IOrchestrator: Improving the Performance of Multi-node I/O Systems via Inter-Server Coordination

Xuechen Zhang
The ECE Department
Wayne State University
Detroit, MI 48202, USA
Email: xczhang@wayne.edu

Kei Davis
CCS Division
Los Alamos National Laboratory
Los Alamos, NM 87545, USA
Email: kei.davis@lanl.gov

Song Jiang
The ECE Department
Wayne State University
Detroit, MI 48202, USA
Email: sjiang@eng.wayne.edu

***Abstract*—A cluster of data servers and a parallel file system are often used to provide high-throughput I/O service to parallel programs running on a compute cluster. To exploit I/O parallelism parallel file systems stripe file data across the data servers. While this practice is effective in serving asynchronous requests, it may break individual program's spatial locality, which can seriously degrade I/O performance when the data servers concurrently serve synchronous requests from multiple I/O-intensive programs.**

In this paper we propose a scheme, *IOrchestrator*, to improve I/O performance of multi-node storage systems by orchestrating I/O services among programs when such inter-data-server coordination is dynamically determined to be cost effective. We have implemented IOrchestrator in the PVFS2 parallel file system. Our experiments with representative parallel benchmarks show that IOrchestrator can significantly improve I/O performance—by up to a factor of 2.5—delivered by a cluster of data servers servicing concurrently-running parallel programs. Notably, we have not observed any scenarios in which the use of IOrchestrator causes substantial performance degradation.

***Index Terms*—Spatial Locality, Synchronous Requests, and PVFS2.**

I. Introduction

Parallel programs are becoming increasingly data intensive. As an example, in its analysis of astronomical data, the *astro* program generates more than 50GB data and 62% of its total execution time is attributed to disk I/O operation in one run [16]. To provide adequate I/O support parallel file systems such as PVFS2 [25], [20], Lustre [18], and GPFS [26] exploit the natural parallelism provided by a shared cluster of data servers by striping file data over them. A parallel file system allows requests from a program running on the compute nodes to be served by multiple data servers in parallel. However, when the server cluster is a shared resource—the usual case—it must concurrently serve requests from multiple programs. While requests from multiple programs help increase workload concurrency and keep data servers busy, it can also reduce hard disk efficiency by compromising programs' spatial locality.

A. Spatial Locality and Hard Disk Performance

The hard disk is still the most cost-effective mainstream storage device, but the spatial locality of its accesses dramatically affects its performance. Spatial locality is the property of a sequence of accesses (or of requests for those accesses, or of a program that generates those requests) to a particular storage medium for data that are close to each other. Data on a hard disk are accessed using moving disk heads and rotating disk platters, and sequential access can be more than an order of magnitude faster than random access.

A challenge in exploiting spatial locality is that many requests with good spatial locality are synchronous. For synchronous requests, a process will not issue its next request until its last request is served. Programmers generally prefer to use synchronous requests over asynchronous ones because it is simpler to manage control flow with synchronous function calls. However, when multiple programs, each with good spatial locality, concurrently issue synchronous requests to the same disk, the result can be severe disk head thrashing that cripples performance. To preserve the spatial locality of synchronous requests from one process when multiple processes are simultaneously issuing requests, schedulers such as the Anticipatory Scheduler (AS) [11] and Completely Fair Queuing (CFQ) [2] are used in many high-performance computing installations. These schedulers are predicated on the assumption that there will be no more than a small time interval (*think time*) between synchronous requests from a given process, that these requests are likely to have good spatial locality, and data requested by other processes will be remote on the disk. For this to be advantageous the think time must be short enough and the locality of the process must be strong enough that the benefit of serving next request from the same process in a non-work-conserving fashion is greater than the cost paid for idle waiting.

B. Spatial Locality with Multi-node I/O Systems

The AS and CFQ schedulers have proven effective at preserving the spatial locality exhibited by individual processes,

SC10 November 2010, New Orleans, Louisiana, USA 978-1-4244-7558-2/10/$26.00

but their effectiveness is limited to the case where the process's requested data reside on a single disk. When file data are striped over multiple disks or multiple data servers, these schedulers are often unable to exploit individual processes' spatial locality. The key reason is that in a multi-disk system it is not solely the process's think time that determines how soon the process's next request to a given disk will arrive. We refer to the time period between two requests from a process that hit a given disk as *the reuse distance of the disk by the process*. When file data are striped in a multi-disk system the reuse distance can become so large that it is not profitable for the disk to wait for a process's subsequent request. This is a direct consequence of striping—sequential contiguous requests wrap around the disks or data servers. Even if the disks, or data servers, whose service times contribute to the reuse distance, are synchronized to provide dedicated service to the process, the distance can be still too long for the disk head to wait, instead of leaving for requests from other processes. Consequently, each disk may end up thrashing its disk head among processes, breaking spatial locality in the processes. The potential I/O performance advantage from spatial locality thus gets lost in the larger-I/O-system behavior.

C. Preserving Spatial Locality for Parallel Programs

Schedulers' inability to exploit spatial locality poses an especially serious problem for I/O-intensive parallel programs. These programs usually rely on strong spatial locality to ensure high I/O performance. To this end, techniques such as collective I/O [28] and data sieving [28], have been widely used to help form large contiguous accesses. In addition, a common practice for coordinating computation and I/O is to use synchronization, such as *barrier*, between I/O requests in a parallel program. Thus, the synchronization separates the I/O operations into distinct time regions and makes the requests issued in the same time slot related to the same computation, which helps improve spatial locality. However, the locality created by these techniques is usually only from the perspective of the program. I/O requests are still sent simultaneously from a number of processes of the running program (e.g., collective I/O for MPI programs). It would still be hard for a data server to exploit the spatial locality of individual processes because the reuse distance of any data server by a process could still be too large.

To more effectively discuss spatial locality as observed by such techniques as collective I/O and *barrier*, we introduce the notion of reuse distance at each data server *by a parallel program*, which is the time period between two requests from the same running program that hit a data server. Because the parallel program consists of multiple processes and the requests from these processes usually have a relatively strong spatial locality, the reuse distance by a program may be much shorter than the reuse distance by an individual process. Therefore, it can be profitable for the disk head to wait for the next request from the same program.

In this work we propose a scheme, *IOrchestrator*, that orchestrates the serving of requests from multiple programs over a set of data servers so that the reuse distance of programs can be minimized alternately to exploit the spatial locality of each, when sufficient spatial locality exists. Specifically, we made the following contributions.

- We propose methods to measure reuse distance for programs and to quantitatively evaluate whether it is cost-effective to dedicate all involved data servers to a program to exploit its spatial locality.
- We design algorithms, comprising IOrchestrator, to coordinate request scheduling across data servers according to monitored programs' access behaviors so that useful spatial locality is exposed and efficiently exploited.
- We implement these algorithms in the PVFS2 parallel file system with minor support from the operating system and the MPI library. We evaluate it with representative benchmarks, including *mpi-io-test*, *ior-mpi-io*, *hpio*, and *mpi-tile-io*. Experimental measurements show that I/O throughput can increase by up to 2.5 times, and 39% on average, compared to the vanilla system for these benchmarks.

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

II. Related Work

We review the research literature, mainly on high-performance computing, in two areas: (1) how to improve spatial locality for disk-based storage systems from two perspectives, i.e., modification of request streams and rearrangement of on-disk data layout; and, (2) recovering lost spatial locality when running multiple processes/programs.

A. Modifying Request Streams for Greater Spatial Locality

To improve I/O performance for data-intensive parallel applications, researchers have expended much effort on developing I/O middleware to transform a large number of small non-contiguous request into a smaller number of larger contiguous requests. Data sieving [28] is one such technique wherein instead of accessing each portion of the data separately, a larger contiguous chunk that spans multiple requests is read. If the overhead for accessing additional unneeded data, called holes, is not excessive, its benefit can be significant.

Datatype I/O [5] and list I/O [6] are the two other techniques that allow users to access multiple non-contiguous data using a single I/O routine. Datatype I/O is used to access data with certain regularity, while list I/O is designed to handle more general cases. Considering the data accesses across processes, collective I/O [28] was proposed to enable optimization in a greater scope in comparison to those techniques applied individually in each process. It rearranges the data accesses collectively among a group of processes of a parallel program so that each process has a larger contiguous request. While collective I/O can incur communication overhead because of data exchange among processes, its performance advantage is well recognized, making it one of most popular I/O optimization

techniques for MPI programs. Observing that current ROMIO implementations of collective I/O [28] can cause requests to arrive at each data server in an order inconsistent with data placement, resonant I/O was proposed as an enhanced implementation of collective I/O to restore spatial locality [34].

While these techniques can be effective in enhancing spatial locality, the locality may not be translated into high I/O performance when shared I/O systems are concurrently serving requests from multiple programs. By orchestrating requests' service on different data servers IOrchestrator can better exploit the locality, resulting in higher I/O throughput.

B. Rearrangement of On-disk Data Layout for Greater Spatial Locality

Spatial locality can also be improved by rearranging data layout on disk to create a layout consistent with expected request patterns. This can be achieved through data relocation [9] or data replication. With data replication, some actively used data would have multiple copies on the disk and the copy that is closest to the disk head is accessed. Replication can be carried out within one disk [10], [3], [15] or across disks [33], [30]. The effectiveness of this method relies on two factors: a stable and predictable access pattern to know where to relocate or replicate data; and, a relatively small on-disk working set so that the replication overhead is not excessive. However, in a parallel system where requests are concurrently issued by multiple processes of a program and multiple running programs, the uncertainty in their relative execution speed can make stable access patterns hard to detect. For data-intensive programs that process huge amounts of data this approach becomes impractical. In contrast, IOrchestrator does not copy or move user data on the data servers, so it does not have the concern of potentially high overhead.

C. Recovering Spatial Locality Lost when Running Multiple Processes/Programs

The weakened or even lost spatial locality with concurrent servicing of interleaved requests issued by multiple processes can be recovered by non-work-conserving disk schedulers, such as the AS scheduler. Here the disk head is kept idle after serving a request of a process until either the next request from the same process arrives or the wait threshold expires. Anticipatory scheduling is implemented in some popular Linux disk schedulers [21]. However, a disk on a data server is not likely to see the next request soon when file data are striped over multiple data servers. Consequently, the disk scheduler on the data server would choose to serve requests from other processes and precipitate disk head thrashing. This problem may be replicated on all the data servers in the system. In this sense, IOrchestrator may be viewed as a non-work-conserving request scheduler for an array of data servers.

D. Our Work in Context

By coordinating the servicing of requests from different programs it is possible to reduce the time gap between two requests from the same program to the extent that spatial locality of a program is worth exploiting. IOrchestrator is designed to exploit such spatial locality for eligible programs, which will be defined in Section III.C, by coordinating scheduling at different data servers. A technique known as co-scheduling was first applied to synchronize CPU scheduling of processes of a parallel program on multiple nodes of an HPC cluster so that the overhead of CPU context switching could be reduced [24], [8]. A similar idea was used for disk spindle synchronization in a disk array to reduce platter rotation time in serving small requests [17]. Researchers also found that the communication latency among cluster-based webservers can be reduced by co-scheduling accesses to remote caches rather than mixing the accesses to cache and the disk together when there is a sufficient time difference between these two kinds of accesses [13]. Wachs et al. proposed timeslice co-scheduling for cluster-based storage [31]. The objective of this latter work is better performance insulation quantified by *R-value* [32] while meeting user-specified QoS requirements. Though their work is similar to ours in the coordination of some or all disks and dedication of them to one process at a time, it cannot be effectively used as a solution in the context of the data servers managed by parallel file systems. One reason is that their work requires an offline-calculated scheduling plan according to QoS specifications that does not adapt to the workload dynamics. Another reason is that it does not evaluate the benefits of dedicated service to a program relative to the cost of disk synchronization, and indiscriminately applies the synchronization to all programs. In contrast, IOrchestrator dynamically evaluates the cost-effectiveness of synchronization and opportunistically allows the data servers to provide dedicated service to one program at a time.

III. Design and Implementation

The design objective of IOrchestrator is to selectively recover spatial locality, in a parallel program, that is lost when the program runs together with other programs, all sharing a multi-node storage system. This is achieved by synchronizing data servers and dedicating them to one program at a time under the conditions that (1) adequate spatial locality exists in the program but gets lost due to co-running programs; and, (2) the data servers can be sufficiently well utilized to justify dedicated service. In dedicated service for a selected program, each data server would only serve requests from that program, keeping its disk head(s) idle even in the presence of pending requests from other programs. This approach could disrupt system performance if it were indiscriminately applied. To be effective, IOrchestrator tracks the spatial locality and reuse distances within each program, and that across programs, and continuously evaluates the cost-effectiveness of dedicated service. A program is selected for dedicated service only when it is expected to improve the system's I/O performance.

A. The IOrchestrator Architecture

We implemented IOrchestrator in the PVFS2 parallel file system. PVFS2 seeks to provide scalable, high-performance

I/O service for parallel programs using a cluster of data servers [25]. It has a metadata server for managing all file metadata for PVFS files, and a number of data servers on which PVFS files are striped. The PVFS file system is built on top of local file systems. That is, a PVFS file actually consists of a number of local files that are managed by local file systems. The metadata server records how a PVFS file is laid out on the data servers. A process running on a compute node first contacts the metadata server before it issues requests for data directly to the data servers.

One of our design objectives is to enable program-level I/O scheduling so that eligible programs can receive dedicated I/O service. To this end, we need to correlate the spatial locality and reuse distance detected at the data servers to the programs running at the compute nodes. However, this cannot be achieved within data servers. As we know, PVFS uses an *iod* daemon at each data server to receive I/O requests from processes on the client side and issue the requests to the kernel on behalf of the processes. Therefore, the local file system, which actually schedules requests to the disk, does not know which process or running program at the client side issued the requests. To evaluate spatial locality exhibited within a program and among programs on a data server, IOrchestrator needs this information. To achieve this IOrchestrator uses a daemon at the metadata server that is responsible for collecting information about which files have been opened by each program. This daemon, the program-files daemon (*pf* daemon), maintains the map between program names and file names. At the compute nodes, when a new MPI program is launched and its member processes are spawned, IOrchestrator sends unique identifiers for the running program (*job* in MPI) and its processes to the *pf* daemon.[1] We also instrument the MPI file-opening functions in the ROMIO library to report to the *pf* daemon when a file is opened by a particular process. Using the information from the compute nodes the *pf* daemon knows which files are opened by each program.

Because the metadata server knows how a PVFS file is striped over the data servers, the *pf* daemon at the server knows what local files at each data server are accessed by a particular running program and passes the information to a *locality* daemon at each relevant data server. The *locality* daemons are responsible for measuring the spatial locality among local files. Once the *locality* daemon knows the relationship between local files and programs, it can derive the spatial locality exhibited within and among PVFS files (detailed later) and passes the information to another daemon of IOrchestrator, the *orchestrator*, at the metadata server, that collects the information about spatial locality sent by each data-server's *locality* daemon. The *orchestrator* daemon identifies programs for dedicated service and creates the program-level scheduling plan. This plan is executed by the *iScheduler* daemon, which is actually the PVFS2 *iod* daemon at each data server with an added component for request scheduling required by IOrchestrator. The *iScheduler* daemon sits above the local disk scheduler, to which it relays requests. Figure 1 illustrates the architecture of IOrchestrator.

[1]In MPI, the information on the processes that are spawned in a job is recorded in file "*mpdlistjobs*".

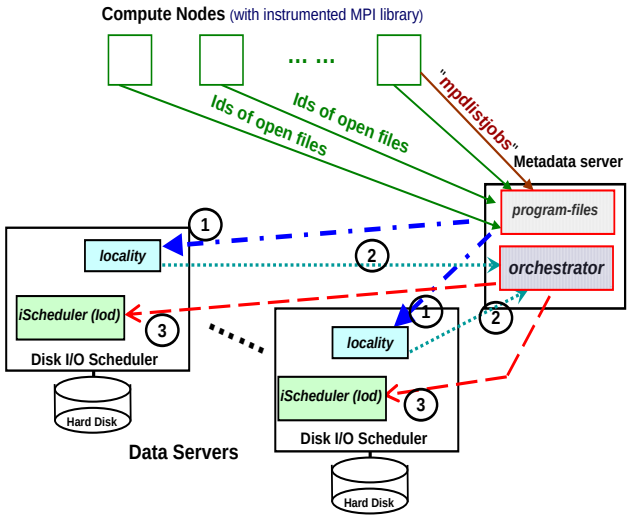

Fig. 1. The IOrchestrator software architecture: the *program-files* daemon collects the program-to-files mapping information from the compute nodes, and uses it to determine the program-to-local-file mapping information, which is passed to the *locality* daemons at the data servers (Step 1); the *locality* daemons collect locality and reuse distance statistics and pass them to the *orchestrator* daemon (Step 2); and the *orchestrator* daemon makes the scheduling plan and sends it to the *iScheduler* daemons at data servers to execute (Step 3).

B. Measuring Spatial Locality and Programs' Reuse Distance

While the spatial locality of a sequential program is solely a property of the program, reflecting its intrinsic access patterns, the spatial locality observed at each data server for a parallel program with a multi-node storage system is additionally determined by how processes run on the compute node and how file data are striped over data servers. In addition, it is the spatial locality of all the programs in the system that collectively determines the I/O efficiency of a data server. We denote the spatial locality observed at data server *i* for program *j* as SL_{ij} and the spatial locality observed at data server *i* for all programs as SL_i. For a particular program *j* running together with other programs, SL_{ij} may not be significant in determining the program's I/O efficiency unless it is given a dedicated time slice to access the data server.

There are two conditions for a time slice to be dedicated to a program *j* at data server *i* to be cost effective. The first condition is that SL_{ij} be substantially stronger than SL_i (a smaller value indicates a stronger locality; quantitative definitions are given below). The second condition is that the reuse distance of program *j* at data server *i*, denoted by RD_{ij}, is sufficiently small relative to a given SL_i. The first condition

ensures that efficiency can be improved by dedicating a time slice of the data server to the program. The second condition ensures that the cost paid for providing dedicated service to the program is justified. During a dedicated service period for one program, the concurrency of the workload on the storage system is reduced, and thus there is a higher probability for some disks to stay idle while requests from other programs are pending. To answer the question on whether a disk head should wait for the next request from the same program within an expected period RD_{ij} or take a time period $SeekTime_i$, determined by SL_i, to serve other programs, we adopt the approach described by Huang et al. [10], Section 4.2, to derive $SeekTime_i$ from SL_i.

To statistically quantify the locality (SL_i and SL_{ij}) and reuse distance (RD_{ij}), we use an approach that is similar to the one developed in Linux on anticipatory scheduling [19] for a similar purpose:

$$SL_i(k) = \frac{1}{8} * SL_i(k-1) + \frac{7}{8} * LBA\_Gap_i(k)$$

$$SL_{ij}(k) = \frac{1}{8} * SL_{ij}(k-1) + \frac{7}{8} * LBA\_Gap_{ij}(k)$$

$$RD_{ij}(k) = \frac{1}{8} * RD_{ij}(k-1) + \frac{7}{8} * ReuseDistance_{ij}(k)$$

where $SL_i(k)$ is SL_i when the kth request to data server i is served, $SL_{ij}(k)$ is SL_{ij} when the kth request from program j to data server i is served, and $RD_{ij}(k)$ is RD_{ij} when the kth request from program j to data server i is served. $LBA\_Gap_i(k)$ is the LBA gap between the $(k-1)$th and kth requests to data server i, and $LBA\_Gap_{ij}(k)$ is the LBA gap between the $(k-1)$th and kth requests from program j to data server i. The LBA of a request is the logical block address of the requested data, reflecting location of the data on the disk. $ReuseDistance_{ij}(k)$ is the time gap between the $(k-1)$th and kth requests from program j to data server i. In these formulas we consider both recent and historical statistics to smooth out short-term dynamics, and phase out historical statistics by giving recent statistics greater weight.

The *locality* daemon at each data server, obtaining request LBAs from the instrumented kernel, collects the various measurements and calculates these statistics. Among them, $SL_i(k)$, $SeekTime_i(k)$, and $SL_{ij}(k)$ for any program j are periodically sent to the *orchestrator* daemon at the metadata server. $RD_{ij}(k)$ is only reported for the program that is receiving dedicated service. At other times $RD_{ij}(k)$ should be significantly larger as it could include the time periods when the program's requests to other data servers are not scheduled. As mentioned, the *locality* daemon uses the information on the relationship between running program and local files, received from the *pf* daemon, to determine which requests belong to the same program, assuming files are not shared among different programs.

C. Scheduling of Eligible Programs

When the *orchestrator* daemon at the metadata server receives the statistics from the *locality* daemons, it uses the latest values of SL_i, $SeekTime_i$, SL_{ij}, and RD_{ij} to check three conditions to determine whether program j is eligible for a dedicated service, or whether it is an eligible program: (1) the standard deviations of SL_i and SL_{ij} ($i = 0, 1, ..., n-1$), where n is the number of data servers, are less than 20% of their respective mean values; (2) $(\sum_{i=0}^{n} SL_i)/\sum_{i=0}^{n} SL_{ij} \geq 1.5$; and (3), $(\sum_{i=0}^{n} RD_{ij})/n \leq SeekTime_i$. The first two conditions are used to ensure that the benefit to the program from a dedicated service is potentially substantial and consistent across the data servers. The threshold values (20% and 1.5) are set empirically and our measurements show that performance is not sensitive to them in a relatively large range in our experiments. (We leave a comprehensive study of their impact as future work.) The third condition is to ensure that the benefit of dedicated service is greater than its cost, and is only checked when dedicated service is granted to the program so that RD_{ij} can be reported.

If there are m running programs in the system that are identified as eligible programs, there are $m+1$ scheduling objects for the *orchestrator* daemon. Each eligible program is a scheduling object and the remaining programs (the ineligible ones) constitute object $m+1$. While each eligible program would receive a time-slice of dedicated service and obtain its reuse distance from the *locality* daemon at each data server, we enhance the daemon to collect the reuse distance for scheduling object $m+1$ and pass it to the *orchestrator* daemon. Because the daemon knows the reuse distance of each of its scheduling objects, averaged over all data servers, it decides the scheduling time slice size for each object. With a fixed scheduling window, set to 500ms by default in our prototype, each object receives a portion of it as the time slice for its dedicated service, whose size is *inversely* proportional to the percentage of its average reuse distance over the sum of average reuse distances of all objects. The scheduling plan is then to schedule the programs in a window-by-window manner. In a window, each object receives its dedicated service slice. To schedule an object the *orchestrator* daemon broadcasts the object identifier to all *iScheduler* daemons. Each *iScheduler* daemon then releases the requests from program(s) matching the object identifier to the kernel until another object identifier is received. These requests are scheduled by the local disk scheduler for further optimization. As such, all of the ineligible programs have their requests scheduled together in the same time slice.

In the design of the scheduling, we take both efficiency and fairness into account. Smaller reuse distance indicates a higher request arrival rate, or higher I/O demand from one or multiple programs. Giving a larger service time slice to a program, or programs, of higher I/O demand is fair for all programs. At the same time, dedicated service to eligible programs allows their performance potential to be fully realized, rather than getting lost in the multiplexed use of data servers. A program with weak spatial locality, or large SL_{ij} values for program *j*, should get a small time slice in the interests of disk efficiency. However, we do not have to explicitly use this statistic in the allocation of time slices to induce this effect. This is because

large SL_{ij} values would usually imply a large reuse distance, which automatically leads to a small scheduling time slice.

IV. Performance Evaluation and Analysis

To evaluate the performance of IOrchestrator, we set up a cluster consisting of six compute nodes, six data servers, and one dedicated metadata server for the PVFS2 file system. All nodes were of identical configuration, each with dual 1.6GHz Pentium processors, 1GB memory, and a SATA disk (Seagate Barracuda 7200.10, 150GB) with NCQ enabled. Each node ran Linux 2.6.21 with CFQ (the default Linux disk scheduler), and used GNU libc 2.6. The PVFS2 parallel file system version 2.8.1 was installed. We used MPICH2-1.1.1, compiled with ROMIO, to generate executables for MPI programs. All nodes were interconnected with a switched Gigabit Ethernet network. A striping unit size of 64KB was used to stripe files over the six data servers in the PVFS2 file system. To ensure that all data were accessed from disk the system buffer caches of each compute node and data server were flushed prior to each test run.

A. The Benchmarks

We selected five MPI-IO applications of different and representative I/O access patterns to benchmark the PVFS2 parallel file system enhanced with IOrchestrator: *mpi-io-test*, *ior-mpi-io*, *noncontig*, *hpio*, and *mpi-tile-io*, which are briefly described following.

mpi-io-test is an MPI-IO benchmark from the PVFS2 software package [25]. In our experiments, we ran the benchmark with five MPI processes spawned, each on one compute node, to read or write one 10GB file. Each process accessed one segment of contiguous data at a time. If collective I/O is used, in each collective call five processes access five segments in a row, respectively. In the next call, the next five segments are accessed. The size of a request from each process was 64KB.

ior-mpi-io is a program in the ASCI Purple Benchmark Suite developed at Lawrence Livermore National Laboratory [12]. In this benchmark each of the five MPI processes is responsible for reading or writing its own $1/5$ of a file whose size is 10GB. Each process continuously issues sequential requests, each for a 64KB segment. If collective I/O is used, the processes' requests for the data are at the same relative offset in each process's access scope and are organized into one collective-I/O function call.

noncontig is a program from the Parallel I/O Benchmarking Consortium [23] developed at Argonne National Laboratory. This benchmark uses five MPI processes to read a 10GB file with a vector-derived MPI data type. If we consider the file to be a two-dimensional array, there are five columns in the array. Each process reads a column of the array, starting at row 0 of its designated column. In each row of a column there are *elmtcount* elements of the MPI_INT type, so the width of a column is $elmtcount \times sizeof(\text{MPI\_INT})$. If collective I/O is used, in each call the total amount of data read by the processes is fixed, determined by the buffer size, which is 8MB in our experiment.

hpio is a program designed by Northwestern University and Sandia National Laboratories to systematically evaluate I/O performance using a diverse set access patterns [7]. This benchmark program can generate differing data access patterns by changing three parameters: *region_count*, *region_spacing*, and *region_size*. In our experiment we set *region_count* to 4096, *region_spacing* to 10, and *region_size* to 64KB. Using five MPI processes, the access pattern is similar to the one described for benchmark *noncontig*. The length of a column is 4096 and the width of a column is 64KB. When collective I/O is used, each process accesses its designated column.

mpi-tile-io is also from the Parallel I/O Benchmarking Consortium [22]. It uses MPI processes to read or write a file in a tile-by-tile manner, with two adjacent tiles partially overlapped. Each process accesses 8KB, with 64B of overlapping between two consecutive accesses.

In all of these benchmarks file access can be set to either *read* or *write*. Additionally, both *hpio* and *noncontig* have the option of configuring their data access patterns as either contiguous or non-contiguous for memory access and file access. In summary, these selected benchmarks cover a large spectrum of access behaviors: from sequential access among processes (e.g., *mpi-io-test*) to non-sequential access among processes (e.g., *ior-mpi-io*), from read access to write access, from requests that are well-aligned with the 64KB striping unit size (e.g., *mpi-io-test* and *ior-mpi-io*) to requests of different sizes (e.g., *noncontig* and *mpi-tile-io*).

B. Performance of Homogenous Workloads

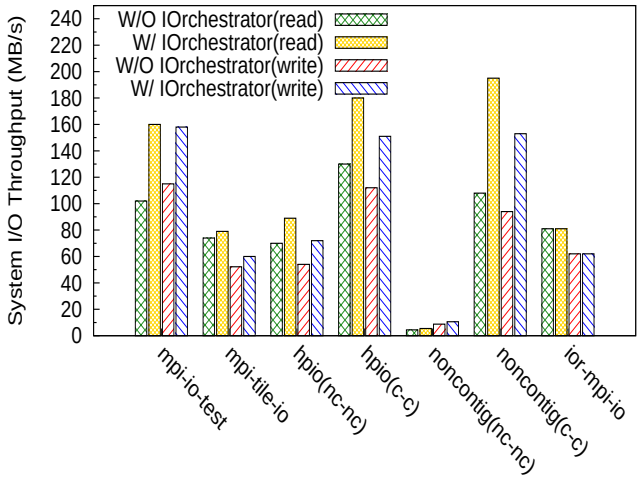

Fig. 2. Aggregate I/O throughput with running of two instances of the same program when only *barrier* is used. For each program, the data access is set as either *read* or *write*. For *hpio* and *noncontig* data access pattern is set as either contiguous (*c*) or non-contiguous (*nc*) for memory access and file access. The first symbol in the parentheses after a program name shows memory access configuration and the second symbol shows file access configuration. In our experiments only the configuration of file access (or the second *c*/*nc* symbol) directly affects I/O throughput.

In this experiment we run two instances of each benchmark on the PVFS2 parallel file system and measure the aggregate

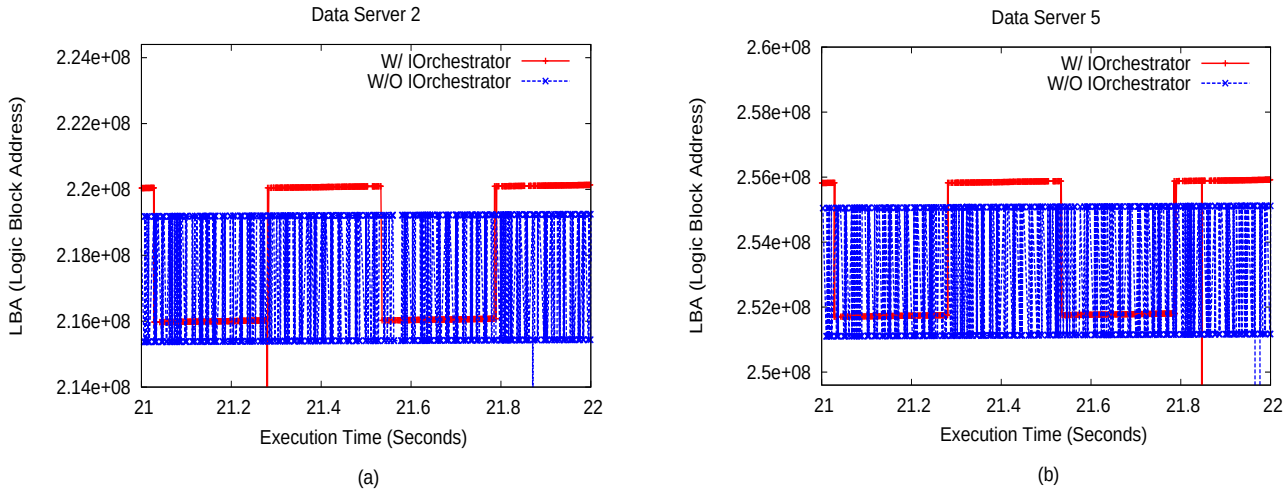

Fig. 3. Disk addresses (LBAs) of data access on the disks of data servers 2 and 5, shown in (a) and (b), respectively, in a sampled one-second execution period, when two *mpi-io-test* programs ran together with and without using IOrchestrator. Note that because the programs run much faster with IOrchestrator, they access disk positions somewhat different from those accessed by the programs without IOrchestrator during the same execution period.

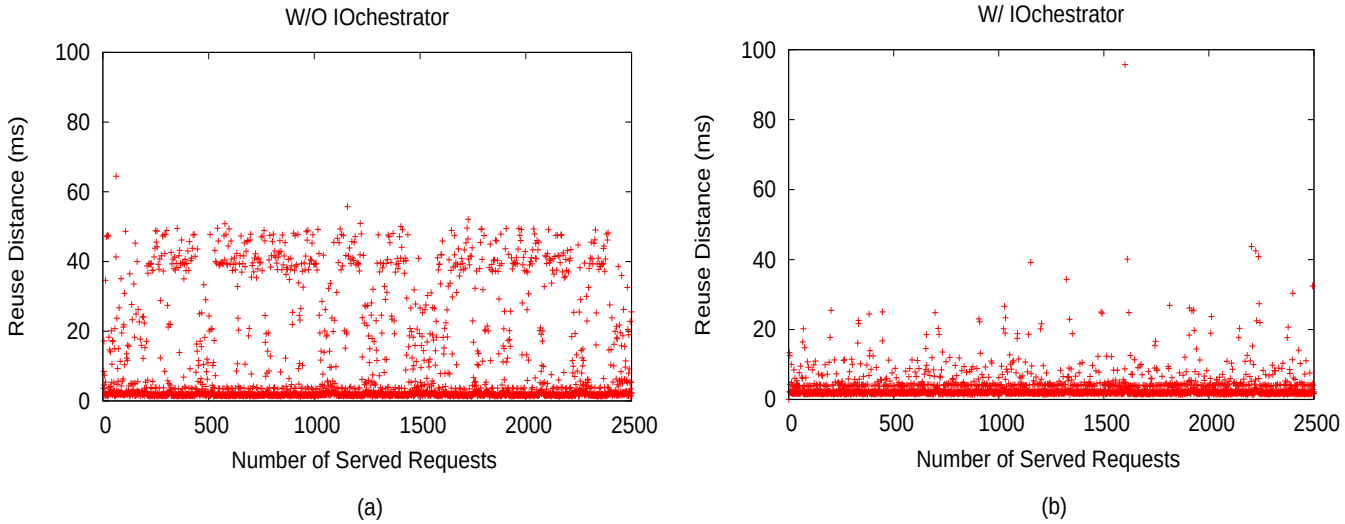

Fig. 4. Reuse distances of requests served at data server 5 measured for one program instance when two instances of the *mpi-io-test* benchmark ran together without and with using IOrchestrator.

I/O throughput with and without using IOrchestrator, respectively. Each running program accesses its own data file, which is striped over the six data servers.

Performance using only *barrier***.** Figure 2 presents the I/O throughput for the five benchmarks when only *barrier* is used between I/O operations and collective I/O optimization is not used. In the experiments their file access is configured either as *read* or as *write* and the access patterns of *hpio* and *noncontig* are configured either as *contiguous* or as *non-contiguous*. IOrchestrator improves the I/O throughput of the entire file system by up to 89% and 43% on average.

For the *mpi-io-test* benchmark, when IOrchestrator is used the I/O throughput is increased by 57% for *read* and 37% for *write*. The data access pattern of the program, or that of its process if only one process is created, is sequential. However, when two running programs, each with five processes, are sending their requests to the data servers, the disk head at each data server cannot turn this strong spatial locality (sequential access) into high disk efficiency. Figures 3(a) and 3(b) show the order of accessed disk addresses, or roughly the path of disk head movement at data servers 2 and 5, respectively, in a one-second execution sample. When IOrchestrator is not used, the disk head rapidly alternates between two disk regions, each storing a data file for one running program. The disk I/O scheduler, CFQ, does not preserve spatial locality within each program, though it conducts anticipatory scheduling similar to the AS disk scheduler. To discover why, we collected the reuse distances of one running program at data server 5 during

certain time period and show them in Figure 4(a) (without IOrchestrator) and Figure 4(b) (with IOrchestrator). Without IOrchestrator, there are many very large reuse distances (between 20ms and 50ms).[2] With such large reuse distances, it is impossible for the disk heads to be idly waiting for the next request from the same program without making the disks less productive. Thus, we see frequent disk head seeks between two distant disk regions in Figure 3. When detecting the strong locality within each running program, IOrchestrator provides dedicated service time slices to each. In its dedicated service period, all disks are coordinated to service one program and its reuse distance can be significantly reduced (Figure 4(b)). This helps exploit the strong locality inherent in the program into efficient disk access (see the lines showing access with IOrchestrator in Figure 3). We can make similar observations for other benchmarks with sequential access patterns, such as *hpio(c-c)* and *noncontig(c-c)*.

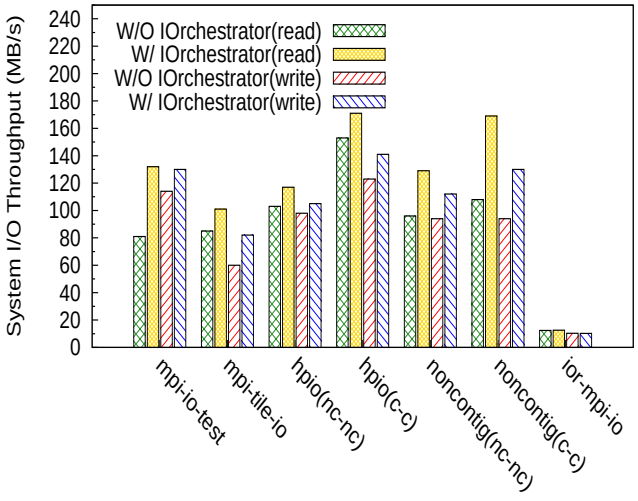

Fig. 5. Aggregate I/O throughput running two instances of the same program when *barrier* and *collective I/O* are used. For each program, the data access is set as either *read* or *write*. For *hpio* and *noncontig* the data access pattern is configured as either contiguous (*c*) or non-contiguous (*nc*)

For *mpi-tile-io* I/O throughput increased by 11% for *read* and 15% for *write*. The benchmark has a typical random data access pattern. The difference between spatial locality within each running program and that among running programs is relatively small, though it is larger than the threshold required for IOrchestrator to enable dedicated I/O service for them. For this reason, the performance improvement with IOrchestrator is small compared to the programs with strong intra-program locality. This explanation for smaller improvements also applies to benchmarks *noncontig(nc-nc)* and *hpio(nc-nc)*. The *ior-mpi-io* benchmark has very weak spatial locality. Requests from its processes access five different disk regions that are distant from each other (around 2GB). The cost of moving disk heads within one program is comparable to the cost of moving them between different running programs. Therefore, IOrchestrator disqualifies both running programs for dedicated services and essentially does not change the scheduling of the current PVFS2 file system. As we expected, the experimental results show little difference when using IOrchestrator. These results also indicate that the overhead introduced by IOrchestrator is trivial compared to I/O operations.

Performance with using both *barrier* and collective I/O. Figure 5 presents the I/O throughput for the five benchmarks when both *barrier* and collective I/O are applied. IOrchestrator improves the I/O throughput by up to 63%, and 28% on average.

For benchmarks with non-sequential access patterns such as *mpi-tile-io*, *hpio(nc-nc)*, and *noncontig(nc-nc)*, the use of collective I/O effectively improves the I/O performance because it transforms small random accesses to large sequential accesses within each program. However, the interference between running programs offsets the potential benefits of collective I/O. When requests involved in a collective I/O call do not have dedicated service by the data servers, the local disk I/O scheduler thrashes the disk head between programs to avoid long idle waiting periods. When IOrchestrator enables the dedicated service for eligible programs, the improved spatial locality can be exploited. For benchmarks that already have sequential access patterns, such as *mpi-io-test*, collective I/O may introduce overhead without improving locality and thus reduce I/O throughput. In such cases, the advantage of IOrchestrator is also apparent.

We also observe that the throughput of benchmark *ior-mpi-io* is significantly reduced when collective I/O is used. After analyzing the data accesses of the benchmark, we find that in one collective call only one or two data servers are busy serving requests while the others are idle because of a mismatch between the data request pattern and the striping pattern, severely under-utilizing the system. IOrchestrator does not apply dedicated I/O service to the program because of weak spatial locality within the program, and because the difference between intra-program and inter-program localities is not consistent across the data servers.

Performance without *barrier* or collective I/O. Figure 6 presents the I/O throughput of the five benchmarks in which the *barrier* routines between parallel I/O routines are removed and collective I/O is not used. Without *barrier* and collective I/O, the throughput of the benchmarks is reduced except for *hpio(c-c)* and *noncontig(c-c)*. For example, the throughput of *mpi-io-test* is reduced from 102 MB/s to 86 MB/s for *read*, and from 115 MB/s to 108 MB/s for *write*, and the throughput of *hpio(nc-nc)* is reduced from 70 MB/s to 31 MB/s for *read*, and from 54 MB/s to 28 MB/s for *write* (Figures 2 and 6). Without *barrier* and collective I/O, each process proceeds at its own pace, making the on-disk distances of data accessed by different processes of the program increasingly larger. For *hpio(c-c)* and *noncontig(c-c)*, the size of requests is very large (more than 10MB), which by itself can make the disk efficient. The overhead paid by *barrier* and collective I/O does not pay

[2]Those very small reuse distances shown in Figure 4 are mostly produced by requests from different processes of the running program, and can be exploited by the CFQ scheduler.

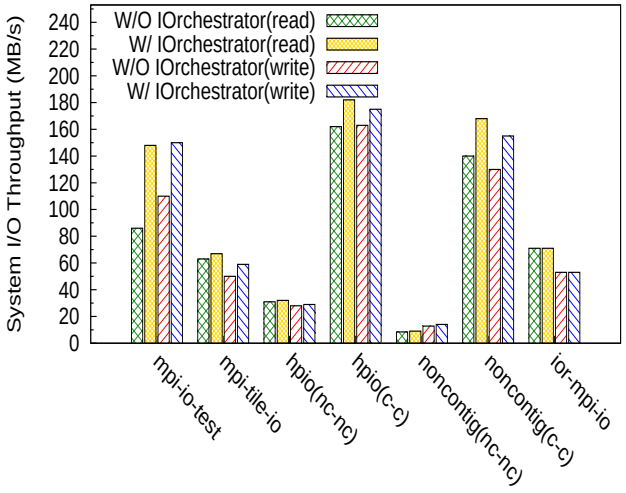

Fig. 6. Aggregate I/O throughput with running of two instances of the same program when both *barrier* and collective I/O are not used. For each program, the data access is set as either *read* or *write*. For *hpio* and *noncontig*, data access pattern is configured as either contiguous (*c*) or non-contiguous (*nc*)

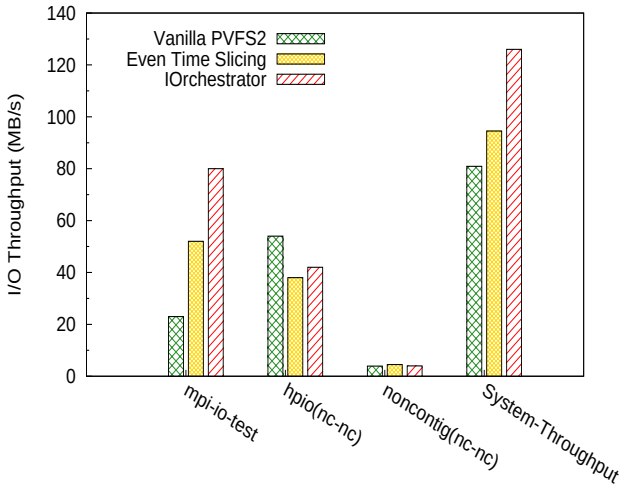

Fig. 7. I/O throughputs of three different programs, *mpi-io-test*, *noncontig(nc-nc)*, and *hpio(nc-nc)*, when they are running together to read three data files of 10GB, respectively. The entire system's throughputs are also shown. Three systems are tested in the experiment: the vanilla PVFS2, PVFS2 with IOrchestrator with even time slicing, and PVFS2 with IOrchestrator.

off, and the throughputs are even higher when these techniques are not used.

When the spatial locality within a program is weakened by not using *barrier* and collective I/O, the relative performance advantage of IOrchestrator is usually reduced, as shown in Figure 6. The exception is *mpi-io-test*, in which 72% and 36% throughput increases for *read* and *write*, respectively, are achieved without *barrier* and collective I/O, compared with 57% and 37% with only *barrier*, and 63% and 15% using both *barrier* and collective I/O, respectively. This is because dedicated I/O service enforced by IOrchestrator allows processes of a program of sequential access pattern to receive equal service in a time slice and forces them to progress at almost the same speed.

C. Performance of Heterogenous Workloads

Next we study the effectiveness of IOrchestrator when different programs are running concurrently. We select three programs of different access patterns, *mpi-io-test*, *noncontig(nc-nc)*, and *hpio(nc-nc)*, and run one instance of each concurrently to read three 10GB files, respectively. In addition to running the programs with the vanilla PVFS2 and with IOrchestrator, we test a version of IOrchestrator restricted to *even time-slicing*, wherein the time slice is evenly allocated to each scheduling object instead of proportionally allocated according to reuse distance. Figure 7 shows both the throughput for each running program and the throughput of the entire system. With just vanilla PVFS2 the throughout of *hpio(nc-nc)* (with weak locality) has greater throughput than *mpi-io-test* (with strong locality). When more disk time is allocated to serve random, rather than sequential, requests, the disk's efficiency is reduced. Thus the entire system's throughput is the lowest among the three tested scenarios. By dedicating one third of system service time to each program, the program with stronger locality will produce higher throughput without interference from programs of weaker locality. Even time-slicing improves the system throughput by 17%. With IOrchestrator, *mpi-io-test* is identified as being eligible for dedicated service while the other two programs are not. According to their reuse distances, *mpi-io-test* is allocated about half of the disk service time, while *noncontig(nc-nc)* and *hpio(nc-nc)* together receive the other half. Both *mpi-io-test* and *hpio(nc-nc)* enjoy increased throughput while *noncontig(nc-nc)* is little affected. IOrchestrator further improves aggregate system throughput by 30%. Though further improving throughput of *mpi-io-test* as well as system throughput is possible by allocating more disk service time to *mpi-io-test*, it would unduly compromise fairness among the co-running programs. IOrchestrator, by its design, has addressed this issue.

D. Effect of File Distances among Programs on IOrchestrator

The distance between files accessed by different programs has a direct effect on the spatial locality among programs. The larger the distance, the weaker the locality, and consequently the greater potential for IOrchestrator to improve performance. To confirm this speculation, we run two instances of *mpi-io-test* reading two files of 1GB, respectively, at different distances apart. The on-disk distance is the size of the space (difference in LBA times block size) separating the files. In our experiment we use distances of 0GB, 10GB, 20GB, and 30GB.[3] Figure 8 shows the system's I/O throughputs. The results are consistent with our hypothesis: when the distance

[3]In the previous experiments 0GB between files was used. Thus the performance measurements reported in those experiments represent lower bounds (on our testbed) on possible performance improvements made by IOrchestrator.

is increased from 0GB to 30GB, I/O throughput is reduced by 48% without using IOrchestrator. This reduction is especially significant when the distance is still relatively small, such as from 0GB to 10GB, and from 10GB to 20GB. When IOrchestrator is used both running programs are identified as eligible for dedicated service. With a 30GB distance IOrchestrator improves the system throughout by 2.5 times. As the file distance increases, we only observe minor reductions of throughput (5% from 0MB to 30GB). When dedicated service time slices are alternated between these two running programs, the frequency of disk head seeks between programs becomes much lower, and the cost for the seeks becomes less significant to the I/O efficiency.

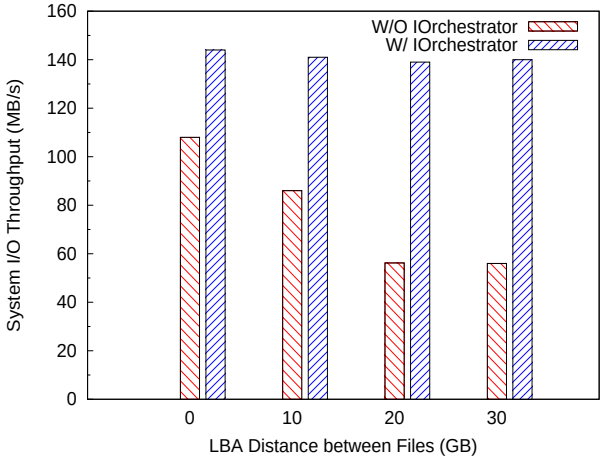

Fig. 8. Aggregate I/O throughputs measured when we increase on-disk file distances from 0GB to 30GB.

E. Impact of Scheduling Window Size

Each scheduling object receives a portion of each scheduling window as its time slice for dedicated service. In the experiments we have so far described the 500ms default scheduling window size was used. Next we study the impact of scheduling window size on the effectiveness of IOrchestrator. To this end we run two instances of the *mpi-io-test* program concurrently, each reading one 10GB files, with window varying among 125ms, 250ms, 500ms, and 1000ms. Figure 9 shows the system I/O throughputs with different window sizes. Compared to the vanilla system, the I/O throughput is increased by 40.2%, 48.5%, 58.6%, and 59.6% with the selected window sizes, respectively. Apparently a larger window allows a scheduling object to stay with its dedicated I/O service for a longer time period and reduces the frequency of disk head switches among scheduling objects, consequently improving I/O performance. This is consistent with our observation on the experiment results. The improvement is substantial when the window size is relatively small. However, when the window is sufficiently large, such as 500ms, further increasing the window size, such as 1000ms, receives diminishing return on I/O performance. In the meantime, a too-large window can allow one scheduling object to exclusively hold disk service for a very long time period at a time and make programs less responsive. For this reason we select a modest time period as the default window size for IOrchestrator.

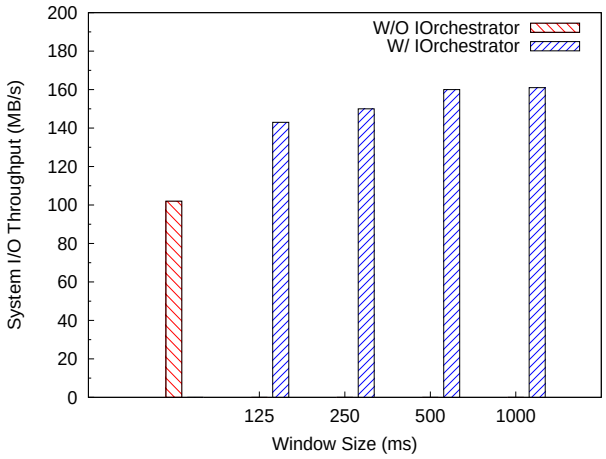

Fig. 9. Aggregate I/O throughputs measured when we increase the scheduling window size from 125ms to 1000ms. The I/O throughput without using IOrchestrator is also shown for comparison.

V. Conclusions

In the paper we have described the design and implementation of IOrchestrator, a technique for identifying and exploiting spatial locality that is inherent in individual parallel programs but gets lost with the use of a shared multi-node I/O system. With careful, dynamic analysis of cost-effectiveness, IOrchestrator gives programs with strong locality dedicated I/O service time by coordinating data servers. IOrchestrator is implemented in the PVFS2 parallel file system with modest instrumentation in the Linux kernel and the ROMIO MPI library. Our experimental evaluation of the scheme with representative I/O-intensive parallel benchmarks, such as *mpi-io-test* and *mpi-tile-io*, shows that it can improve system I/O performance by up to 2.5 times, and 39% on average, without compromising fairness of I/O service. Furthermore, the implementation of IOrchestrator does not rely on specific functionalities or features of PVFS2 and MPICH2. We expect the principle and design of IOrchestrator can be effectively applied to high-performance computing platforms with other parallel file systems or parallel libraries.

Acknowledgment

We thank the anonymous reviewers for their constructive comments. This work was supported by US National Science Foundation under grants CCF 0702500, CRI-0708232, and CAREER CCF 0845711. This work was also funded in part by the Accelerated Strategic Computing program of the Department of Energy. Los Alamos National Laboratory is operated by Los Alamos National Security LLC for the US Department of Energy under contract DE-AC52-06NA25396.

REFERENCES

[1] J. Axboe, "Blktrace", http://linux.die.net/man/8/blktrace, Online-document, 2010.

[2] J. Axboe, "Completely Fair Queueing (CFQ) scheduler", http://en.wikipedia.org/wiki/CFQ, Online-document, 2010.

[3] 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 Conferenece on File and Storage Technologies*, San Fancisco, CA, 2009.

[4] P. Carns, W. Ligon III, R. Ross, and R. Thakur, "PVFS: A Parallel File System For Linux Clusters", *In Proceedings of the 4th Annual Linux Showcase and Conference*, Atlanta, GA, 2000.

[5] 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.

[6] 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.

[7] A. Ching, A. Choudhary, W. Liao, L. Ward, and N. Pundit, "Evaluating I/O Characteristics and Methods for Storing Sturctured Scientific Data", *In Proceedings of IEEE Internatinal Parallel & Distributed Processing Symposium*, Honolulu, Hawaii, 1996.

[8] D. Feitelson and L. Rudolph, "Gang scheduling performance benefits for fine-grain synchronization", *Journal of Parallel and Distributed Computing*, 16:306-18, 1992.

[9] W. Hsu, A. Smith, and H. Young, "The Automatic Improvement of Locality in Storage Systems", *ACM Transactions on Computer Systems*, Volume 23, Issue 4, Nov. 2005, pages 424-473.

[10] 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.

[11] 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.

[12] Interleaved or Random (IOR) benchmarks, *URL: http://www.cs.dartmouth.edu/ pario/examples.html*.

[13] J. Kim, G. Choi, D. Ersoz, and C. Das,"Improving Response Time in Cluster-Based Web Servers through Coscheduling", *In Proceedings of IEEE Internatinal Parallel & Distributed Processing Symposium*, Santa Fe, NM, 2004.

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

[15] R. Koller and R. Rangaswami,"I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance", *In Proceedings of the 8th USENIX Conferenece on File and Storage Technologies*, San Jose, CA, 2010.

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

[17] M. Kim, "Synchronized disk interleaving", *IEEE Transactions on Computers*, C-35(11):978-988, November 1986.

[18] Lustre File System. http://www.lustre.org/. Online-document, 2010.

[19] Linux Code on Anticipatory Scheduling, "http://lxr.linux.no//#linux+v2.6.28.5//block//cfqiosched.c"

[20] R. Latham, N. Miller, R. Ross, and P. Carns. "A next-generation parallel file system for linux clusters", *Linux World Magazin*, January, 2004.

[21] A. Morton, "Linux: Anticipatory I/O Scheduler", *http://kerneltrap.org/node/567*

[22] Mpi-tile-io Benchmark, *URL: http: www-unix.mcs.anl.gov/thakur/pio-benchmarks.html*.

[23] Noncontig I/O Benchmark, *URL: http://www-unix.mcs.anl.gov/thakur/pio-benchmarks.html*.

[24] J. Ousterhout, "Scheduling techniques for concurrent systems", *In Proceedings of International Conference on Distributed Computing Systems*, Miami/Ft. Lauderdale, FL, 1982.

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

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

[27] 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.

[28] 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.

[29] R. Thakur, W. Gropp and E. Lusk, "On Implementing MPI-IO Portably and with High Performance", *In Proceedings of Sixth Workshop on I/O in Parallel and Distributed Systems*, Atlanta, GA, 1999.

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

[31] M. Wachs and G. Ganger,"Co-scheduling of disk head time in cluster-based storage", *In Proceedings of 28th International Symposium on Reliable Distributed Systems*, Nigara Falls, NY, 2009.

[32] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. Ganger,"Argon: performance insulation for shared storage servers", *In Proceedings of the 5th USENIX Conferenece on File and Storage Technologies*, San Jose, CA, 2007.

[33] X. Zhang and S. Jiang,"InterferenceRemoval: Removing Interference of Disk Access for MPI Programs through Data Replication", *In Proceedings of International Conference on Supercomputing*, Tsukuba, Japan, 2010.

[34] X. Zhang, S. Jiang, and K. Davis, "Making Resonance a Common Case: A High-Performance Implementation of Collective I/O on Parallel File System", *In Proceedings of IEEE Internatinal Parallel & Distributed Processing Symposium*, Rome, Italy, 2009.