Parallel I/O Prefetching Using MPI File Caching and I/O Signatures

Surendra Byna, Yong Chen, Xian-He Sun
Department of Computer Science
Illinois Institute of Technology
Chicago, IL, USA
{sbyna, chenyon1, sun}@iit.edu

Rajeev Thakur
Mathematics & Computer Science Division
Argonne National Laboratory
Argonne, IL, USA
thakur@mcs.anl.gov

William Gropp
Computer Science Department
University of Illinois Urbana-Champaign
Urbana, IL, USA
wgropp@illinois.edu

Abstract — **Parallel I/O prefetching is considered to be effective in improving I/O performance. However, the effectiveness depends on determining patterns among future I/O accesses swiftly and fetching data in time, which is difficult to achieve in general. In this study, we propose an I/O signature-based prefetching strategy. The idea is to use a predetermined I/O signature of an application to guide prefetching. To put this idea to work, we first derived a classification of patterns and introduced a simple and effective signature notation to represent patterns. We then developed a toolkit to trace and generate I/O signatures automatically. Finally, we designed and implemented a thread-based client-side collective prefetching cache layer for MPI-IO library to support prefetching. A prefetching thread reads I/O signatures of an application and adjusts them by observing I/O accesses at runtime. Experimental results show that the proposed prefetching method improves I/O performance significantly for applications with complex patterns.**

Keywords – parallel I/O; prefetching; I/O signatures; MPI-IO

I. INTRODUCTION

Many scientific and engineering simulations in critical areas of research, such as nanotechnology, astrophysics, climate, and high energy physics, are highly data intensive. These applications contain a large number of I/O accesses, where large amounts of data are stored to and retrieved from disks. However, disparity of technology growth is causing a gap between processor performance and storage performance that has been increasing over the last few decades. This poor I/O performance has been attributed as the cause of low sustained performance of existing supercomputers. Although advanced parallel file systems (such as PVFS [7], Lustre [4], GPFS [32]) have been developed in recent years, they provide high bandwidth only for large, well-formed data streams, but perform poorly in dealing with large number of small and noncontiguous data requests. While techniques such as collective I/O and data sieving [37] can be used to merge small I/O requests into large ones, many small I/O requests cannot be eliminated due to the inherent nature of the underlying applications. The so-called *I/O wall* is thus a critical performance bottleneck.

I/O prefetching is a promising technique to improve file access performance of many applications [13][20][25][28]. The effectiveness of prefetching depends on determining future I/O accesses of an application and fetching data closer to it. I/O accesses follow regular patterns in many applications, where data is accessed regularly for processing, and the processed data is stored back in files [12][17][18][24]. In these applications, regular patterns of I/O accesses can be identified via post-analysis. We propose a prefetching method with a combination of post-analysis and runtime analysis of I/O accesses. The idea behind our method is to detect the pattern of I/O accesses of an application, store the pattern information as a signature representation, and use that signature in the future runs of the application. To develop the signature notation of I/O accesses, we present a classification of I/O access patterns based on a study of a collection of widely used parallel benchmarks. We developed a tracing library to collect information of an application's MPI-IO calls. We analyzed these traces to generate a representation of an I/O access pattern, called *I/O signature*. Conventional prefetching methods are fast, but simple, and only detect simple strided patterns. The reason is that complex data access patterns are difficult to identify and the pattern search methods take a longer time to learn. The signature-based approach reduces the runtime processing cost and makes in-time prefetching possible. In our design, a prefetching thread runs alongside the main computing thread during I/O operations to predict data requirements of the main thread and to bring that data into a prefetch cache that is closer to the application. This prefetch thread uses both the I/O signature and the information of I/O accesses during runtime to predict future I/O accesses. The prefetching thread does not perform any computing but is solely responsible for prefetching data into the prefetch cache. To store the prefetched data, we introduce a client-side prefetch cache for parallel applications. We have modified the MPI-IO library of MPICH2 to add the client-side cache to support prefetching.

This research was supported in part by National Science Foundation under NSF grant EIA-0224377, CNS-0406328, CNS0509118, and CCF-0621435.

SC2008 November 2008, Austin, Texas, USA 978-1-4244-2835-9/08 $25.00 ©2008 IEEE

The paper is organized as follows. Section II discusses related work in I/O prefetching and access pattern analysis. We present the overall I/O signature-based prefetching method in Section III. A classification of I/O access patterns, I/O signature representation of access patterns are presented in Section IV. The design and implementation of trace collection and analysis, the client-side prefetch cache, and prefetching method in the MPI-IO library are presented in Section V. I/O signatures of various benchmarks are presented in Section VI, and performance results with our prefetch strategy are given in Section VII. We conclude the paper in Section VIII with a discussion of our observations and future work.

II. Related Work

A. I/O Prefetching

Many I/O prefetching approaches have been proposed including Chang and Gibson's SpecHint [5][6], Patterson and Gibson's Informed Prefetching (TIP) [25], and Yang's AAFSP [41]. SpecHint and TIP use idle cycles to speculate future I/O accesses. The AAFSP technique is lightweight, but is only designed for sequential applications. Our client-side prefetching cache technique is targeted for parallel applications that have I/O access patterns with some degree of regularity. Static information about the I/O pattern is captured during post-execution analysis. A prefetching thread then adjusts the pattern information based on dynamic observation of I/O accesses. PPFS2 [39] offers runtime optimization for caching, prefetching, data distribution, and sharing. In our approach, we use a combination of trace analysis from previous runs and runtime adjustment of the I/O signature. This reduces the overhead of detecting patterns at runtime, especially for regular but complex patterns, which require more time in detecting patterns.

B. I/O Access Patterns

Several previous parallel I/O studies observed that the I/O accesses of many applications follow certain patterns. Miller and Katz [18] studied several applications running on a Cray Y-MP vector computer and detected that these applications access data in chunks ranging from 32 KB to 512 KB. They concluded that I/O access sizes of the tested applications were relatively constant, cyclic, bursty, and predictable. Keeton et al. [12] observed small and large jumps (stride or difference between file offsets) among sequential accesses and interferences between concurrent accesses. Pasquale et al. [26][27] observed similar regularity among I/O patterns on a Cray C90. Studies of I/O accesses on distributed memory systems such as CM-5, iPSC/860, and the Intel Paragon XP/S [3][14][29] show that many I/O requests are small and have irregular patterns. Crandall et al. [3], Madhyastha et al. [17], and Smirni et al. [35] studied scalable I/O applications and provided a classification of patterns based on three dimensions of file access features: Type of I/O operation (read/write), sequentiality, and size of I/O requests. In this study, we classify access patterns further in the dimensions of repetitiveness and temporal behavior. Using these dimensions, we provide a representative notation for I/O accesses. Marathe et al. [19] used a notation to represent memory access traces that represents the length of a memory access and sequentiality. Our pattern representation notation is for I/O accesses and covers more information in multiple dimensions. Each of these dimensions can contain complex nested pattern notation. This notation not only reduces the size of traces, but is also useful in selecting prefetching strategies.

C. Client-side Caching

Many attempts have been made to use caching at the MPI-IO level, including collective caching [15] and active buffering [16]. Application-aware collective caching at the MPI-IO level [15] is an effective solution in caching data that is used frequently. This can be used for both reads and writes. We use the collective cache approach in our prefetching cache design. Our prefetching cache is a complement to the caching approach and further improves I/O read performance.

III. I/O Signature-based Prefetching

I/O signature-based prefetching is a two step process. In the first step, traces of a running application are collected and analyzed to detect any patterns among them. The detected patterns are stored as an I/O signature. An I/O signature contains information regarding the strides between successive I/O accesses (spatial pattern), how many times the pattern is repeated, its temporal pattern, the size of requested data, etc. We give more details about the I/O signature in Section IV. The second step involves prefetching at runtime, where signatures of an application are read, verified, and used to prefetch data when a stable pattern is found.

Figure 1 illustrates the prefetching thread operations of one

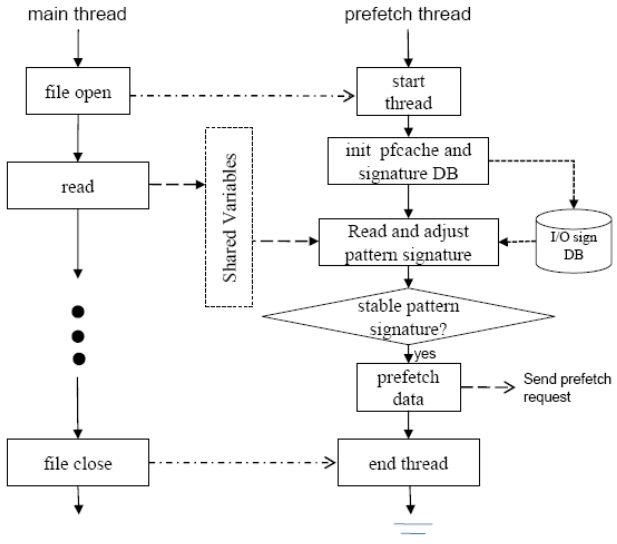

Figure 1. I/O thread and prefetching thread operations from a single MPI process view

client node. Our design follows the design of client-side collective caching [15], which was developed to cache frequently used pages. Since we designed our prefetching strategy to work within an MPI-IO library, our prefetch caching also uses the MPI communicator of an MPI program to detect the scope of processes that collaborate with each other in performing prefetching. We intend to preserve MPI-IO optimizations such as data sieving and collective I/O [37]. Therefore, we integrate our prefetching design in the ADIO (Abstract Device Interface for I/O) layer used in the ROMIO implementation of MPI-IO. As shown in Figure 1, a prefetching thread starts for each MPI process when the first file is opened and ends when the last file is closed. The prefetching thread initiates a user level prefetching cache and reads I/O signatures from a file (I/O signature DB). The parameters of the I/O signatures are adjusted dynamically by observing the I/O accesses of the MPI process to which the prefetching thread is attached. The main thread communicates I/O access information to its prefetching thread using shared variables. When the prefetching thread finds a stable pattern, it starts prefetching data into the prefetching cache, which the main thread looks up before sending a request to its underlying file system. If data is found in the prefetch cache, the main thread accesses that data. Otherwise, its normal operation of sending an I/O request to file system is performed.

IV. I/O Access Patterns

I/O accesses of parallel applications can be divided into *local patterns* and *global patterns*. Local patterns are determined per process (or thread), showing how a file is accessed by a local process. Global patterns are over the parallel application, representing how multiple processes access a file. For example, a file can be accessed sequentially by all processes of an application, where every process reads a chunk of data. Many parallel file systems, including PVFS [7], Lustre [4], GPFS [32], provide optimizations to stripe files on disks efficiently to improve the locality of global patterns. In this study, we focus on local patterns of each process.

Classifying I/O accesses into a set of patterns gives us an opportunity to tune performance. Prefetching strategies can utilize these patterns to predict future I/O accesses. I/O accesses can be reordered to improve cache reuse by having the information of access patterns. For instance, in out-of-core applications, data processing operations can be performed on all the data that has been fetched into memory before it is swapped to disk. Access pattern information is also helpful in selecting efficient cache replacement strategies.

In previous work, we classified the *memory* access patterns of an application based on distances between successive accesses and request size [1]. We extend that classification for I/O accesses. Based on our study of various I/O benchmarks that represent real parallel applications, we introduce a five-dimensional classification for access patterns of a local process. The five dimensions are spatiality, request size, repetitive behavior, temporal intervals, and type of I/O operation, as shown in Figure 2. The sequence of file locations accessed represents the spatial pattern of an application. They can be contiguous or non-contiguous or a combination of both. Non-contiguous accesses refer to gaps (or strides between successive file offsets) in accessing a file. These gaps can be of fixed size or variable size. Variable size gaps can follow a pattern of two or more dimensions (*2d* or *kd*). Another possible pattern is one with decreasing (negative) strides. Some I/O accesses have no regular pattern, where the strides are random.

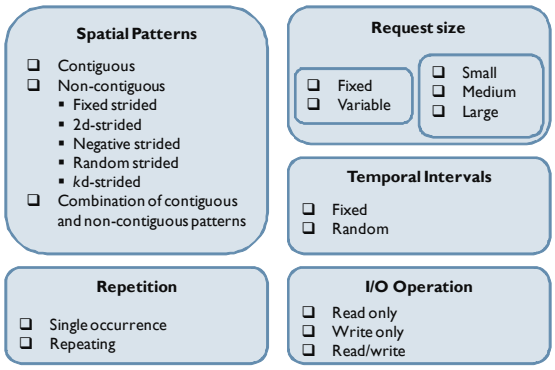

Figure 2. I/O access pattern classification

Applications exhibit repetitive behavior when a loop or a function with loops issues I/O requests. We classify I/O access patterns either with repetitive behavior or without (i.e., pattern occurs only once). When I/O access patterns are repetitive, caching and prefetching can effectively mask their access latency. By capturing repetitive behavior, cached data can be kept longer or accesses can be reordered in a way that the fetched data is completely used before replacing it by new pages of data. Prefetching can utilize this repetitive behavior by storing previous pattern information and reuse that information to calculate future I/O access offsets without running prediction routines to search for the same pattern multiple times. We use this approach in our I/O signature-based strategy.

Request sizes can be small, medium, or large. The sizes of requests can be either fixed or varying. We characterize a request as a small request when it is only a fraction of a page size, and as a large request when it is multiple times larger than a page size. Because of the high I/O latency, small I/O requests commonly cause performance bottlenecks if disks must be accessed multiple times for a small number of bytes. If possible, multiple small accesses can be combined into a larger contiguous request to reduce the number of disk seeks.

Temporal patterns capture regularity in I/O bursts of an application. They can occur either periodically (at fixed intervals) or irregularly. Capturing temporal regularity can be used in prefetching strategies to initiate prefetch requests *in time*, so that prefetched data reaches its destination cache neither too early nor too late.

The type of I/O operation is the last criteria for pattern classification. We classify the operations as read, write, or read/write.

Compared to the I/O access pattern classification provided by previous characterization studies [3][35][36], we add two more dimensions: *repetition* and *temporal intervals*. In the spatial pattern dimension, we classify existing *variably strided* pattern further into various non-contiguous patterns. This further classification is useful in selecting prediction methods of future I/O accesses by a prefetching strategy, reordering I/O accesses, and choosing victim pages by replacement policies.

A. I/O Signature Notation

We have developed a set of notations to describe I/O access patterns, which we call the *I/O signature* of access patterns for an application. The I/O signature can be given in two forms; the first describing the sequence of I/O accesses in a pattern and the second identifying I/O patterns. We call the description of a sequence of I/O accesses in a pattern a *Trace Signature*, and the abstraction of a pattern a *Pattern Signature*. Using the five dimensions mentioned above, *trace signature* takes the form as follows:

{I/O operation, initial position, dimension, ([{offset pattern}, {request size pattern}, {pattern of number of repetitions}, {temporal pattern}], […]), # of repetitions}

It stores information of an I/O operation, starting offset, depth of a spatial pattern, temporal pattern, request sizes, and repetitive behavior. In some instances, offsets, request sizes, timing, and number of repetitions also contain a pattern. Random temporal patterns are not captured in the trace signature, as the usage of randomness is limited.

To illustrate I/O signature with an example, we use I/O traces of out-of-core LU decomposition [11][40]. This application accesses an 8192 x 8192 matrix of double-precision elements. Table 1 shows the trace information of the first few I/O reads. It has a 3-dimensional stride access pattern. We use *italicized,* **bold face**, and normal fonts to represent three dimensions of patterns. The *initial position* of read accesses is 1049088. The first dimension of offsets (*italicized* in Table 1) has a pattern {1049088, 1, (524544, 1)}, where initial offset is 1049088. The next offset is 1573632 (i.e. 1049088+ 524544). This stride pattern has only one access in each iteration, and the request size is 524544 bytes. The second dimension of offsets (**bold faced**) start at 524544. In this dimension, the sizes of requests also follow a pattern where each request size is 4096 bytes less than the previous request. There is no temporal pattern due to lack of regularity among I/O read times. The number of repetitions also has a pattern, where the number of repetitions in an iteration is one more than that of the previous iteration. The third dimension of the offsets starts at the start of the file (0), and accesses 522368 bytes of data. This 3-dimensional pattern repeats 125 times. The overall *trace signature* is shown below:

{READ, 0, 3, (*[{1049088, 1, [{(524544, 1)}, 524544], 1}]*,
[{524544, 1, [{524544}, {518272, 1, (-4096)}, {1, (1)}]},
[{0, 1, [(0, 522368, 1])}]), 125}

While a *trace signature* provides a way to reconstruct the sequence of I/O accesses, a *pattern signature* provides an abstract description that explains the nature of a pattern. A *pattern signature* takes the following form:

TABLE 1. TRACES OF LU DECOMPOSITION

Offset	Request size
1049088	*524544*
524544	**518272**
0	522368
1573632	*524544*
524544	**518272**
1049088	**514176**
0	522368
2098176	*524544*
524544	**518272**
1049088	**514176**
1573632	**510080**
0	522368
2622720	*524544*
524544	**518272**
1049088	**514176**
1573632	**510080**
2098176	**505984**
0	522368
3147264	*524544*
524544	**518272**
1049088	**514176**
1573632	**510080**
2098176	**505984**
2622720	**501888**
0	522368

{I/O operation, <Spatial pattern, Dimension>, <Repetitive behavior>, <Request size>, <Temporal Intervals>}

In a pattern signature, we store information consisting of all five factors of our classification. Here random temporal patterns are represented. An example of pattern signature for the out-of-core LU traces is shown below:

{READ, <Non-contiguous, 3d strided>, <repetitive>, <large, variable size>, <random>}

The use of an I/O signature has multiple advantages. An I/O signature provides an automatic and systematic way of describing the I/O property of an application. I/O access optimization methods work effectively for specific patterns. Having an I/O signature that can be recognized automatically gives an opportunity to select efficient caching and prefetching optimization strategies based on patterns. This notation is useful in performing cost-benefit analysis of prefetching to decide whether to use prefetching. If the I/O signature of an application indicates complex access patterns that require more computing overhead to detect (the prediction cost is high), it is advisable to avoid prefetching in that case. The trace signature of an I/O signature reduces the size of trace files when regular patterns exist. Instead of having traces with regular patterns repeating throughout trace files, our representation can compress those repetitions into an I/O signature.

A significant advantage of an I/O signature, which has been the motivation behind this research, is its usage in predicting future I/O accesses and prefetching that data. Post-execution analysis of an application's I/O accesses can be stored as a

signature, which can be used by a prefetching strategy as hints for generating prefetching requests. Let us take an example where a process is accessing I/O in a strided pattern. Assume a fixed stride of 8192 bytes and each I/O read request is of 2048 bytes. The trace signature of that pattern is *{READ, initial position, 1, ([8192, 4096, 100]), 1}*. Prefetching strategies can use this trace signature to calculate future file offsets from *initial position*, i.e. (0 * 8192, 1 * 8192, 2 * 8192,… , 99 * 8192).

A limitation of the trace signature is its usage for representing random accesses. While usage of trace signature for regular patterns reduces the size of a trace file, there is no such benefit for representing random I/O accesses. For random accesses, the storage space required for a trace signature representation and a trace file are of the same order. In such cases, instead of having a large trace signature, providing a pattern signature can itself express the information that these I/O accesses are random. This can guide history-based prefetching strategies to avoid trying to predict future offsets, as it is difficult to predict random I/O accesses from the history of accesses.

Another limitation in using an I/O signature, which is obvious to any post-execution analysis of traces, is the variation of values of its elements when an application is rerun or when the stride pattern changes with the input values. While the *pattern signature* for most applications does not change with multiple runs and with different inputs, the *trace signature* must be modified in this situation. For example, the *initial position* or *request size* can vary based on the number of processes. As shown in the fixed stride I/O signature above, *initial position* can be process dependent, where a process *i* may start accessing data from *i*partition*, where *partition* is a function of process rank. In these cases, the I/O signature can be adjusted during runtime analysis, which is the second step of our signature-based prefetching method. In most cases, using a *pattern signature* itself is sufficient for adaptively selecting a future I/O access prediction algorithm of a prefetching strategy.

V. Implementation of Signature-based Prefetching

Implementation of I/O signature-based prefetching contains many challenges including detecting patterns in history of I/O accesses and generating signatures with post-execution analysis, implementing the prefetch cache, collecting runtime information to adjust signatures that are generated offline, developing a software library to issue prefetch requests, synchronizing main and prefetching threads in fetching data, and maintaining coherence of prefetched data when it is updated by a process. We discuss solutions to each of these issues in the following subsections.

A. Trace Collection and Analysis

Characterizing I/O workloads typically starts with tracing application I/O requests. Each trace record contains the information of *the type of request (read, write, seek, etc.), initial position, size of request, time stamp, client process that is accessing the file, file offset, and file which is being accessed*. Madhyastha et al. have used Pablo [31] to trace I/O calls and to store traces in Self-Defining Data Format (SDDF). As our focus is entirely on studying MPI-IO calls, we developed a tracing tool that traces all MPI-IO read and write related calls. To capture MPI-IO calls, we use the profiling interface of MPI [20], which provides a convenient way for us to insert our code in an implementation-independent fashion. In MPI implementations, every function is available under two names, MPI_ and PMPI_. User programs use the MPI_ version of the function, for example, MPI_File_read. We intercept the user's call to MPI_File_read by implementing our own MPI_File_read function in which we retrieve information required for tracing and then call PMPI_File_read to do actual file read. As a result, application programs do not need to be recompiled; they need only to be re-linked with our version of the MPI functions appearing before the MPI library in the link command line. Traces are stored in a text file, with a header describing the records of each trace.

Automatic generation of I/O signatures from trace files is necessary to use them in a prefetching environment. We developed an analysis tool that reads through traces and gives the I/O signatures of an application as output. It has five pattern detectors for finding patterns among initial positions, offsets, request sizes, temporality, and repetitions. Each of these patterns follows the algorithm shown in Figure 3. For successive trace records, we search for various patterns including fixed strided, *k*-d (*k* = 2 and 3), and negative strided patterns. While searching for patterns, we keep three states: initial state, learning state, and stable state. The search is in the initial state at the beginning and goes into learning state when a

```
Algorithm detect_pattern
Input: Trace file containing IO Read access information
Output: Pattern of a field in a trace file
{
    for (each trace record) {
        read a target field of trace record
        if (state = init)
            read next record
        else {
            search for fixed stride pattern
            if (pattern found)
                state = stable
            else {
                read next record
                search for k-d stride pattern
                    /* k=2 and 3 i.e. 2-d and 3-d nested patterns*/
                If (k-d pattern found)
                    state = stable
                else {
                    read next record
                    find negative strides and search for pattern
                    If (pattern found)
                        state = stable
                    else {
                        read next record
                        (search for pattern using Markov chains
                            for 1st level strides)
                        If (pattern found)
                            state = stable
                            else {
                                state = init
                                read next record
}}}}}}}
```

Figure 3. I/O Pattern Search Algorithm

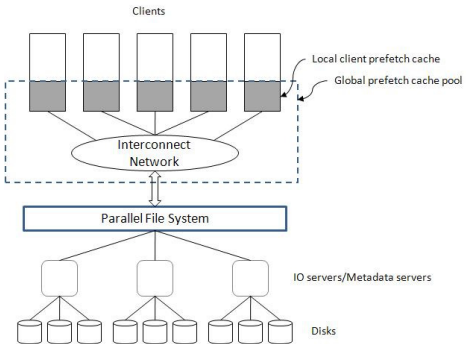

Figure 4. Collective client-side prefetch cache

pattern is first detected. If the same pattern is detected in the following traces, the search state becomes stable. Otherwise, it returns to the learning state and searches for next type of pattern. If no pattern is found, the search will be in initial state. Once a pattern is in stable state, it is written into a signature and the number of repetitions is updated in the signature during the next iterations of the algorithm. All the sub-signatures are combined to form trace signature and pattern signature.

B. Prefetch Cache

A prefetch cache is necessary to store the data that has been prefetched. It is also necessary to keep the data coherent. Several researchers have been working on the idea of a cache on the client side. Cooperative caching has been proposed as a solution for caching and coherence control [8]. Active buffering is an optimization for MPI collective write operations. Liao et al. developed an application-aware collective caching library at the MPI-IO level [15]. Collective caching maintains a global buffer cache among multiple processes in the client side. We use this library as a starting point for our prefetch cache. We modified the cache to hold prefetched pages. Figure 4 shows the high-level design of our collective cache. Each client contributes part of its memory to construct the global cache pool and the high-speed interconnect network between client nodes enables the rapid transfer of cached data among clients. Metadata of cached blocks is maintained to locate them. The contents of metadata include the file descriptor, file offset, rank of process that prefetched the block, and dirty status. A specialized cache-coherency protocol is used to maintain consistency among cache copies in the cache pool. In our prefetch cache, all write caching is disabled and only prefetched pages are cached. We also use a prediction-based replacement policy combined with LRU policy for this cache. In this policy, if a victim is chosen by the LRU policy, and if it is among the pages to be accessed in the near future, it will get a higher priority to stay in the prefetch cache and a different page is selected for replacement.

C. Adjusting Signatures

The post-execution analysis generates I/O signatures of an application and stores them in a text file. Prefetching threads, at runtime, reads the corresponding signature of an application. In order to start prefetching, it is necessary to verify whether the

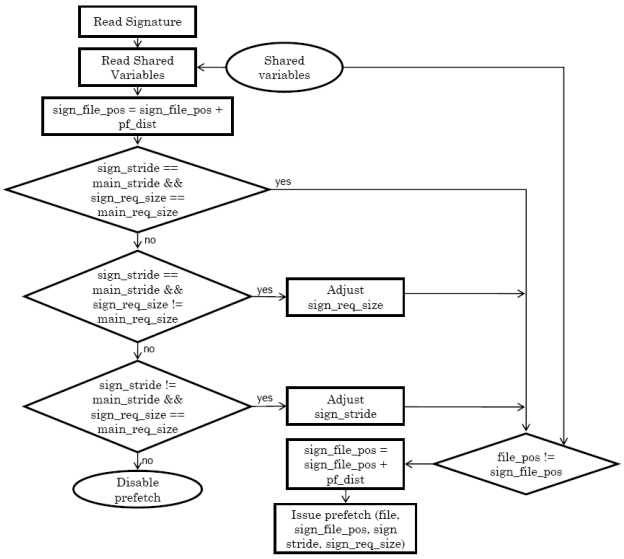

Figure 5. Adjusting I/O Signatures using Shared Variables

signature is following the current I/O requests of the main thread. It is possible that a trace signature values depend on the rank of an MPI process and according to the current I/O accesses of an MPI process, the signature has to be adjusted to start prefetching data efficiently. We implement this signature adjustment process using shared variables. The main thread uses these shared variables to communicate with the prefetching thread. These variables, which are protected with a POSIX mutex, include file handle of the file that is being read, file location (i.e. where in the file the I/O read is occurring), and request size of an I/O read. When the shared variables are available, the prefetching thread reads them and compares them with the current signature. Figure 5 shows the algorithm to adjust signatures. If the stride (*sign_stride*) and request size (*sign_req_size*) of the current signature are same as the stride (*main_stride*) and request size (*main_req_size*) of the main thread I/O reads, respectively, then the prefetching thread assumes that the signature is in a stable state and issues prefetching requests. The initial file location in the trace signature is set from the current file location from shared variables. If either the strides or request sizes of the signature are different with current I/O read parameters (from shared variables), the prefetching thread stays in the learning state and retrieves current strides or request size values to adjust the signature. If both strides and request sizes are different from current I/O read values, prefetching thread does not issue any prefetching requests assuming that prefetching thread does not have the correct signature.

D. Prefetching and Reading from MPI-IO

After a stable pattern is found, prefetching has to be performed to fetch data from I/O servers to prefetch cache. We developed a prefetching library by adding functionality to MPI-IO library. The implementation of the prefetching request has similarities to the implementation of the MPI-IO *read* function,

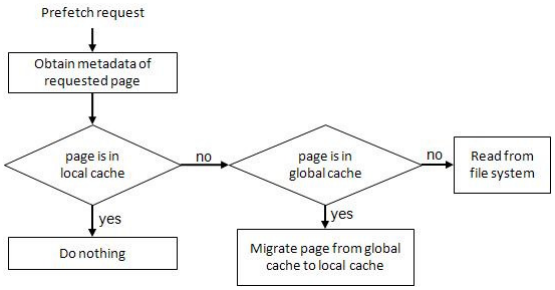

Figure 6. The execution flow for a prefetch request to a file block

but there are some differences. Prefetching requests have a special client-side prefetch cache into which they load, whereas MPI-IO fetches data into the user-specified buffer. Also, prefetching calls do not modify the file pointers. To implement prefetching, we maintain a separate file pointer and offset pointer for the prefetching library, which do not interfere with the original MPI-IO implementation. Prefetching requests are given lower priority over regular MPI-IO requests. If there is contention between a regular I/O request and a prefetching request, the regular request is served first. Also, errors and exceptions caused by prefetching requests are discarded.

Figure 6 shows the execution flow of performing a prefetching request. First, the metadata of the requested page is obtained. If the requested block is in the local client prefetch cache, nothing has to be done and the block is ready to be accessed by the main thread. If it is in the global prefetch cache (at another MPI process's local client prefetch cache), the data will be migrated into the local client cache. If a requested block (page) is not prefetched by any of the clients, data is prefetched from file system. We use a tag to differentiate regular I/O requests from prefetching requests.

To benefit from prefetching, the regular MPI-IO library implementation is modified to be able to access the prefetch cache for requested data in addition to sending the requests to underlying file system. We modified the MPI-IO read operation to lookup in the client-side prefetch cache before issuing a request to the file system. First, the prefetch cache is searched for data. If the data is found in the prefetch cache, data is copied into user buffer. If not, a read request is sent to the file system as in the unmodified version of code. More on design and implementation of prefetching library are available in [2].

E. *Synchronization of Main and Prefetching Threads*

The prefetching thread of an MPI process has to issue prefetching requests early enough to bring data into the cache by the time a regular I/O read is issued by the main thread. in order to be efficient. In our prefetching implementation, we set a prefetch distance of eight. The *prefetch distance* here we mean how many strides away in a pattern we want to prefetch. For instance, if the prefetch distance is eight, prefetching thread requests for data which is eight strides away from current regular I/O read file location, i.e. in a 1-d stride pattern, prefetch position = sign file position + (8 * stride). However, it is possible that the main thread of an MPI process goes ahead of its prefetching thread in sending its regular I/O reads. Prefetching thread has to adjust its prefetching distance in such cases. The prefetching thread again uses shared variables to monitor the progress of regular I/O reads. If prefetching thread is falling behind, we adjust the file position of the signature to progress further and prefetch data ahead.

F. *Maintaining Coherence of Prefetched Data*

Maintaining coherence of prefetched data is important. It is possible that data, which has been prefetched, be modified by another client. In that case, a client reading stale prefetched data causes undesirable results. To avoid that, we use invalidation method to protect coherency of prefetched data. We modify the implementation of MPI-IO write operation to look up data blocks that have been prefetched and invalidate them in the prefetch cache if they are found. No data transfer occurs to or from the prefetch cache to avoid an increased overhead on a write operation.

VI. I/O SIGNATURES OF BENCHMARKS

In our experiments, we carried out experiments first to obtain the I/O signatures of a variety of benchmarks that are widely used to study the behavior of I/O accesses in parallel workloads. From the analysis of their pattern signatures, we then selected I/O workloads that would benefit from prefetching. We discuss I/O signatures of benchmarks in this section, and analyze prefetching performance results in the following section. The benchmarks we studied include IOR [34], mpi-tile-io [30], ASCI FLASH [9], BTIO from NAS parallel benchmark set [23], and PIO-Bench [33]. We characterized the I/O accesses of each benchmark using our classification of patterns and determined their I/O signature.

We ran our experiments on a 17-node Dell PowerEdge Linux-based cluster. This cluster has one Dell PowerEdge 2850 head node with dual 2.8 GHz Xeon processors and 2 GB memory, and 16 Dell PowerEdge 1425 compute nodes, each with dual 3.4 GHz Xeon processors and 1 GB memory. The head node has two 73GB U320 10K-RPM SCSI drives. Each compute node has a 40 GB 7.2K-RPM SATA drive. These nodes are connected with 10/100 Mb/s Ethernet. We conducted our experiments using NFS and PVFS [7]. The PVFS server was configured with one metadata server node and 8 I/O server nodes. All 16 compute nodes were used as client nodes.

A. *IOR*

Interleaved Or Random (IOR) [34] is a parallel file system benchmark developed at Lawrence Livermore National Laboratory. It is available with three APIs: MPI-IO, POSIX, and HDF5. In this study, we present the results with MPI-IO. This test performs write and read operations to a file using MPI_File_write_at and MPI_File_read_at function calls. In our test, each client writes and reads a contiguous chunk of data of size 128 MB overall, and each I/O call writes or reads data of size 32 KB. The total number of writes is 4096 (i.e. 128 MB / 32 KB) and the number of reads is 4096.

We observed offset patterns for write and read operations. The trace signature of IOR for our test is as follows:

{WRITE, 0, 1, ([{0, 32768, 1}, 32768, 4096]), 1}, {READ, 0, 1, ([{0, 32768, 1}, 32768, 4096]), 1}

In these patterns, WRITE and READ operations follow a fixed stride of 32768 bytes offsets and access a fixed size request of 32768 bytes in each I/O operation. This makes the pattern of IOR spatially contiguous, repeating 4096 times. From the trace signature, the pattern signature can be derived as: *{Write (or Read), <Contiguous>, <single occurrence>, <fixed size, medium>, <random>}*.

B. MPI-tile-IO

MPI-tile-io [30] is a test application that implements tiled access to a two-dimensional dense dataset. This benchmark tests the performance of the noncontiguous access pattern obtained by logically dividing a data file into a dense two-dimensional set of tiles. In this test, we studied *read* operations of MPI-tile-IO. Each process accesses a chunk of data based on the size of each tile and the size of each element. For example, we set the size of the tile in the both *x* and *y* directions to 100. The size of each element is 1024 bytes, which makes the size of each tile 10240000 bytes. The trace signature for each process is: *{READ, initial position, 1, ([0, 1024000, 0]), 1}*, where *initial position* is calculated based on the process rank. For example, for process *i*, the *initial position* is (*i*sz_tile_x*sz_element*). From the I/O signature, it can be observed that there is no local spatial pattern because there is only one access from each process. The trace signature of READ operation is similar to that of WRITE pattern.

The pattern signature for MPI-tile-IO traces is: *{Write (or Read), <No spatial pattern>, <single occurrence>, <fixed size, large>, <no temporal pattern>}*. As there is only one I/O operation (write or read) in this trace, a spatial and temporal pattern cannot be defined.

C. NAS Parallel Benchmarks – BTIO

The BT benchmark [23] is based on a CFD code that uses an implicit algorithm to solve the 3D compressible Navier-Stokes equations. A finite-difference grid is assumed, and systems of 5 x 5 blocks at each node are solved using a block-tridiagonal solver. The BTIO version of the benchmark uses the same computational method, but with the addition that the results must be written to disk at every fifth time step. We have tested the CLASS B benchmark with 16 processes (NPROCS= 16) using *full* subtype, which uses collective file operations and describes the solution vector with MPI derived datatypes [22]. This benchmark uses a matrix with a size of (102 x 102 x 102).

The trace signature of I/O signatures for BTIO class B *full* benchmark running on 16 processes is: *{WRITE, initial position, 1, ([42450944, 5308416, 40]), 1}, {READ, initial position, 1, ([42450944, 5308416, 40]), 1}*

The write and read operations of this benchmark are fixed strided with fixed request size and single occurrence. The *initial position* of processes are 0, 1*5308416, 2*5308416, …, 15*5308416 for 0-to-15 process, respectively.

The pattern signature of the BTIO test is: *{Write (Read), <Non-contiguous, fixed strided>, <single occurrence>, <fixed size, large>, <random>}*.

D. ASCI Flash I/O

ASCI FLASH [9] is a parallel application that simulates astrophysical thermonuclear flashes. It uses the MPI-IO parallel interface [22] to the HDF5 data storage library [10] to store its output data, which consists of a checkpoint file, a plotfile with centered data, and a plotfile with corner data. The plotfiles have single-precision data. In this benchmark, the underlying MPI-IO functions are used to create a single file. All processes write data directly to this file with the size of a standard grid of 8 x 8 x 8. This data is of double-precision. The computational domain is divided into blocks, which are distributed across the processors. Each block contains 8 zones in each coordinate direction (x,y,z) and a perimeter of guard-cells (4 zones deep) to hold information from the neighbors. Typically, there are 24 variables per zone, and fit about 100 blocks on each process. So, the size of each record in each process is (8 bytes/number * 8 zones in *x* * 8 zones in *y* * 8 zones in *z* * 100 blocks) = 400 KB.

In this test, we executed the Flash IO benchmark with 4 processes to retrieve its I/O access patterns. Each process calls the collective write function (MPI_File_write_all) 50 times. There is no regular pattern among all 50 writes; however, there are three sets of patterns, while the remaining accesses are random. The trace signature of the I/O signature for the patterns is: *{WRITE, initial position_1, 1, ([1323008, request_size_1, 25]), 1}, {WRITE, initial position_2, 1, ([661504, request_size_2, 4]), 1}, {WRITE, initial position_3, 1, ([941868, request_size_3, 4]), 1}*

Another interesting characteristic of trace signature of this benchmark is that not only the initial position of each process is different, but also the request size of each process. The initial positions of the first pattern (i.e. *initial position_1*) of the signature for the four processes are: 54680, 382360, 714136, and 1050008. The request sizes of the first pattern (i.e. request_size_1) are: 327680, 331776, 335872, and 327680. We observe similar variation in initial positions and request sizes in the two other patterns. This result is interesting because many prefetching systems assume that each process requests the same amount of data. However, in this benchmark, the I/O signature demonstrates the necessity of a prefetching strategy to adapt to variable sizes of requests among multiple processes. From these observations, the pattern signature in all three patterns in this test is: *{Write, <non-contiguous, fixed strided>, <single occurrence>, <fixed size, large>, <random>}*.

We also tested this benchmark with 16 processes to verify the variability of *initial positions*, *offsets,* and *request sizes* with a different number of processes. The *initial positions* and *offsets* with this test are different from those observed by running the benchmark on 4 processes. However, the *request*

sizes are similar in both these tests. We have also observed that the trace signature of I/O signature is the same for multiple runs of the benchmark with the same number of processes. This is useful for prefetching strategies to use an I/O signature when running the benchmark with the same number or processes but working on different datasets.

E. PIO-Bench

PIO-Bench is a synthetic parallel file system benchmark suite that is designed to reflect I/O access patterns appearing in typical workloads of real applications. This benchmark suite tests several I/O access patterns including sequential, simple strided, nested strided, random strided, segmented access, and tiled patterns. We studied simple-strided, nested-strided and segmented accesses of this benchmark. In each of the PIO-Bench tests, we set various parameters including I/O operation, type of access pattern, number of repetitions, and request size. In these tests, we analyzed the read accesses. Tests with the write operations also follow the same patterns.

In simple-strided test, we set the number of repetitions as 100, and the request size as 4096 bytes. All the running processes use the same pattern, but with different *initial position*. The trace signature of this test is: *{READ, initial position, 1, ([8192, 4096, 100]), 1}* and the pattern signature of this pattern is: *{Read, <non-contiguous, fixed strided>, <single occurrence>, <fixed size, small>, <random>}.*

In nested-strided test, we set the number of repetitions as 100, and the request size as 4096 bytes. The trace signature of this test is: *{READ, initial position, 2, ([4096, 4096, 1], [12288, 2048, 1], 100), 1}* and the pattern signature of this pattern is: *{Read, <non-contiguous, fixed strided>, <single occurrence>, <fixed size, small>, <random>}.*

For segmented accesses test, processes access different segments of a file. The local access pattern for each process is contiguous. The trace signature for this test with request size 4096 bytes is: *{READ, initial position, 1, ([4096, 4096, 100]), 1}* and the pattern signature of this pattern is: *{Read, <contiguous>, <single occurrence>, <fixed size, small>, <random>}.*

VII. PERFORMANCE RESULTS WITH PREFETCHING

I/O signatures of an application are useful for preliminary analysis of whether prefetching is beneficial in improving its performance. From observation of I/O signatures, we can conclude that the read pattern of IOR is contiguous, whose benefit from prefetching is similar to that of PIO-Bench test with simple strided pattern, where the stride is zero. MPI-tile-io only has one read access, where prefetching is not be beneficial. Flash IO only has writes and no reads. This leaves PIO-Bench with simple strided patterns, PIO-Bench with nested strided patterns, and BTIO are good candidates for benefiting from prefetching. We conducted three sets of experiments: two with strided and nested strided patterns using PIO-Bench and another with BTIO benchmark. Each PIO-Bench test is run with multiple strides by varying the number

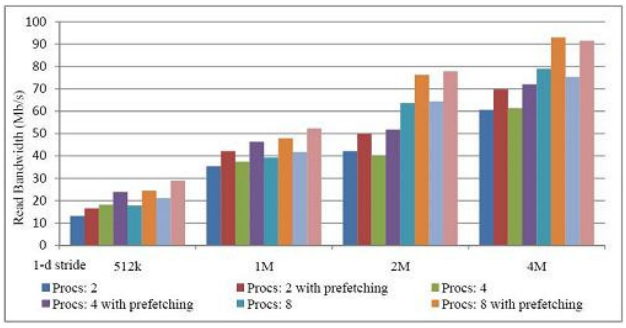

Figure 7. Bandwidth of PIO-Bench, with Simple Strided pattern on NFS

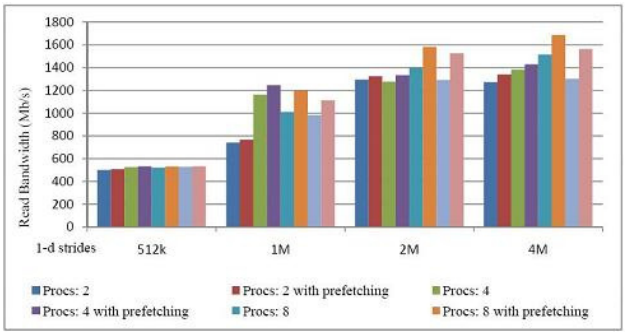

Figure 8. Bandwidth of PIO-Bench, with Simple Strided pattern on PVFS

of processors 2, 4, 8, and 16. For BTIO we evaluated Class B I/O size with 4, 9, and 16 processors. We set the cache block size for the collective prefetch cache as 64KB, and the prefetch cache size at each client was set as 32MB. We ran these experiments on the same 17-node Dell PowerEdge Linux-based cluster as we mentioned above.

Figures 7 compares the I/O read bandwidth results of PIO-Bench with 1-d strides (from 512 KB to 4MB) using NFS with collective prefetch cache and native approach (without prefetch cache). Figure 8 shows the same bandwidth results with PVFS. In order to focus on the performance benefits of prefetching, we measured only the time for I/O read requests. Each reported result is the least value of multiple runs; we observed that the variation among performance numbers in multiple runs was negligible. With PIO-Bench, the amount of data transferred between disks and MPI processes is higher as the size of stride increases for the same number of I/O reads. Hence, there is an increase in read bandwidth as the stride increases. From the figures we can see that the I/O read bandwidth with prefetching is better than that with no prefetching for all stride sizes with various numbers of processors. On average, the performance gain with prefetching for multiple stride sizes on NFS is ~23%. On PVFS, the performance gain is lower than that on NFS, probably because PVFS has optimizations for strided patters to combine multiple accesses. Prefetching benefits are low for small number of processors as well as for small stride between successive I/O reads. As the number of processors increases, the performance gains increase to ~14%.

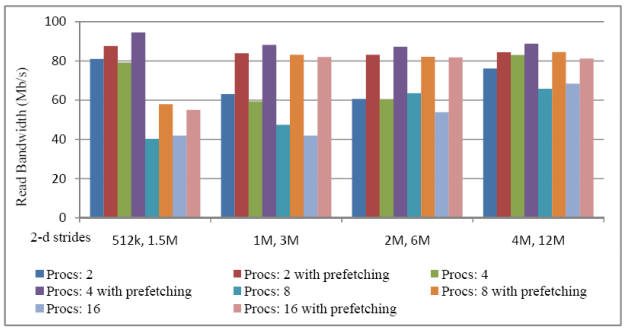

Figure 9. Bandwidth of PIO-Bench, with Nested Strided pattern on NFS

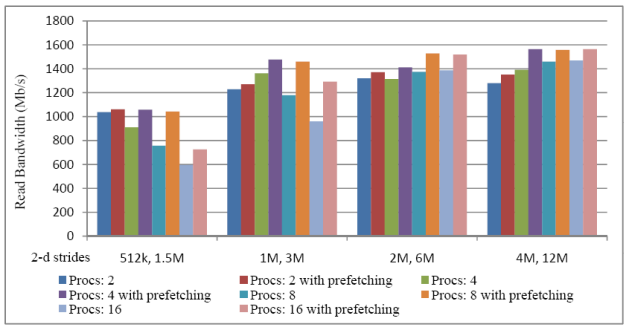

Figure 10. Bandwidth of PIO-Bench, with Nested Strided pattern on PVFS

Figure 9 compares the I/O bandwidth results of PIO-Bench Nested strided pattern using NFS with and without prefetching, and Figure 10 compares them with PVFS. The difference in offsets between successive I/O reads is 2-d strided in these tests. It is noticeable that bandwidth reduces as the number of processes increase for smaller strides. The I/O read performance with prefetching is better than that with no prefetching in all cases here. The performance gain with prefetching is low for small number of processors and it increases as the number of processors grows in both NFS and PVFS tests. In the 2-d strided I/O reads, there is a lack of locality, which requires loading a different page for each access. As the number of processors grows, serving each processor increases the load on I/O servers and results in poor performance. Prefetching methods benefits highly in these cases by sending requests early. The average performance gain on NFS is ~36% and that on PVFS for larger strides is ~20%.

Figures 11 and 12 compare the I/O bandwidth performance of BTIO (Class B) with and without prefetching on NFS and on PVFS, respectively. We can see that on both NFS and PVFS, we can see I/O read bandwidth improvement with different number of processors. The read accesses have fixed 1-d stride pattern, and the prefetching benefits are similar to the case of 1-d stride in the PIO-Bench test (Figures 7 and 8). The performance gain for all processors is same on NFS (~25%). On PVFS, with 4 processors, the gain is ~8%, and as the number of processors increase, the performance gain increases to 15% (9 and 16 processor tests).

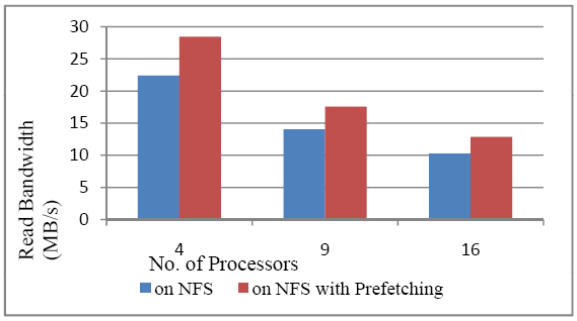

Figure 11. Bandwidth of BTIO Reads with Prefetching on NFS

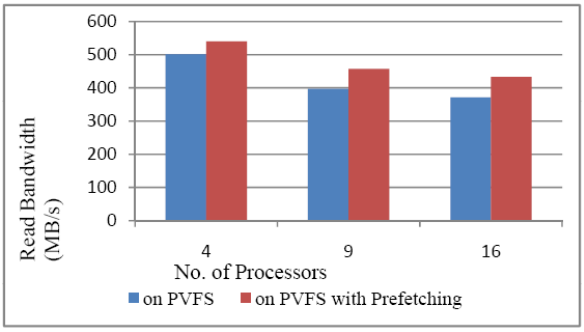

Figure 12. Bandwidth of BTIO Reads with Prefetching on PVFS

VIII. Conclusions and Future Work

Poor parallel I/O performance has been a bottleneck in many data-intensive applications. Prefetching is an effective latency hiding optimization. However, traditional prefetching strategies limit themselves to simple strided patterns and avoid complex prediction strategies so that a prefetching decision can be made swiftly. In this study, we have introduced a combination of post-execution analysis with runtime analysis to reduce the overhead of predicting future I/O reads. We expanded the classification of I/O accesses and introduced a new I/O signature notation. This notation can be used in reducing the size of trace file, in making decisions about prefetching, and in selecting prefetching strategies. The signature is predetermined and is attached to the underlying application for future executions. In our prefetching strategy, we used a separate thread to prefetch at runtime. We implemented this in an MPI-IO implementation (ROMIO [38]) and our results show significant performance benefits. There is no extra effort required by an MPI application developer to utilize our prefetching strategy. In the future, we plan to extend this research in two directions: (1) to move the prefetching thread to the file system level, and (2) to use a separate server to push data from I/O server nodes to client nodes. Moving the prefetching thread into a file system removes overhead on the client side, and the file system can combine multiple small I/O reads of multiple processes and prefetch large chunks of data. Using a separate server to move data reduces the burden on both I/O servers as well as clients. We also believe that prefetching is only one application of the signature-based

approach. The signature approach has the potential to guide data access optimizations of general heterogeneous computing by describing data requirements and is worthy of further investigation.

ACKNOWLEDGMENT

We are thankful to Dr. Wei-Keng Liao and Dr. Alok Choudhary of Northwestern University for providing their collective caching code.

REFERENCES

[1] Surendra Byna, Xian-He Sun, William Gropp and Rajeev Thakur, "Predicting the Memory-Access Cost Based on Data Access Patterns", in Proceedings of the IEEE International Conference on Cluster Computing, San Diego, September 2004.

[2] Y. Chen, S. Byna, X.-H. Sun, R. Thakur, W. Gropp. "Exploring Parallel I/O Concurrency with Speculative Prefetching", in Proc. 37th International Conference on Parallel Processing (ICPP '08), Sept. 2008.

[3] Phyllis E. Crandall, Ruth A. Aydt, Andrew A. Chien, and Daniel A. Reed, "Input/output characteristics of scalable parallel applications", in Proceedings of the ACM/IEEE conference on Supercomputing, pp.59-es, December 1995.

[4] Cluster File Systems Inc., "Lustre: A scalable, high performance file system", Whitepaper, http://www.lustre.org/docs/whitepaper.pdf

[5] F. Chang, "Using Speculative Execution to Automatically Hide I/O Latency", Carnegie Mellon Ph.D Dissertation CMU-CS-01-172, December 2001.

[6] F. Chang and G.A. Gibson, "Automatic I/O Hint Generation through Speculative Execution", in Proceedings of the 3rd Symposium on Operating Systems Design and Implementation, February 1999.

[7] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur, "PVFS: A Parallel File System For Linux Clusters", in Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta, GA, October 2000, pp. 317-327

[8] M. Dahlin, R. Wang , T. Anderson , D. Patterson, "Cooperative caching: using remote client memory to improve file system performance", in Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation, November 1994.

[9] FLASH IO Benchmark, Routine – Parallel HDF5, http://www.astro.sunysb.edu/mzingale/flash_benchmark_io/

[10] The HDF5 Group, HDF5 - A New Generation of Hierarchical Data Format, http://hdf.ncsa.uiuc.edu/products/hdf5/index.html

[11] B. Hendrickson and D. Womble, "The torus-wrap mapping for dense matrix calculations on massively parallel computers", SIAM Journal of Scientific Computing, 15(5), September 1994.

[12] K. Keeton, G. Alvarez, E. Riedel, and M. Uysal. "Characterizing I/O-intensive Workload Sequentiality on Modern Disk Arrays", in Proceedings of the 4th Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW-01), January 2001.

[13] D.F. Kotz and C.S. Ellis, "Prefetching in File Systems for MIMD Multiprocessors", IEEE Transactions on Parallel and Distributed Systems, 1(2), pp. 218-230, 1990.

[14] David Kotz and Nils Nieuwejaar, "Dynamic File-Access Characteristics of a Production Parallel Scientific Workload", in Proceedings of Supercomputing '94, pp. 640-649, November, 1994.

[15] W.K. Liao, K. Coloma, A. Choudhary, L. Ward, E. Russel and S. Tideman, "Collective Caching: Application-Aware Client-Side File Caching", in Proceedings of the 14th International Symposium on High Performance Distributed Computing, 2005.

[16] X. Ma, J. Lee and M. Winslett, "High-level Buffering for Hiding Periodic Output Cost in Scientific Simulations", IEEE Transactions on Parallel and Distributed Systems, Vol. 17, No. 3, 2006.

[17] T.M. Madhyastha and Daniel A. Reed, "Learning to classify parallel input/output access patterns", in IEEE Transactions on Parallel and Distributed Systems, Volume 13, Issue 8, pp. 802 – 813, Aug 2002.

[18] Ethan L. Miller , Randy H. Katz, "Input/output behavior of supercomputing applications", in Proceedings of the 1991 ACM/IEEE conference on Supercomputing, pp. 567-576, November 1991.

[19] Jaydeep Marathe, Frank Mueller, Tushar Mohan, Sally McKee, Bronis de Supinski, and Andy Yoo, "METRIC: Memory tracing via dynamic binary rewriting to identify cache inefficiencies", ACM Transactions on Programming Languages and Systems, 29(2), April 2007.

[20] J. May, "Parallel I/O For High Performance Computing", Morgan Kaufmann Publishing, 2001.

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

[22] Message Passing Interface Forum, "MPI-2: Extensions to the Message-Passing Interface", July 1997 1996. http://www.mpi-forum.org/docs/docs.html

[23] NAS Parallel benchmarks, http://www.nas.nasa.gov/Resources/Software/npb.html

[24] Nils Nieuwejaar, David Kotz, Apratim Purakayastha, Carla Schlatter Ellis, and Michael Best, "File-Access Characteristics of Parallel Scientific Workloads", IEEE Transactions on Parallel and Distributed Systems, 7(10) pp. 1075–1089, October 1996.

[25] R.H. Patterson and G. Gibson, "Exposing I/O Concurrency with Informed Prefetching", in Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems, 1994.

[26] Barbara Pasquale and George C. Polyzos, "A static analysis of I/O characteristics of scientific applications in a production workload", in Proceedings of the 1993 ACM/IEEE Conference on Supercomputing, pp. 388-397, December 1993.

[27] Barbara K. Pasquale and George C. Polyzos, "Dynamic I/O characterization of I/O intensive scientific applications", in Proceedings of the 1994 Conference on Supercomputing, pp. 660-669, 1994.

[28] A. Papathanasiou and M. Scott, "Aggressive Prefetching: An Idea Whose Time Has Come", in Proceedings of the Tenth Workshop on Hot Topics in Operating Systems, 2005.

[29] Apratim Purakayastha, Carla Ellis, David Kotz, Nils Nieuwejaar, and Michael Best, "Characterizing Parallel File-Access Patterns on a Large-Scale Multiprocessor", In Proceedings of the Ninth International Parallel Processing Symposium (IPPS), pp. 165-172, April, 1995.

[30] Parallel I/O Benchmarking Consortium, http://www-unix.mcs.anl.gov/pio-benchmark/

[31] Daniel A. Reed, Ruth Aydt, Roger Noe, Philip Roth, Keith Shields, Bradley Schwartz, and Luis Tavera, "Scalable Performance Analysis: The Pablo Performance Analysis Environment", in Proceedings of the Scalable Parallel Libraries Conference, October 1993, pp. 104-113.

[32] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters", in Proceedings of the First USENIX Conference on File and Storage Technologies, pp. 231-244, USENIX, January 2002.

[33] Frank Shorter, "Design analysis of a performance analysis standard for parallel file systems", Masters Thesis, August 2003, ftp://ftp.parl.clemson.edu/pub/techreports/2003/PARL-2003-001.ps

[34] Scalable I/O Project Software Downloads, IOR software, http://www.llnl.gov/icc/lc/siop/downloads/download.html

[35] Evgenia Smirni and Daniel A. Reed, "Workload Characterization of Input/Output Intensive Parallel Applications," Proceedings of the Conference on Modeling Techniques and Tools for Computer Performance Evaluation, Springer-Verlag Lecture Notes in Computer Science, vol. 1245, pp. 169-180, June 1997.

[36] Evgenia Smirni and Daniel A. Reed, "Lessons from Characterizing the Input/Output Behavior of Parallel Scientific Applications", in Performance Evaluation, volume 33, pp. 27-44, 1998.

[37] Rajeev Thakur, William Gropp, and Ewing Lusk, "Data Sieving and Collective I/O in ROMIO", in Proceedings of the 7th Symposium on the

Frontiers of Massively Parallel Computation, February 1999, pp. 182-189.

[38] Rajeev Thakur, Robert Ross, Ewing Lusk, and William Gropp, "Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation," Technical Memorandum ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, Revised May 2004.

[39] Nancy Tran, Daniel A. Reed. "Automatic ARIMA Time Series Modeling for Adaptive I/O Prefetching," IEEE Transactions on Parallel and Distributed Systems, vol. 15, no. 4, pp. 362-377, April, 2004.

[40] Mustafa Uysal, Anurag Acharya, and Joel Saltz, "Requirements of I/O Systems for Parallel Machines: An Application-driven Study", Technical Report, CS-TR-3802, University of Maryland, College Park, May 1997.

[41] C.K. Yang, T. Mitra and T. Chiueh, "A Decoupled Architecture for Application-Specific File Prefetching", in Freenix Track of USENIX 2002 Annual Conference, 2002.