cache hit rate. The I/O throughput is expressed as the ratio of the total number of bytes transferred to/from file system to the time required to transfer data. A higher I/O throughput can lead to better performance of the application, i.e. less application execution time. We also choose the cache hit rate metric because it has a direct impact on application execution time. The buffer cache hit rate is defined as the ratio of the total number of buffer cache hits to the total number of I/O accesses made by the application.

The experiments were conducted on a 640-node Linux-based cluster test bed with DataDirect Network storage systems. Each node contains two Intel Xeon 2.8 GHz 6-core processors with 24 GB main memory. All nodes are connected with double-data-rate Infiniband networking that provides full cross-section bandwidth among the parallel nodes. A 600TB Lustre file system and MPICH-3.0.2 library manage the storage system and runtime environment. Files were striped over all I/O servers with the round robin default striping strategy (with 1 MB unit size in the experiments).

In the following experiments, we compare the CIO-aware cache management and the baseline scenario, in which client applications are configured to access the shared file system directly via MPI-IO layer. In addition to the experimental setup described above, we have also built a trace driven buffer cache simulator to measure the cache hit rates. In order to compare the hit rate of our strategy with the hit rate of a traditional system, the trace collector captures the traces of our applications twice while they ran on the cluster. The I/O operation parameters are collected once by using the Profiling MPI interface (PMPI) before the actual collective I/O function is issued and second time in ADIO layer after I/O requests are aggregated.

4.2 MPI-Tile-IO Benchmark

MPI-Tile-IO is a widely used benchmark designed to test the performance of non-contiguous data access. In this application, data I/O access is issued in a single step by using collective I/O. It tests the performance of concurrently accessing a two-dimensional dense data set, simulating the type of workload that exists in visualization and numerical applications. In our experiments, each process renders one tile with 1024×1024 pixels and the size of each element is 8 bytes. Tiles overlap by 128 elements in X axis, 128 elements in Y axis. Because this benchmark closes the file between write and read operations, we slightly modified the benchmark to avoid close/re-open in order to show the effect of collective buffer cache.

Figure 7 compares the total execution time for the implementation with CIO-aware strategy and the native approach which is oblivious of the application access pattern. The experimental results were measured with 8, 16, 32, 64, 128, 256, 512, and 1024 processes on Lustre respectively. The total data are set as 10GB in each I/O phase. The buffer cache size at each client was set as 64MB. We measure the latency for each processesâĂŹ number by 10-time runs and plot the figure with the median value. In the Figure 7, the

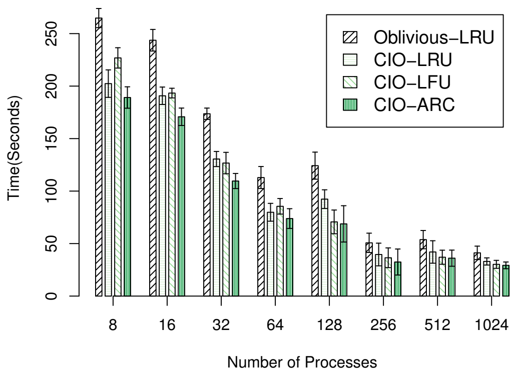


Figure 7: Total execution time comparison of Mpi-Tile-IO, each run with 10 GB data and 64 MB buffer cache per process

first bar of every column represents the original execution time. The second, third and fourth bar represent the execution time with CIO-aware strategy under LRU, LFU and ARC replacement schemes.

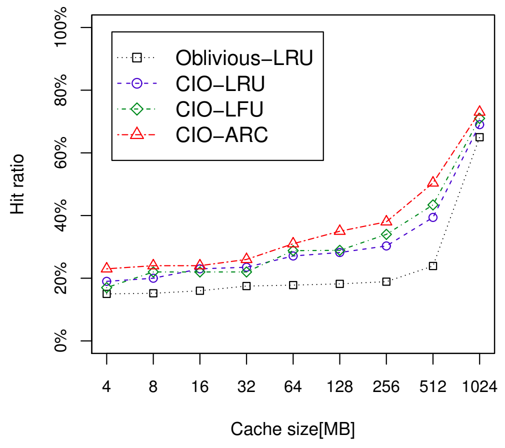


Figure 8: Hit rate comparison of cache replacement schemes for Mpi-Tile-IO, each run with 32 processes

The proposed CIO-aware cache management is on top of existing collective buffer optimization technique and complements the existing approaches. We observe that CIO-aware approach can reduce I/O access latency further when combined with existing collective buffering techniques. Additionally, the improvement increases with the number of processes. The decrease in read execution time was up to 80.6%. Overall, the average execution time decrement was 31.1%, 38.5% and 52.6% in CIO-LRU, CIO-LFU and CIO-ARC schemes respectively. These performance improvements are attributed to two causes: the improvement of buffer cache organizations and the reduction of underlying file system accesses. With CIO-aware buffer cache, subsequent collective read operations takes better advantage of cache capacity than the native approach. These results also indicate that the choice of a good replacement policy is crucial to CIO-aware scheme. It can be observed that the CIO-aware approach with ARC policy outperforms the LRU and LFU based schemes by 16.4% and 10.7%, respectively.

Figure 8 shows the hit ratios of different cache replacement algorithms. The original collective I/O trace managed by LRU was selected as the baseline. Compared to the original approach, the ARC with CIO-aware strategy provided the best hit rate among all the algorithms. It improved the hit rate by as much as 110.8% with an average of 66.7% improvement over the nine cache sizes. With limited cache size, CIO-aware LRU provides a relatively high hit rate with small cache size. Each block in an LRU cache has a long life before it is discarded, and thus has a high possibility to be referenced again by different clients with high-correlated workloads. The gain becomes smaller as the buffer cache is larger, since a large cache size retains a block for a long enough time, within which it is accessed by most clients.

4.3 IOR benchmark

The Interleaved Or Random (IOR) benchmark measures the performance of parallel I/O through different I/O interfaces, including MPI-IO, POSIX as well as high-level libraries. In this study, we performed interleaved read operations to a file as we varied the number of processes for collective I/O. The tests were carried out with 8MB I/O message size per process.

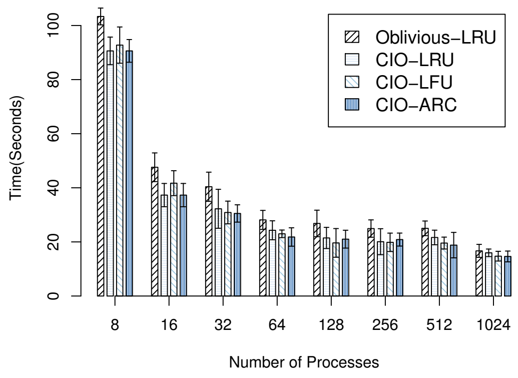


Figure 9: Total execution time comparison of IOR, each run with 10 GB data and 64 MB buffer cache per process

The total execution time results with IOR benchmark are plotted in Figure 9. From Figure 9, clear improvements of CIO-aware cache management over the original strategy can be observed. At 64MB cache size per process with CIO-aware strategy, the execution for CIO-LRU, CIO-LFU and CIO-ARC was decreased by 19%, 22.8% and 24.6%, respectively. Figure 10 shows the hit ratios for IOR benchmark under different cache sizes. Our results with 32 processes and CIO-aware cache showed that the percentage improvements brought by our scheme over the original LRU replacement are 54.4%, 50.8%, 75.5% for CIO-LRU, CIO-LFU and CIO-ARC, respectively.

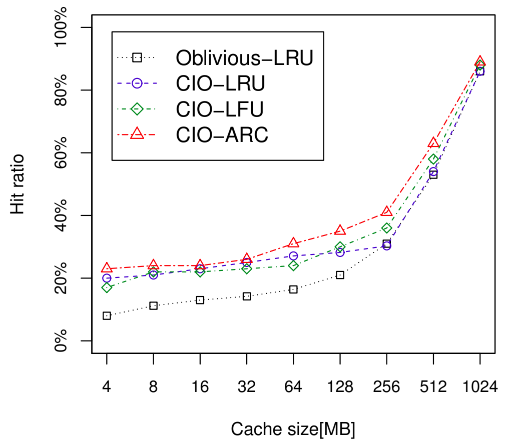


Figure 10: Hit rate comparison of cache replacement schemes for IOR, each run with 32 processes

Both sets of experiments have verified the proposed collective I/O aware cache management achieved considerable hit rate increments and sustained bandwidth improvements.

4.4 MPI-IO-Test Benchmark

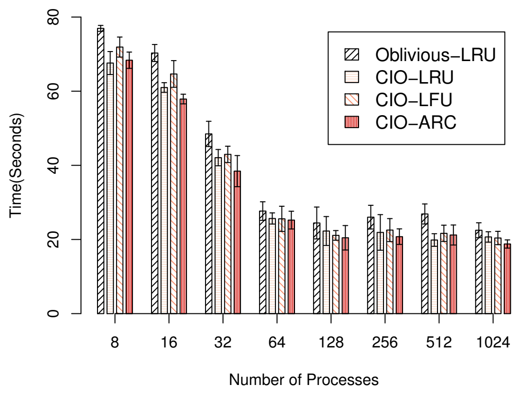


Figure 11: Total execution time comparison of MPI-IO-Test, each run with 10 GB data and 64 MB buffer cache per process

We have carried out various tests with the mpi-io-test benchmark as well. Figure 11 compares the execution time with normal collective I/O and CIO-aware strategy at 64 MB buffer cache per process with a varying number of processes at each run. Compared to the normal collective I/O, the average performance improvements for read is 15.6%, 12.8% and 20.1% for CIO-LRU, CIO-LFU and CIO-ARC. We can observe that the caching improves the performance, but the improvement is not very substantial. One reason is that read caching can perform well if large amount of data reuse exists. If there is no much data reuse, the read caching may not perform as well as expected. Figure 12 reports another set of test where we used 32 processes for evaluating the cache hit

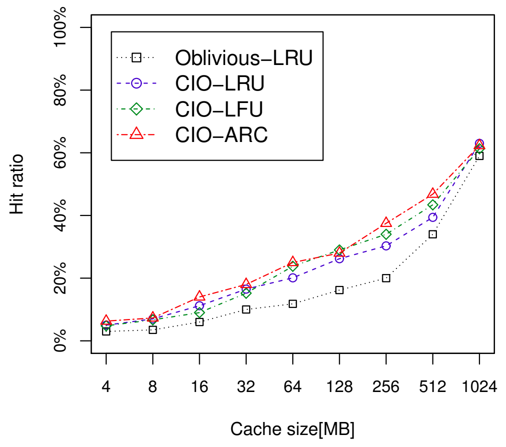


Figure 12: Hit rate comparison of cache replacement schemes for MPI-IO-Test, each run with 32 processes

rate with increasing the buffer cache size. The average hit rate improvement was 58.2%, 59.5%, 83% respectively for CIO-LRU, CIO-LFU and CIO-ARC in this series of tests. Compared with the IOR benchmark, the mpi-io-test cache hit rate increased but at a relatively moderate rate with an increasing cache size.

All these results indicate that the CIO-aware cache management is beneficial to achieve better throughput for collective I/O operations and higher hit rate for the underlying buffer cache.

5. CONCLUSION

Parallel I/O systems have become increasingly critical due to many growingly data-intensive high-performance computing simulations and applications. These data-intensive scientific simulations and applications rely on a highly efficient parallel I/O system to make productive scientific discovery. In current parallel I/O software stack, collective I/O has been widely recognized as a critical I/O strategy that leverages correlations among parallel processes to carry out parallel I/O requests more efficiently. The current collective I/O and parallel I/O software stack, however, are not well integrated and suffer dropping out useful access patterns from applications during the aggregation process of the collective I/O strategy.

In this study, we have thoroughly demonstrated and analyzed this issue. We have shown that the collective I/O filters away many useful I/O access patterns that can be critical to underlying cache management.These thrown-away access patterns can have a negative impact on cache management algorithms on their views of locality, which leads to many unnecessary cache misses in low level buffer caches and additional disk accesses. We have thus proposed a new collective I/O aware cache management methodology that reveals unseen access pattern to underlying caching algorithms. We have prototyped such an idea and the new methodology. We have also carried out evaluations with widely-used cache management algorithms based on recency and frequency of access patterns. The evaluations have confirmed the performance advantage of revealing unseen access patterns in collective I/O. We believe that the issue identified in this study and the new methodology proposed can be helpful and can provide guidance for the community to build an even more efficient parallel I/O system.

6. ACKNOWLEDGMENT

This research is sponsored in part by the National Science Foundation under grant CNS-1162488 and supported by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357. The authors thank the High Performance Computing Center (HPCC) at Texas Tech University at Lubbock for providing HPC resources that have contributed to the research results reported in this paper.

7. REFERENCES

[1] H. Abbasi, M. Wolf, and G. e. Eisenhauer. DataStager: Scalable Data Staging Services for Petascale Applications. In *HPDC*, 2009.

[2] S. Byna, Y. Chen, X.-H. Sun, R. Thakur, and W. Gropp. Parallel I/O Prefetching Using MPI File Caching and I/O Signatures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.

[3] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross. Understanding and Improving Computational Science Storage Access through Continuous Characterization. In *Mass Storage Systems and Technologies (MSST), 2011 IEEE 27th Symposium on*, 2011.

[4] Y. Chen, X.-H. Sun, R. Thakur, P. Roth, and W. Gropp. LACIO: A New Collective I/O Strategy for Parallel I/O Systems. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 794 –804, 2011.

[5] J. Choi, S. H. Noh, S. L. Min, and Y. Cho. Towards Application/File-level Characterization of Block References: A Case for Fine-Grained Buffer Management, 2000.

[6] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proceedings of the first Symposium on Operating Systems Design and Implmentation*, 1994.

[7] A. Dan and D. Towsley. An Approximate Analysis of the LRU and FIFO Buffer Replacement Schemes. In *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, SIGMETRICS '90, New York, NY, USA, 1990. ACM.

[8] J. Dongarra, P. H. Beckman, and T. M. etc. The International Exascale Software Project roadmap. *IJHPCA*, 25(1):3–60, 2011.

[9] J. Li, W. keng Liao, A. Choudhary, R. Ross, R. Thakur, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netCDF: A high-performance Scientific I/O Interface. In *In Proceedings of Supercomputing*, 2003.

[10] W.-K. Liao, A. Ching, K. Coloma, A. Choudhary, and L. Ward. An Implementation and Evaluation of Client-Side File Caching for MPI-IO. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, 2007.

[11] W.-k. Liao and A. Choudhary. Dynamically Adapting File Domain Partitioning Methods for Collective I/O based on Underlying Parallel File System Locking Protocols. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, SC'08, 2008.

[12] J. F. Lofstead, S. Klasky, and K. e. Schwan. Flexible IO and Integration for Scientific Codes through the Adaptable IO System (ADIOS). In *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, 2008.

[13] N. Megiddo and D. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of the 2003 Conference on File and Storage Technologies (FAST)*, pages 115–130, 2003.

[14] A. Nisar, W.-k. Liao, and A. Choudhary. Scaling Parallel I/O Performance through I/O Delegate and Caching System. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 9:1–9:12, 2008.

[15] R. Oldfield, L. Ward, R. Riesen, A. B. Maccabe, P. Widener, and T. Kordenbrock. Lightweight I/O for Scientific Applications. In *CLUSTER*, 2006.

[16] N. J. Sarhan and C. R. Das. Caching and Scheduling in NAD-Based Multimedia Servers. *IEEE Trans. Parallel Distrib. Syst.*, 15(10):921–933, Oct. 2004.

[17] R. Thakur, W. Gropp, and E. Lusk. *Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation.* Mathematics and Computer Science Division, Argonne National Laboratory, Oct. 1997.

[18] R. Thakur, W. Gropp, and E. Lusk. Data Sieving and Collective I/O in ROMIO. In *Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*, FRONTIERS '99, 1999.

[19] M. Vilayannur, A. Sivasubramaniam, M. Kandemir, R. Thakur, and R. Ross. Selective Caching for Parallel File Systems on Clusters. *Special Issue on Parallel I/O in Computational Grids and Cluster Computing Systems*, Jan 2006.

[20] T. M. Wong and J. Wilkes. My Cache or Yours? Making Storage More Exclusive. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, 2002.

[21] W. Yu and J. Vetter. ParColl: Partitioned Collective I/O on the Cray XT. In *Proceedings of the 2008 37th International Conference on Parallel Processing*, ICPP '08, 2008.

[22] X. Zhang, S. Jiang, and K. Davis. Making Resonance a Common Case: A high-performance Implementation of Collective I/O on Parallel File Systems. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, 2009.

[23] F. Zheng, H. Abbasi, C. Docan, J. F. Lofstead, Q. Liu, S. Klasky, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf. PreDatA - Preparatory data Analytics on Peta-scale Machines. In *IPDPS'10*, pages 1–12, 2010.

[24] Y. Zhou, J. F. Philbin, and K. Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proceedings of the 2001 USENIX Annual Technical Conference*, pages 91–104, 2001.