InterferenceRemoval: Removing Interference of Disk Access for MPI Programs through Data Replication

Xuechen Zhang and Song Jiang
The ECE Department
Wayne State University
Detroit, MI, 48202, USA
{xczhang, sjiang}@wayne.edu

ABSTRACT

As the number of I/O-intensive MPI programs becomes increasingly large, many efforts have been made to improve I/O performance, on both software and architecture sides. On the software side, researchers can optimize processes' access patterns, either individually (e.g., by using large and sequential requests in each process), or collectively (e.g., by using collective I/O). On the architecture side, files are striped over multiple I/O nodes for a high aggregate I/O throughput. However, a key weakness, the access interference on each I/O node, remains unaddressed in these efforts. When requests from multiple processes are served simultaneously by multiple I/O nodes, one I/O node has to concurrently serve requests from different processes. Usually the I/O node stores its data on the hard disks, and different process accesses different regions of a data set. When there are a burst of requests from multiple processes, requests from different processes to a disk compete with each other for its single disk head to access data. The disk efficiency can be significantly reduced due to frequent disk head seeks.

In this paper, we propose a scheme, *InterferenceRemoval*, to eliminate I/O interference by taking advantage of optimized access patterns and potentially high throughput provided by multiple I/O nodes. It identifies segments of files that could be involved in the interfering accesses and replicates them to their respectively designated I/O nodes. When the interference is detected at an I/O node, some I/O requests can be re-directed to the replicas on other I/O nodes, so that each I/O node only serves requests from one or a limited number of processes. *InterferenceRemoval* has been implemented in the MPI library for high portability on top of the Lustre parallel file system. Our experiments with representative benchmarks, such as NPB *BTIO* and *mpi-tile-io*, show that it can significantly improve I/O performance of MPI programs. For example, the I/O throughput of *mpi-tile-io* can be increased by 105% as compared to that without using collective I/O, and by 23% as compared to that using collective I/O.

ICS'10, June 2–4, 2010, Tsukuba, Ibaraki, Japan.

Categories and Subject Descriptors

D.4.2 [**OPERATING SYSTEMS**]: Storage Management—*Secondary storage*

General Terms

Design, Performance

Keywords

MPI program, MPI-IO, and I/O Interference

1. INTRODUCTION

I/O-intensive applications are widely used in scientific computation and engineering simulation. For example, running an *astro* application for analysis of astronomical data can easily result in more than 50GB data on disk and 62% of the total execution time involving disk I/O [13]. Therefore, efficient I/O support is critical for their performance. Without significantly increasing efficiency of accessing data on disk and ameliorating the I/O bottleneck, the application performance can hardly benefit from the increasing number of compute nodes or number of cores on a node, and the entire system can be seriously underutilized by leaving many processor cycles idle. While MPI parallel programs dominate scientific and engineering applications running in major parallel systems [17], many efforts have been made over years both on the software side and on the system architecture side to improve I/O performance.

The efforts on the software side are mainly on forming large and sequential requests, which can reduce request processing overhead and significantly improve hard disk efficiency. To this end, many middlewares, most in the form of libraries, have been developed. Programmers can optimize data access pattern within individual processes, using techniques including data sieving [20], datatype I/O [2], and list I/O [3], or across processes of an MPI program, such as collective I/O [20, 10, 18, 25]. Using these techniques, requests sent by a process can become larger and more sequential. However, requests from different processes are sent to the storage system in an uncoordinated order. On the hardware side, today's high performance computing platform usually adopts a dedicated storage system, which is composed of multiple I/O server nodes and managed by parallel file systems such as Lustre [4], PVFS [16], or GPFS [19]. In these systems, user files are striped over multiple I/O nodes with a constant striping unit size. Thus, the data requested in a large request could span multiple I/O nodes, and the sys-

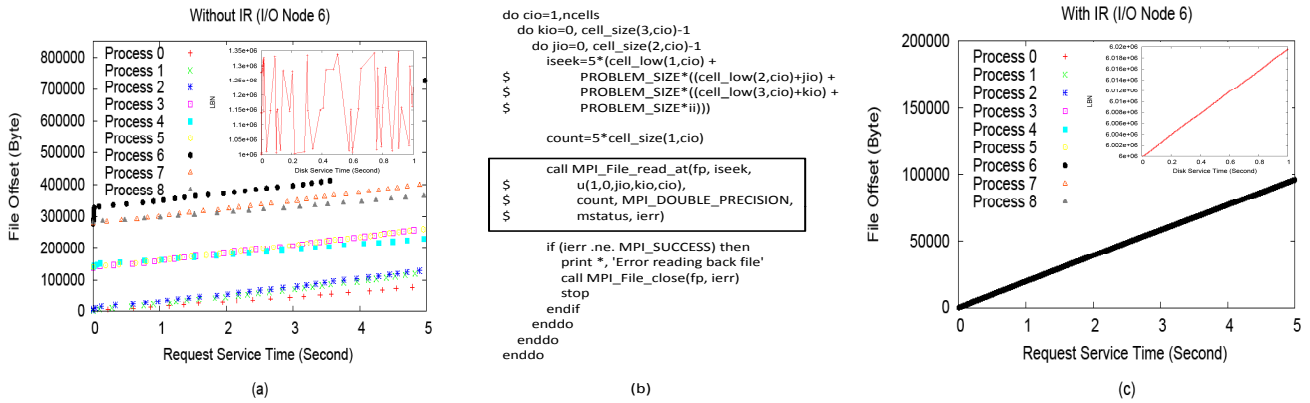

Figure 1: (a) The file offsets that are mapped onto an I/O node (Node 6) and are accessed in a five-second execution period of program *BTIO*. Note that the file offsets represent relative on-disk locations assuming data of a large file are generally contiguously laid out on the disk. (b) A fragment of source code of the MPI program issuing the requests (wrapped in the box). (c) The corresponding file offsets that are accessed in a five-second execution period of program *BTIO* on I/O node 6 with *InterferenceRemoval* enabled. The small figures on the upper right corners of figures (a) and (c) illustrate the paths of disk head seeks in the first seconds in their respective runs[1].

tem efficiency is expected to improve through exploring the parallelism in serving large requests.

1.1 A Motivating Example

Unfortunately, the I/O throughput experienced by an MPI program can be still low, even though requests from processes, probably with optimized access pattern, are served simultaneously by multiple I/O nodes. To obtain the insights behind the observation, we run the NAS BTIO benchmark [15], an MPI program, on a cluster of 13 nodes. In the cluster, six nodes are configured as compute nodes and the other seven as I/O nodes, managed by Lustre parallel file system. File data is striped over six I/O servers (More details of the experimental platform can be found in Section 4.). In the experiment, the benchmark spawns nine processes using non-collective I/O to write 1.5GB data to one on-disk file and then read them. We monitor data accesses at each I/O node and record from which process requests are issued. Figure 1 (a) shows file offsets that are mapped onto *one* particular I/O node and are accessed by each of the nine processes during a five-second execution period. Figure 1 (b) shows the corresponding source code issuing the requests. From Figure 1 (a), we can see that requests made by each process are well aligned to allow the disk to efficiently serve them if they were the only requests sent to the disk. However, the multiple processes of the MPI program simultaneously send their requests to the I/O node, causing them to be served on the disk in an interleaving fashion. That is, the disk head has to move back and forth to access the requested data, which significantly harms the efficiency of the disk and the I/O system. To reveal the severe performance impact of this inter-process interference in the program's execution, we run the same program with our *InterferenceRemoval* scheme enabled, in which no more than two processes send their requests to one I/O node. Figure 1 (c) shows file offsets that are accessed on the aforementioned I/O node with the scheme. By having sequential requests received and efficiently served at each I/O node, the program execution time is reduced by 3.4X, from 4854 seconds to 1415 seconds.

[1]We use disk block-level tracing tool *blktrace* to collect information on accessed disk locations and access times. The disk location is represented by its corresponding logical block number (LBN).

1.2 Inadequacy of Existing Solutions

The root cause of the inter-process interference is that each I/O node receives and serves concurrently multiple requests from different processes. When the processes usually do not coordinate the order for them to issue the requests, the timings for the requests to be sent depend on the relative speeds of the processes, which are usually indeterministic . Therefore, the request arrival order is indeterministic or essentially random, and is likely to cause the disk head to thrash in accessing these requested data. Furthermore, a request may span over multiple I/O nodes, and an I/O node may service only part of the request. This would cause each I/O node to be more likely to receive requests from multiple processes.

Currently there are some strategies that could ameliorate the problem in certain circumstances. If the requests from a process are for fully contiguous file data, the file prefetching mechanism on the I/O node can increase the amount of data read from the disk with each disk head seek, amortizing the seek overhead. However, it is effective only for large sequential read. For the *BTIO* benchmark, the access is mixed with read and write and is not fully sequential, thus prefetching would not help. In addition, the overhead for extracting the sequential access from mixed request streams can be high [11]. As we know, the disk scheduler can optimize disk operations by sorting and merging requests when they reach its dispatch queue for improved I/O efficiency. However, it requires many requests be available in the queue for its manipulation. In many cases, requests are synchronously issued and the next request would not be generated until the completion of the previous request. Usually it is the request arrival order that determines how the disk head moves.

Understandably programmers may have taken great ef-

forts in arranging sequential data access. As these efforts are mostly made within each process's control flow, it does not remove the identified interference, which occurs among requests from different processes. Furthermore, the coordination among processes' I/O operations, such as collective I/O, does not effectively address the issue. Collective I/O rearranges the data access scope among participating processes to form large contiguous requests and synchronizes the processes before or after issuance of the I/O requests. Even if the processes are synchronized before sending their requests, the synchronization does not optimize the order in which the requests arrive at a disk, which can still be random and substantially degrade disk performance [25]. Conceivably, programmers can enforce a request issuance order friendly to disk efficiency, possibly with the help of middleware, by coordinating the timings of sending requests from different processes with the goal that requests from processes for data at lower disk addresses arrive earlier. Unfortunately, this approach is not feasible or effective. First, as each request may touch multiple disks, the approach may have to serialize the issuance of requests to achieve its goal, eliminating the benefits of parallelism available in multiple I/O nodes. Second, the approach has to synchronize between pairs of processes, instead of a global synchronization used in the collective I/O, causing a high overhead. Third, the order in which requests arrive at the disks can be different from the order in which requests are sent.

1.3 Opportunities and Our Solution

Though inter-process and across-nodes interference issue identified in this paper has not been well addressed, the interference within a hard disk has been well studied, and many techniques for reducing the interference have been proposed, which are usually to replicate data within one disk and make the originally distant data close to each other [5, 1]. However, the data requested by processes of an MPI program may have already been close to each other and the interference is caused by the random request-arrival order in an I/O burst. As MPI programs dominate the applications running on parallel computers, and I/O efficiency becomes increasingly important on these platforms, removing interference specifically for this running environment becomes an urgent issue. Meanwhile, addressing the interference issue specifically in this context provides us with unique opportunities for a better solution. First, a production MPI program is usually executed on a parallel computer for numerous times, possible with different set of data files. As the data access patterns of its processes are mostly consistent from one run to another run, we can identify segments of files that are involved in the interference in one run and expect access of these data would be the source of interference in other runs. Thus, we can remove the interference by eliminating this source identified beforehand. Second, in the previous studies on the on-disk interference, the focus is on the rearrangement of the data layout within one disk to minimize the frequency and distance of its disk head seeks. As there is only one disk head in a disk, the efficacy of these efforts for removal of interference is limited. In contrast, the existence of multiple I/O nodes in a parallel system provides an opportunity to remove interference detected on each disk through the coordinated efforts among multiple disks. In other words, instead of adapting disk layout to predicted request arrival order[2] via data replication within one disk, we replicate data that can be involved in an interference from one disk to other disks, so that multiple disk heads can serve the requests independently without interference.

In this paper, we propose a data replication method, named as *InterferenceRemoval*, to eliminate I/O interference among processes of an MPI program. In the approach, we use a profiling run of the program to identify segments of files involved in the disk access interference and replicate them to other I/O nodes. In this work we made the following contributions.

- We design a method to collect I/O traces in which the recorded request times are consistent across processes, allowing us to identify interference at each I/O node.
- The interference is determined based on comparison of the I/O efficiency between the original striped-data layout and intended replicated-data layout. To facilitate the evaluation, we develop a simulation-based approach to evaluate their relative efficiency and carry out only those replications deemed as cost effective.
- As removing interference for each I/O node independently through data replication may simply move interference from one node to another node or aggravate other nodes' interference, we design a scheme that coordinates the data replication across the nodes with a concerted effort to minimize number of processes from which an I/O node receives requests.
- We implemented the interference removal scheme fully in MPI-IO library in a cluster using the Lustre parallel file system. It is highly portable and can be easily adopted without assuming any special support from OS, file system, or I/O nodes. Our experiments with representative benchmarks, such as *BTIO*, *ior-mpi-io*, *mpi-tile-io*, *coll_perf*, show that *InterferenceRemoval* can improve I/O throughput by up to 33X.

The rest of this paper is organized as follows. Section 2 discusses the related work. Section 3 describes the design and implementation of *InterferenceRemoval*. Section 4 describes and analyzes experiment results, and Section 5 concludes the paper.

2. RELATED WORK

As I/O-intensive applications become increasingly important on the parallel platforms, a lot of work on the improvement of I/O performance for high-performance computing has been carried out. This work includes development of I/O middlewares and optimization of system architecture. While the hard disk is usually one of the major I/O performance bottlenecks in a parallel system, enabling the peak throughput of the disk is one of the major concerns.

I/O middlewares usually help to turn a large number of requests for small and non-contiguous pieces of data into a smaller number of large contiguous requests. In addition to reducing request processing overhead, this can potentially allow disks to be accessed sequentially, which is critical for a disk's throughput. Data sieving [20] is one of the techniques

[2]Actually such prediction is hard to be accurate, especially for requests from different processes.

proposed to aggregate small requests into large ones. Instead of accessing each small piece of data separately, data sieving accesses a large contiguous scope of data that includes the small pieces of data. If the additional data, called holes, accessed in data sieving is not excessive, the benefit can be large. Datatype I/O [2] and list I/O [3] are the other two techniques that allow users to specify multiple non-contiguous accesses with a single I/O function call. Between them, datatype I/O is used for accesses with certain regularity and list I/O can handle a more general case. Compared with these techniques applied in each individual process, collective I/O can infer a big picture of access pattern among multiple processes of an MPI program to enable optimization in a greater scope. It aggregates small requests into large contiguous ones by re-arranging requests collectively issued by a group of processes. While collective I/O can incur communication overhead for exchanging data among participant processes, its performance advantage is well recognized and the technique has been widely used.

However, the benefits of these techniques could be reduced or even eliminated in an environment where file data are striped over multiple I/O node, which is the most popular I/O system architecture assumed by major parallel file systems such as PVFS [16] , Lustre [4], and GPFS [19]. In a study of impact of data striping pattern on the performance of ROMIO collective I/O implementation, it has been shown that the I/O throughput of some popular I/O benchmarks can be reduced to as low as 38% by using collective I/O [25]. The reason for this large degradation is the on-disk interference caused by random arrival order at an I/O node for requests from different processes. By providing an additional interference-resistant data layout on the I/O nodes through selective data replication, we aim to enable the full potential performance benefits from the I/O middlewares as well as the I/O parallelism provided by the dedicated I/O system.

Regarding the performance degradation of the hard disk due to data access interference, there have been many schemes proposed to reorganize data layout on a disk [6, 7] or replicate data within the same disk (such as FS2 [5] and BORG [1]) according to detected access patterns. These schemes are effective when the access patterns are stable, such as the access patterns exhibited within individual processes or the coordinated issuance requests from distinct processes. For the interference concerned in our work, it is the unpredictable timings when the different processes of an MPI program send their requests that cause the interference. Thus, the order among requests from different processes is hard to repeat. By allowing requests from different processes to be served at different I/O nodes through data replication, our scheme can avoid the uncertainty in the access order by taking advantage of existence of multiple I/O nodes in a system.

Anticipatory scheduling is another widely used technique to remove interference among multiple streams of synchronous requests to a disk [9]. In the scheduling, after serving a request from a process, the disk may be temporarily kept from serving requests from other processes. Instead, it anticipates additional requests from the same process to reduce disk head seeks. However, because file data are spread over multiple I/O nodes and continuous requests can be served at different disks, a disk is not likely to quickly see its next request from the same process as it anticipated and thus potential performance gain could be neutralized. In our scheme, the disk does not have to risk its waiting time by being idle, because a disk is serving requests from only one or a small number of processes once interference is detected.

Data replication has been used in the parallel computing environment. To improve reliability related to data loss due to the failure of I/O nodes, the replication scheme proposed in [24] coordinates with a job scheduler to replicate data files, creating a data copy of a different striping pattern across the I/O nodes. When a failure is detected, replicas on I/O nodes other than the failed ones can be used to restore the original files instead of aborting and resubmitting the whole job. Their experiment measurements show that the cost for their replication is affordable. Wang et al. [23] proposed to replicate frequently accessed data chunks to the compute nodes' local disks to reduce data access latency. The frequently accessed data chunks are identified through analysis of I/O traces collected in a profiling run. The legitimacy of their approach of identifying data of certain access patterns through profiling runs is built on their observations that "In scientific applications, file access patterns are generally independent of the data values stored." and "When the number of processes changes or the size of the input data file changes, the pattern of file accesses changes very predictably, so re-profiling can be avoided.". Our profiling-based interference detection and data replication are based on the similar rationale. In addition, the increasingly large hard disk capacity and under-utilized disk space support the use of idle disk spaces to trade for higher I/O performance [5].

3. THE DESIGN AND IMPLEMENTATION

As I/O requests from processes of an MPI program are served in parallel by multiple I/O nodes, where data files are striped, the interference at each I/O node among different processes' requests is likely to offset the potential performance benefits from parallelism in the program execution and I/O service. To eliminate the interference and fully take advantage of the parallelism, we design a scheme, *InterferenceRemoval*, for the MPI programs, to detect interference for individual I/O nodes and conduct data replication for selected file segments, request scheduling, and management of replica consistency. To make our scheme portable and be easily adopted in different systems, we implement all components of the scheme in the middleware on the compute-node side, specifically in the MPI-IO library.

3.1 Detection of Access Interference

To remove the interference, we need a mechanism to detect it, or identify the interaction among continuously served requests at a hard disk that causes substantial disk performance degradation. However, we cannot simply measure the disk throughput and compare it with its peak throughput to evaluate the impact of the interaction or determine whether an interference takes place. A disk's actual throughout can be low because of random requests from the same process, which can be tackled in the program optimization and thus is not in the scope of this work, or because of low I/O demand. Our method is to use disk simulators to replay the I/O traces collected in a profiling run of an MPI program. Simulations are run with the traces against two data layouts. The first one is the original data layout, in which files are uniformly striped over the I/O nodes, and the second one is the layout created by applying our data replication strategy for interference removal. Then the severity of an interference

can be defined as the potential I/O performance improvement from the interference removal strategy, quantified by the simulation results. That is, the higher performance improvement our strategy can potentially achieve, the higher interference with the original striping layout. The actual data replication is carried out only when the estimated improvement is larger than a pre-defined threshold and thus deemed as cost effective.

The I/O behaviors of an MPI program can be profiled by running it and collecting traces of its I/O requests, which are used to drive the simulators. As we aim for high portability by avoiding the involvement of any software on the I/O nodes and any modification of OS or file systems, we do not directly measure the arrival times or the arrival order at an I/O node. Instead, we record the times for requests to send and to finish observed at each process by modifying the MPI-IO library. However, the times recorded by each process are not comparable, if they run on different compute nodes and their clocks are not synced. For this reason, we run a utility program, which obtains the clock differences among the compute nodes by running an MPI barrier statement. Though the measurements of the differences have errors, they are sufficiently accurate to estimate I/O service times, especially for the times involved in an interference, which are usually much larger than the errors. With these differences, the times in the I/O traces collected for each process are accordingly adjusted to a consistent clock time so that we can merge the traces into one global trace, which records request send and finish events, sorted by their adjusted times, for the entire program. For each request event of a process, the accessed file, file offset, and data size are recorded for the request. As we know how a file is striped over the I/O nodes, including the start striping node, striping unit size, and striping depth, we can derive the I/O trace for every I/O node from the global trace, including process ID (or rank in the context of MPI programs), file offset, and data size local to the I/O node for each request event that this I/O node is involved in.

For the simulation of the performance improvement by our data replication strategy, which will be described in the next section, we use DiskSim [22], a widely used disk simulator producing accurate request response time. DiskSim usually takes a stream of requests, and for each request it takes the location of requested data on the disk (Logic Block Number or LBN) and request size as its input to calculate its disk service time. We assign block contiguous in the logic file space with contiguous LBNs, which is consistent to the disk space allocation policies used in most file systems. The improvement ratio *improvement_ratio* for an I/O node is calculated as the ratio of the service time of this I/O-node's trace on the data layout before replication (*time_before_replica*) and the service time of this I/O-node's trace on the data layout after replication (*time_after_replica*). These two times are derived from the statistics produced by the simulator as follows .

$$time\_before\_replica = \sum_{i=1}^{n}[service\_time(req_i) + disk\_speed\_factor * IO\_idle\_time],$$

$$time\_after\_replica = \max_{j=1}^{m}(\sum_{i=1}^{n_j}[service\_time(req_i, node_j) + disk\_speed\_factor * IO\_idle\_time_j]),$$

$$improvement\_ratio = \frac{time\_before\_replica}{time\_after\_replica}.$$

In the equations, n is the number of requests in the trace, n_j is the number of requests that are re-directed to a particular I/O node j due to data replication, and usually belong to one or a small number of processes. m is the number of I/O nodes. $service\_time(req)$ is the service time of request req before replication and $service\_time(req, node)$ is the service time of request req after replication of its data to $node$. Both times are calculated by the simulator. The requests in a trace are fed into the simulator in the order of their finish times recorded in the trace, reflecting the order for the requests to be processed in the real system. IO_idle_time is the total period of times in the trace that are not covered by any request's processing period, or the period from the request's finish time to the send time of the request next to it. Similarly, $IO\_idle\_time_j$ is the period of idle I/O time for requests re-directed to I/O node j. As we do not assume any a priori knowledge of the disks in the real I/O nodes in the configuration of DiskSim simulator for the sake of portability, the service time produced by DiskSim can be inflated or deflated compared with actual service time recorded in the trace. We take the ratio of these two service times as $disk\_speed\_factor$ and use it to correspondingly adjust the idle times in the formulas. It is noted that we assume one disk in one I/O node in the description and in our experiment setting, and run one DiskSim instance for one I/O node. It is straightforward to extend it to I/O nodes with multiple disks. For the pre-defined threshold improvement ratio, an estimated ratio larger than the threshold indicates an interference. Otherwise, it means that no interference is detected.

To accommodate the scenario where interference appears only during certain periods of an MPI-program's execution, we divide equally a program's total execution period into a sequence of time windows, and carry out the aforementioned interference detection for each window using traces of requests serviced in the corresponding window. The default window size in our experiments is one second.

3.2 Replication of Selected File Segments

For every execution window in which an interference is detected, which hints that an actual deployment of the data replication strategy probably receives sufficiently high I/O performance improvement, we carry out the replication for the window. Figure 2 shows an example scenario: (a) an original data layout on four I/O nodes, where the interference has been detected, and (b) the set of data are replicated on the selected I/O nodes to separate requests from different processes to different nodes. To ensure a truly effective replication, we need to decide effectively what data to replicate and where to replicate.

While a file is striped over I/O nodes with the striping unit size, we partition a file evenly into a sequence of regions whose size equals to the striping unit size, so that each region is entirely stored in one I/O node. The region size in our experiments is 64KB. Then we count the number of times for which a region has been accessed in a time window when interference is detected. A region whose count is larger than a threshold value is deemed as the one contributing to the interference and will be replicated. The threshold is the ratio between a base threshold value and the window's estimated improvement ratio from the simulation. That is, a higher potential improvement ratio would probably lead to more

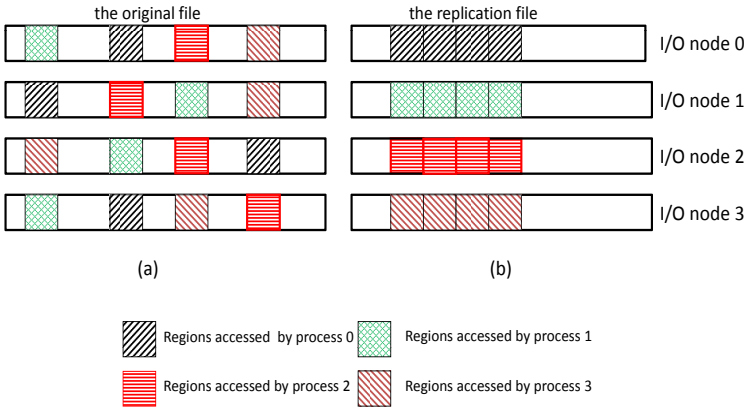

Figure 2: (a) The example data layout on four disks, each in one I/O node. A disk has four data regions, each accessed by a process. Concurrently serving requests from the processes causes disk head thrashing. Therefore, the data regions are selected for replication. (b) After replication, the regions accessed by one process are replicated contiguously on the designated home node of the process.

replicated regions, as it indicates a more serious interference. In our experiments the default base threshold is 100.

As the interference concerned in the work is due to the random arrival order of requests from different processes, we attempt to make replicas for selected file regions on other I/O nodes so that only requests from one or a small number of processes would be served at a node, eliminating the root cause of the interference. Accordingly, the I/O node to which a selected region is replicated, or the home node of the region, is determined by the process that sends requests for the data in the region. Specifically, the home I/O node number is the process rank moduloed by the number of I/O nodes. If a file region is accessed by more than one process in a window, we use the rank of the process that sends the largest number of requests to the region.

At each I/O node we create a replication file, in which replicas of selected file regions that take this node as its home I/O node are stored. The order for the regions to be laid out in the file depends on their smallest finish times among requests for the data of the region in the window, to minimize disk seek time. A region will not be replicated again if it has been replicated in a previous window. Figure 2 (b) shows the data layout in the replication file for the scenario we illustrated.

An important implication of the replication strategy is that the replication operations for all the I/O nodes in a time window must be coordinated. That is, either all nodes do the replications or not. Otherwise, the ones that do the replications and then re-direct their requests to other nodes may simply let the re-directed requests interfere with the requests served on the other nodes and essentially shift their interferences to others. Therefore, when we find that interference is detected at more than half of I/O nodes in a window, all nodes do the replication. Otherwise, none takes the action. After replication, the locations of replicas are recorded in a mapping table, in which all replicas produced after the profiling run are recorded.

3.3 Request Scheduling and Management of Replica Consistency

After a profiling run, the mapping table is stored in a file in the same directory as the MPI program. We modify the MPI library so that the mapping table is loaded with MPI_Init() and is unloaded with MPI_Finalize(). The mapping table entries are hashed in memory for efficient table lookup. With the data that could be involved in the interference replicated, an I/O request from a process of an MPI program will be re-directed to the process's home I/O node, if its requested data can be found there according to the mapping table. If the write request is re-directed, the corresponding table entry is marked as dirty, showing that original copy now is obsolete. To maintain consistency of file regions with replicas, we direct all read and write requests for data in these regions to their replica copies. In this way, we guarantee that an obsolete original copy of the region will not be read within a program's run. Furthermore, the only possible inconsistency among in-memory mapping tables for different processes is the dirty flag for a table entry, which is not used until we need to update the original copies of the regions.

At the end of an MPI-program's execution, each process writes back its dirty entries to the program's mapping table file. As each profiled MPI program has its mapping table file, a dirty entry in one table invalidates all replicas recorded in other tables. For this reason, we maintain a global mapping table file in the system, which tracks all replicas, as well as its validity and dirtiness states, of any file regions that have been replicated in any programs' profiling. When a program's mapping table is written back, the global table file will be accordingly updated. Similarly, before a mapping table is loaded from its file, we validate its entries against the global table. We also provide tools to maintain the consistency of replicas in an off-line manner, including updating the original copies for dirty replicas, validating a mapping table file against the global table file, and removing invalidated replicas.

4. PERFORMANCE EVALUATION

To evaluate the performance of *InterferenceRemoval* (IR in short hereafter), we set up a cluster with six compute nodes, six dedicated I/O nodes, and one metadata server of the parallel file system. All nodes are of identical configuration, each with dual 1.6GHz Pentium processors, 1GB memory, and a SATA disk (Seagate Barracuda 7200.10) with NCQ enabled. Each node runs Linux 2.6.21 with default CFQ I/O scheduler and uses GNU libc 2.6. The cluster is installed with the Lustre parallel file system version 1.6.6. We use MPICH2-1.1.1 [14], complied with ROMIO, to generate executables for MPI programs. All nodes are connected through a switched Gigabit Ethernet. The striping unit size, 64KB, is used to stripe files over six I/O nodes in the Lustre file system.

Major components of IR, including those for interference detection, file region replication, and request redirection based on the mapping table, are implemented in the MPI library, mostly in MPI-IO and ADIO libraries, through instrumenting MPI functions such as *MPI_Init(), MPI_Finalize(), MPI_File_read(), MPI_File_write(), MPI_File_read_all()*, and *MPI_File_write_all()*. There are two instrumented libraries, one for profiling run, another for actual run of the MPI program. According to the purpose of a particular run, different library is linked to generate the executable for the run. For each profiled MPI program, there is a replica file on each I/O node to store replicated regions. The file is in a directory dedicated for replica files. We set the striping attributes for the directory so that the files in it are not striped over more than one I/O node. IR also includes a set of off-line tools to manage consistency between runs of a program and that between runs of different programs, which has been briefly described in Section 3.3.

We choose three widely used MPI-IO benchmarks to evaluate the IR scheme, namely *BTIO* from the NAS parallel benchmark suite [15], *ior-mpi-io* from the ASCI Purple benchmark suite developed at Lawrence Livermore National Laboratory [8], *mpi-tile-io* from the Parallel I/O Benchmarking Consortium at Argonne National Laboratory [12]. We use different input files for profiling runs and for real runs in the experiments.

4.1 General Performance

4.1.1 Benchmark *BTIO*

BTIO is an MPI program designed to solve the 3D compressible Navier-Stokes equations using MPI-IO library for its on-disk data access. We choose to run the program with an input size coded as B in the benchmark, which generates a data set of 1.5GB. The program can be configured to use either non-collective or collective I/O functions for its I/O operations. We profile the executions of *BTIO* using either non-collective I/O or collective I/O with 4, 9, or 16 processes, generating region replicas for detected interferences. Then we run the program with IR using the replicas. We compare the program's I/O throughput using IR with that accessing original striped data in Figure 3.

Let us first look into the improvements by IR for non-collective version of *BTIO*. The throughput is improved by 6X, 26X, and 33X, for program runs with 4, 9, and 16 processes, respectively. These improvements are dramatic, especially with runs using a larger number of processes. As we know, each process of the program periodically issues a number of small sequential write requests and then reads the data back (request size is mostly around 170Bytes). To illustrate the interference on a particular I/O node, we show the file positions (offsets) that are allocated at I/O node 6 and are accessed by each of the nine processes in a 9-process run during a five-second execution period, without IR or with IR, in Figure 1(a) and (c). Without using IR, there are at least four different file ranges that are concurrently touched by the processes at a single disk. From which range the next request would ask for its data depends on which process issues the next request, which is apparently unpredictable. The consequent random data access on a disk causes the disk head thrashing, or interference among requests from the processes. The interference gets worse with the increasing number of processes. By applying IR, node 6 only serves requests from process 6, from low LBN addresses to high LBN addresses, without any interference.

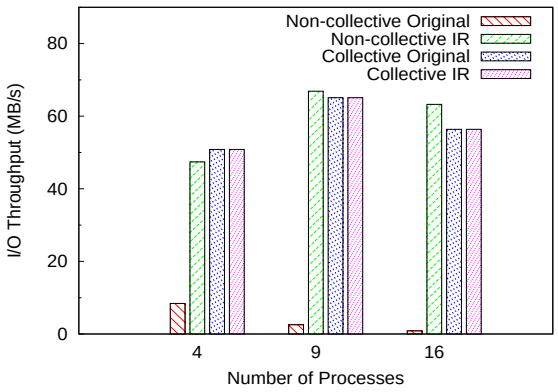

Figure 3: I/O throughputs of the collective-I/O version and non-collective-I/O version of benchmark *BTIO* when they are executed without IR (Original) or with IR, and with different number of processes.

From figure 3 we can observe that the program's throughput with collective I/O performs much better than its non-collective version. Through profiling we find that the size of collective-I/O requests is around 40MB, much larger than size of requests with non-collective I/O. This makes data accessed at one disk for one request nearly as large as 7MB, which is large enough to amortize the cost of a disk head seek and to make the disk work efficiently. Accordingly, IR for the collective-I/O version cannot detect any interference and does not make any replicas. Its throughput is the same as that without using IR, showing that the additional time overhead of IR is minimal. Furthermore, the throughput of the non-collective-I/O version with IR is higher than that of the collective-I/O version without IR for large numbers of processes (9 and 16).

4.1.2 Benchmark *ior-mpi-io*

In benchmark *ior-mpi-io*, each of the m MPI processes is responsible to read or write $1/m$ of a file whose size is 8GB. Each process issues continuously sequential requests, each for a 64KB segment. If collective I/O is used, the processes' requests for data at the same relative offset in each process's access scope are organized into one collective-I/O function call. Figure 4 shows the throughput of two versions of *ior-mpi-io*, non-collective I/O and collective I/O, with IR

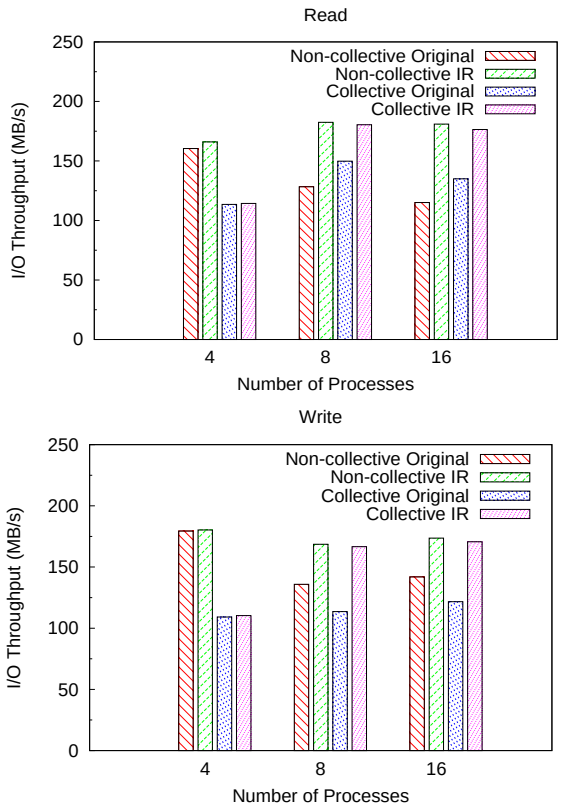

Figure 4: I/O throughputs of the collective-I/O version and non-collective-I/O version of benchmark *ior-mpi-io* when they are executed without IR (Original) or with IR and with different number of processes. The I/O requests are designated either as *read* or as *write*.

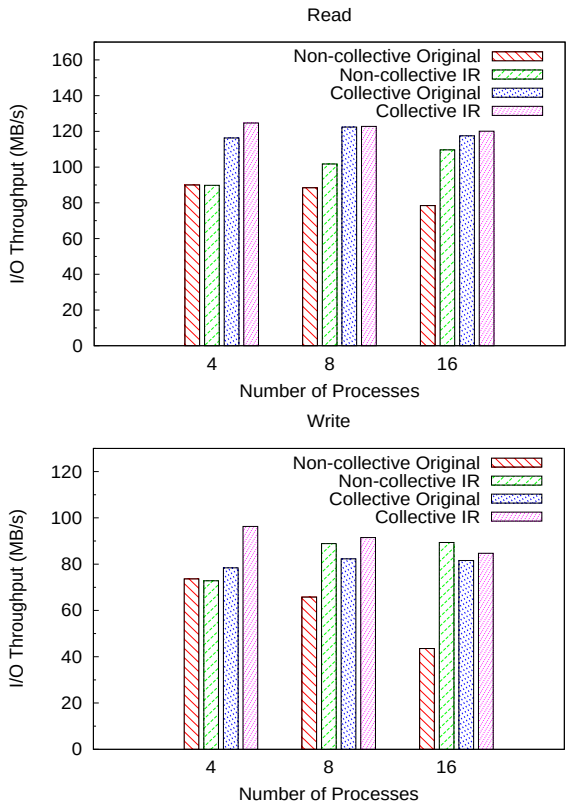

Figure 5: I/O throughputs of the collective-IO version and non-collective-I/O version of benchmark *mpi-tile-io* when they are executed without IR (Original) or with IR, and with different number of processes. The I/O requests are designated either as *read* or as *write*.

or without IR, for different number of processes. The program's I/O can be designated as either *read* or *write*. The throughput for both *read* and *write* are shown in Figure 4. Compared with *BTIO*, the improvement for *ior-mpi-io* with IR is smaller, especially when the number of processes is small. In *ior-mpi-io*, the request size is larger (64KB), and each process accesses contiguous data in its access scope. When the process count is four, the interference is small enough to be deemed as not cost-effective to replicate file regions by IR. Therefore, there is only a few or no regions replicated in the case. When the process count is 8 or 16, interference becomes intensive at each I/O node, which is detected and removed by IR. The I/O throughput improvement ranges from 20.8% to 56.5%. Interestingly, for the original *ior-mpi-io* (without using IR) with four processes, using collective-I/O adversely reduces I/O throughput significantly because of the unbalanced I/O load, rather than interference. Therefore, IR does not help fix the problem.

4.1.3 Benchmark *mpi-tile-io*

Benchmark *mpi-tile-io* uses MPI processes to read or write a file in a tile-by-tile fashion, with two adjacent tiles partially overlapped. Each process accesses 4MB, with 2KB of overlap between two consecutive accesses. Figure 5 shows the I/O throughput of the two versions of the program, non-collective I/O and collective I/O, when we increase the number of processes from 4, 8, to 16, without using IR or with IR enabled. The results of using non-collective I/O show that the I/O throughput is reduced with the increasing number of processes, indicating increasingly severe interference among requests from a larger number of processes at each I/O node. In contrast, the runs with IR consistently increase the throughput, up to a 105% increase over its counterpart. With increasing process count, IR maintains high throughput by removing interference and taking advantage of higher process parallelism. This is accompanied with 0%, 30%, and 60% of the file regions replicated with process count of 4, 8, and 16, respectively. From figure 5 we can also observe that the program with collective I/O achieves higher throughput than its non-collective I/O version because of the increased size of collective I/O request (32KB). As a result, the performance improvement is marginal by using IR for the program with collective I/O optimization.

4.2 Impact of the Replicated Data Amount on IR's Effectiveness

In this section, we study how the effectiveness of IR depends on the amount of data replicated, or how IR is cost effective. As we described in Section 3, a base threshold improvement ratio is used to determine how large an estimated

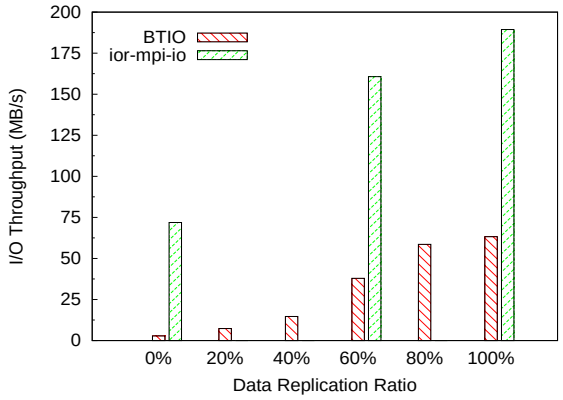

Figure 6: I/O throughputs of programs *BTIO* and *ior-mpi-io* with different data replication ratios.

I/O improvement ratio, obtained from the simulations, has to be for an interference to be detected. The larger the base threshold value is, the more difficult for an interaction among requests to be identified as interference. In other words, the higher the threshold, the smaller the data replication ratio, which is defined as the percentage of total accessed file regions that have been replicated. We incrementally change the base threshold improvement ratio to get the relationship between the amount of replicated data and the performance improvement. Figure 6 shows the relationship for programs *BTIO* and *ior-mpi-io*. For *BTIO* benchmark, we use nine processes and its non-collective version. For *ior-mpi-io*, we use 16 processes and its non-collective version. In the figure, the throughputs for 0% are actually the ones for the programs' runs without using IR. The results show that generally high replication ratio leads to higher I/O performance improvement. However, not all replication ratios are available, such as 20%, 40%, and 80% for *ior-mpi-io* in the figure. This is because not all estimated I/O improvement ratios are available, as shown in Figure 7. Figure 7 shows estimated throughput improvement ratios for each window in the programs' executions. As the ratios are not evenly distributed across the space, especially for *ior-mpi-io*, we have either little replication or more than half of file regions replicated when we increase the base threshold improvement ratio. This indicates that in some scenarios, a relatively high cost is needed to enable IR. For *BTIO*, we see that the replication ratio, as well as the throughput, keeps increasing when we continuously reduce the threshold.

4.3 The Effectiveness of IR with I/O Interference from other Programs

As IR is intended to identify and remove interference for one MPI program, there may be interference caused by competing I/O requests from other programs concurrently running in the system. To see the impact of this interference on the effectiveness of IR, we choose *S3aSim*, a program widely used in the computational biology for sequence similarity search [21], to keep generating interfering I/O requests in background. The program's I/O intensity can be adjusted by setting its *compute speedup* parameter. A larger compute speedup produces a higher I/O intensity. In the experiments we test three speedup values (1, 1.5, and 2), where each value roughly doubles the I/O intensity associated with its

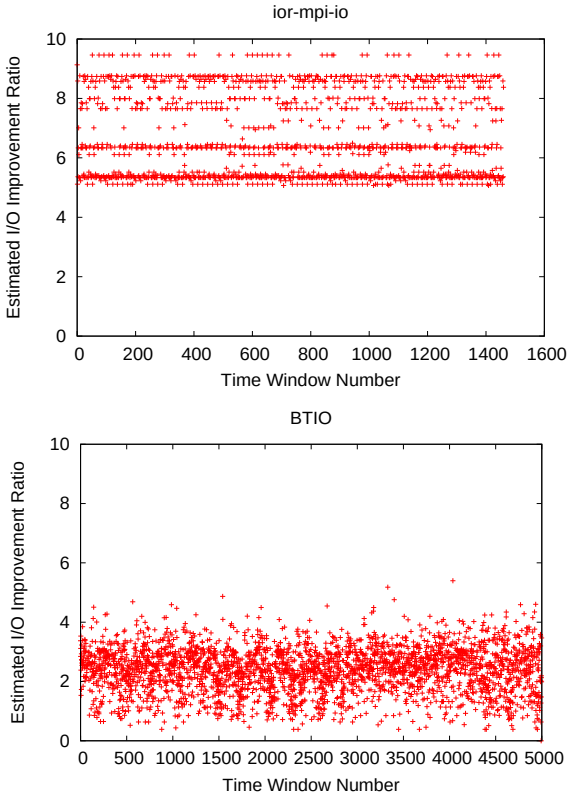

Figure 7: Estimated I/O improvement ratios obtained from the DiskSim simulator for each execution window in running programs *ior-mpi-io* and *BTIO*.

preceding value.

Figure 8 shows the I/O read throughput of program *ior-mpi-io* with 16 processes using non-collective I/O. From the figure we can see that when I/O interference from the background program is low (speedup 1), both the run of the original program and the run with IR experience little performance loss, compared with the throughputs in their dedicated runs. However, as the interference from *S3aSim* keeps increasing, the throughputs of both runs are reduced as some of the I/O nodes' bandwidth is taken by *S3aSim*. However, the reductions for the runs with IR are apparently larger, to the extent that the performance improvement from IR disappears (for speedup 2). When the interference from the other program becomes dominant in affecting a disk's efficiency, removing interference within a single MPI program does not help improve this program's I/O performance. As IR is designed to remove the I/O interference within an MPI program, it is effective only when this type of interference is the major cause of low disk efficiency.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we describe the design and implementation of the *InterferenceRemoval* scheme to identify and remove I/O interference at each I/O node caused by serving requests simultaneously sent by multiple processes of an MPI program. This is achieved in principle by directing requests from different processes to different I/O nodes through replicating selected file regions. To justify the cost

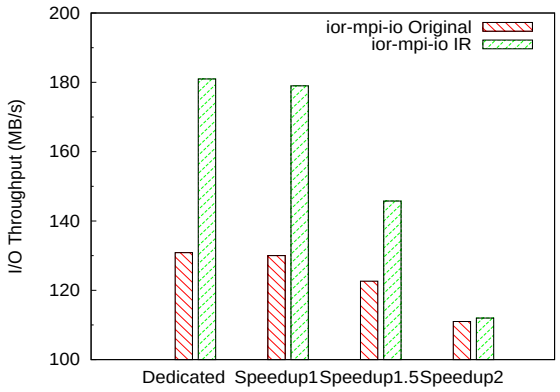

Figure 8: I/O throughputs of program *ior-mpi-io* without IR (Original) and with IR in their respective dedicated runs and runs under interference from another program of varying I/O intensity. A larger compute speedup represents a higher I/O intensity.

associated with the data replication, we replicate data only when a true interference is detected using accurate trace-driven simulations. *InterferenceRemoval* is implemented in the ROMIO MPI library. Our experimental evaluation of the scheme on top of the Lustre file system with representative benchmarks, such as NPB *BTIO*, and *mpi-tile-io*, shows that it can significantly improve I/O performance. However, as we have shown, currently the scheme is designed specific for one MPI program. When interference among concurrently running MPI programs dominates the efficiency of I/O nodes, a scheme to detect and remove this type of interference must be designed and applied for entire system's I/O performance. We plan to study the issue as one of our future work, possibly by extending some methods proposed in this paper.

6. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments. This work was supported by US National Science Foundation under grants CCF 0702500 and CAREER CCF 0845711.

7. REFERENCES

[1] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis, "BORG:Block-reORGanization for Self-optimizing Storage Systems", *In Proceedings of the 7th USENIX Conference on File and Storage Technologies* , San Fancisco, CA, 2009.

[2] A. Ching, A. Choudhary, W. Liao, R. Ross, and W. Gropp, "Efficient Structured Data Access in Parallel File System", *In Proceedings of IEEE International Conference on Cluster Computing*, Hong Kong, China, 2003.

[3] A. Ching, A. Choudhary, K. Coloma, and W. Liao, "Noncontiguous I/O Accesses Through MPI-IO", *In Proceedings of IEEE International Symposium on Cluster, Cloud, and Grid Computing*, Tokyo, Japan, 2003.

[4] Cluster File Systems, Inc. Lustre. "Lustre: A scalable, robust, highly-available cluster file system", http://www.lustre.org/. Online-document, 2010.

[5] H. Huang, W. Hung, and K. Shin, "FS2: Dynamic Data Replication in Free Disk Space for Improving Disk Performance and Energy Consumption", *In Proceedings of ACM Symposium on Operating Systems Principles*, Brighton, UK, 2005.

[6] W. Hsu, A. Smith, H. Young, "The Automatic Improvement of Locality in Storage Systems", *ACM Transactions on Computer Systems*, Volume 23, Issue 4, Nov. 2006, Pages 424-473.

[7] W. Hsu, A. Smith, H. Young, "The Automatic Improvement of Locality in Storage Systems", *Technical Report* CSD-03-1264, UC Berkeley, Jul. 2003.

[8] Interleaved or Random (IOR) benchmarks, http://www.cs.dartmouth.edu/pario/examples.html, Online-document, 2008.

[9] S. Iyer and P. Druschel, "Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O", *In Proceedings of ACM Symposium on Operating Systems Principles*, Banff, Canada, 2001.

[10] D. Kotz, "Disk-directed I/O for MIMD Multiprocessors.", *ACM Transactions on Computer Systems*, Volume 15, Issue 1, Feb. 1997, pages 41-74.

[11] S. Liang, S. Jiang, and X. Zhang, "STEP:Sequentiality and Thrashing Detection Based Prefetching to Improve Performance of Networked Storage Servers.", *In Proceedings of International Conference on Distributed Computing Systems*, Toronto, Canada, 2007.

[12] Mpi-tile-io Benchmark, http: www-unix.mcs.anl.gov/thakur/pio-benchmarks.html. Online-document, 2009.

[13] M. Kandemir, S. Son, M. Karakoy, "Improving I/O Performance of Applications through Compiler-Directed Code Restructuring", *In Proceedings of the 6th USENIX Conference on File and Storage Technologies*, San Jose, CA, 2008.

[14] MPICH2, Argonne National Laboratory, http://www.mcs.anl.gov/research-/projects/mpich2/. Online-document, 2009.

[15] NAS Parallel Benchmarks, NASA AMES Research Center, *http://www.nas.nasa.gov/Software/NPB/*. Online-document, 2009.

[16] PVFS, http://www.pvfs.org. Online-document, 2010.

[17] P. Pacheco, "Parallel Programming with MPI", *Morgran Kaufmann Publishers*, pages 137-178, 1997.

[18] K. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett, "Server-directed collective I/O in Panda", *In Proceedings of Supercomputing*, San Diego, CA, 1995.

[19] F. Schmuck and R. Haskin, "GPFS:A shared-disk file system for large computing clusters.", *In Proceedings of the 1st USENIX Conference on File and Storage Technologies*, Monterey, CA, 2002, Monterey, CA, USA.

[20] R. Thakur, W. Gropp and E. Lusk, "Data Sieving and Collective I/O in ROMIO", *In Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation*, Annapolis, MD, 1999.

[21] S3aSim I/O Benchmark, http://www-unix.mcs.anl.gov/thakur/s3asim.html. Online-document, 2009.

[22] The DiskSim Simulation Environment(v4.0), Parallel Data Lab, http://www.pdl.cmu.edu/DiskSim/. Online-document, 2009.

[23] Y. Wang and D. Kaeli, "Profile-Guided I/O Partitioning", *In Proceedings of International Conference on Supercomputing*, San Fancisco, CA, 2003.

[24] C. Wang, Z. Zhang, X. Ma, S. Vazhkudai, and F. Mueller, "Improving the Availability of Supercomputer Job Input Data Using Temporal Replication", *In Proceedings of International Supercomputing Conference*, Hamburg, Germany, 2009.

[25] 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 IEEE International Parallel & Distributed Processing Symposium*, Rome, Italy, 2009.