1-1-2012

Rethinking the design and implementation of the i/o software stack for high-performance computing

Xuechen Zhang
Wayne State University,

RETHINKING THE DESIGN AND IMPLEMENTATION OF THE I/O SOFTWARE STACK FOR HIGH-PERFORMANCE COMPUTING

by

XUECHEN ZHANG

DISSERTATION

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

2012

MAJOR: COMPUTER ENGINEERING

Approved by:

Advisor	**Date**

ACKNOWLEDGEMENTS

Working towards a dissertation for me is like painting on a blank paper. Back to the year 2007 when I first joined Wayne State University, I did not know what the picture might look like until its every piece is finally completed now. This dissertation would not have been possible without the guidance, friendship, and love of many people in my life.

Foremost, I would like to express my sincere gratitude to my advisor Dr. Song Jiang, who has guided me through all the hard times of my research for this thesis with his fine leadership and immense knowledge. He will always be a role model in my life to pursue the state-of-the-art research and achieve my career potential.

Special thanks to Dr. Kei Davis for providing me the internship at Los Alamos National Laboratory, for his insightful comments on most of my research papers, and for his hard work before deadlines, which made the papers in a much better shape.

I would also like to thank the rest of my thesis committee, Prof. Cheng-zhong Xu, Prof. Nabil J. Sarhan, and Prof. Weisong Shi, for their encouragement, support, and hard questions to improve this work.

To my dear friends in our group, thank you for making our weekly event such a good time to exchange thoughts and ideas, find pros and cons of this work, and improve my presentation and communication skills. Yuehai Xu, Ke Liu, Yizhe Wang, Dr. Jia Rao, Dr. Jiayu Gong, Xiangping Bu, Kun Wang, Yudi Wei, and Pei Yan, thanks for being inspiring and your friendship.

To the staff and students at CCS-7 at LANL and Distributed Resource Management Team at VMware Inc., I am grateful for the chance to visit and be part of the groups. Thanks for welcoming me and helping me out of troubles in my work and life.

Lastly, I wish to thank my entire extended family, my grandparents, my aunts and uncles, my sisters and brothers. But most importantly, my parents Zhengjiang Zhang and Su'e Sun, for their love and support throughout my life. To them I dedicate this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

One of the challenges facing **H**igh-**P**erformance **C**omputing(HPC) in peta-scale computing is the rapidly increasing of I/O demands of both scientific and industrial applications, such as climate change forcasting [60], nuclear security maintaining [31], and financial data modeling [36]. More specifically, storage system for peta-scale computing must have the ability to process tera-bytes, if not peta-bytes, of data which are generated in bursty [24, 28, 31] and handle very high I/O concurrency from parallel processes running on millions of cores [29, 32, 27]. Because the performance of hard-disk-based storage systems is state-dependent for its mechanical moving parts inside [79], many research and optimization efforts have been made at different layers of I/O stack, such as runtime library and OS kernel in order to explore higher **parallelism** and better **locality** of data access for I/O performance improvement. However, isolated optimizations at individual layers of the I/O stack are often unable to achieve the full potential of storage devices since it is the aggregated impact on the whole layers that determines the I/O performance of the systems. Therefore, a comprehensive re-examination of the design and implementation of existing software stack, especially for parallel I/O, is necessary for solving performance bottleneck in

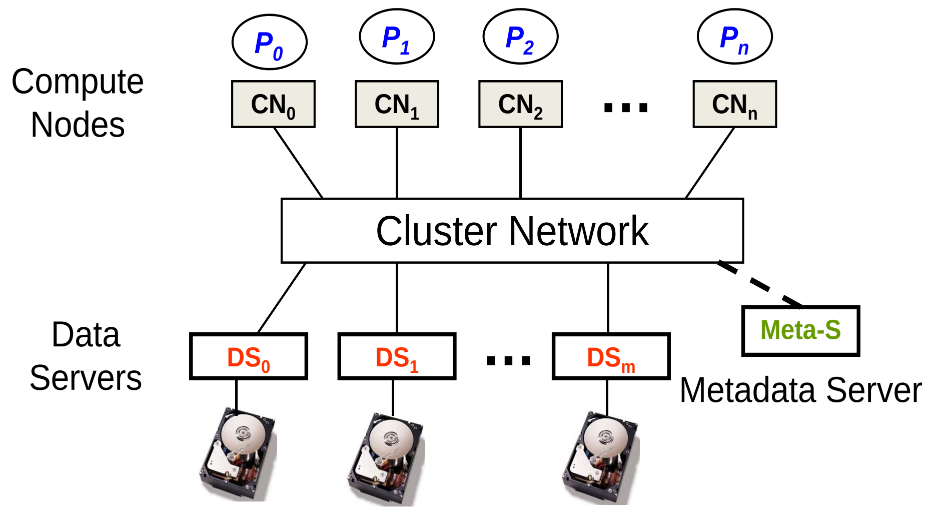


Figure 1.1: System architecture of a representative cluster for high-performance computing using a parallel file system for management of concurrent data access on shared files.

HPC systems. Before describing the issues which can compromise I/O efficiency I will first review the current hardware and software architecture for high-performance I/O.

1.1 Hardware and Software Architecture for High-Performance I/O

The architecture of a high-performance computing machine has evolved over the decades. Since our research focuses on performance impact of increased parallelism and dynamic data access patterns on the storage system using hard disks, in this dissertation we target on a computing environment, where requests are directly served on the storage system without using staging areas [88]. Figure 1.1 shows such a cluster for high-performance computing using a parallel file system [17, 71, 92, 2, 10]. In the system which is widely used in in-house cluster computing environment, some nodes are configured as compute nodes, where processes such as those of an MPI program are running in parallel, and some are data servers, where user data files are striped. There is a meta-data server, which provides meta-data service such as the locations of the requested data in response to the inquiries from compute nodes. Usually, there is a file system daemon running on every data server, which receives I/O requests from clients and issues to operating systems of servers. There is no communication or coordination between data servers. There are three parameters: striping unit size, striping factor, and striping index that describe the on-disk layout of file data. In order to achieve the maximal parallelism of data access, file is often striped over the whole data servers regardless of programs' access patterns.

In addition to using dedicated data servers to handling I/O requests, the software system for high-performance I/O is also highly optimized over all the layers, which form a deep I/O stack. Many of the optimizations concern **Temporal Locality** and **Spatial Locality** of data access. Applications with temporal locality tend to repeatedly access a small set of data. Spatial locality describes the sequentiality of continuously requested data on storage devices. Usually this concept applies to the hard-disk-based storage. Hard disks are such devices that accesses of sequential data is at least a magnitude faster than accesses of random data, because significant seek time is usually involved in accessing non-sequential data. The optimization techniques can be implemented on either compute node side or data server side. Figure 1.2 shows possible techniques which can be applied to an individual layer of an I/O stack.

Application layer contains the software that a user interacts with. Some applications may have special knowledge of its data access pattern, which can be leveraged by operating system making smarter decision [40]. Compiling technique [67] can also help in this layer to maximize reuseness on a small data set.

MPI-IO middleware layer, which brings transparency and adaptability of software

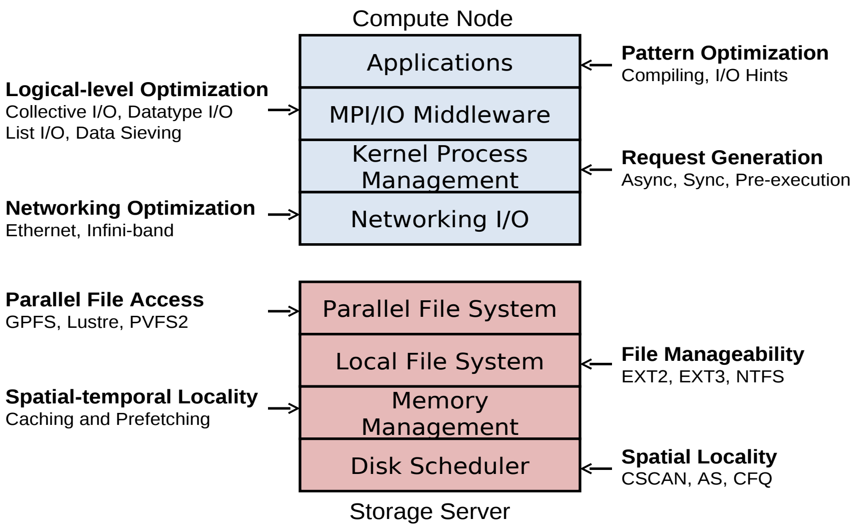


Figure 1.2: System components which form a deep parallel I/O stack for highly efficient I/O services in support of high-performance computing.

to users, is widely used for high-performance computing to improve I/O performance. Many optimization techniques have been proposed to transform many small and non-contiguous I/O requests to a few large and contiguous requests, using techniques such as collective I/O [100], data sieving [100], datatype I/O [48], and list I/O [47] in this layer.

Kernel process management layer Since I/O latency on a regular storage device using hard disks is easily longer than hundreds of CPU cycles, I/O requests of parallel processes are usually completed asynchronously using kernel interrupts [107]. After issuing requests, processes move to waiting mode. Using asynchronous/nonblocking I/O [23], with which processes continue its computing assignment, can significantly improve I/O efficiency by generating a batch of requests as well as the usage of computing resource. When a system is I/O bottlenecked, pre-execution threads [44, 56, 117] can also help to generate I/O hints for underlying layers of I/O stacks.

Networking I/O layer Networking design is crucial to both performance and scalability of a high-performance computing machine. 10-gigabit Ethernet [22] and Infiniband [26] are two of the most popular networking protocals in HPC platform. Performance of collective communications also hinges on networking topology and CPU architecture [77].

Parallel file system layer By leveraging parallel file systems, such as PVFS [17, 71], GPFS [92], and Lustre [2, 10], which are developed to exploit the natural parallelism in a shared cluster having multiple data servers by striping file data over them, if I/O request size is much larger than stripe size and the number of concurrent requests is much more than number of data servers, all the servers will service requests concurrently to provide superior aggregate I/O throughput in the order of GB/s for most systems in production.

Local file system layer Local file systems, such as EXT3 [25] and NTFS [30], provides both data manageability and availability to users. Designed for handling write-intensive workloads, log-structured file systems [91] place updates sequentially on storage medium for better spatial locality.

Memory management layer Memory management is one of the most important component of operating system. Concerning program's locality, a large body of data caching [78, 35, 66, 65, 81] and prefetching [54, 85] algorithms have been proposed according to different criterions such as recency, frequency, reuse distance, and so on. In addition, buffer cache in main memory helps improve write-back efficiency through providing more optimization space.

Disk scheduler layer Before being dispatched to device drivers, the service order of block-level I/O requests are adjusted based on sequentiality in this layer. Disk I/O schedulers such as anticipatory scheduler [64, 1] and completely fairness queuing scheduler [7] are widely adopted in Linux operating system. I/O requests in the scheduling queue are sorted and merged according their logical block addresses to explore spatial locality on hard disks.

1.2 Thesis Contributions

The main contributions of this thesis are to 1) investigate and identify the existence of four performance bottlenecks in the layered I/O stack through benchmarking the HPC system with parallel I/O benchmarks with a wide coverage of workload characteristics; 2) find the root causes through re-examining the limitations in their design and implementation of process management, collective I/O, parallel file systems, and disk scheduler; 3) present and implement efficient solutions to each of them by taking advantages of software techniques such as pre-execution or leveraging novel storage device such as solid-state disks.

1.2.1 *DualPar:* Enabling Data-driven Execution for I/O Efficiency

A parallel system relies on both process scheduling and I/O scheduling for efficient use of resources, and a program's performance hinges on the resource on which it is bottlenecked. Existing process schedulers and I/O schedulers are independent. However, when the bottleneck is I/O, there is an opportunity to alleviate it via cooperation between the I/O and process schedulers: the service efficiency of I/O requests can be highly dependent on their issuance order, which in turn is heavily influenced by process scheduling.

We propose a data-driven program execution mode in which process scheduling and request issuance are coordinated to facilitate effective I/O scheduling for high disk efficiency. Our implementation, DualPar, uses process suspension and resumption, as well as pre-execution and prefetching techniques, to provide a pool of pre-sorted requests to the I/O scheduler. This *data-driven* execution mode is enabled when I/O is detected to be the bottleneck, otherwise the program runs in the normal computation-driven mode. DualPar is implemented in the MPICH2 MPI-IO library for MPI programs to coordinate I/O service and process execution. Our experiments on a 120-node cluster using the PVFS2 file system show that DualPar can increase system I/O throughput by 31% on average, compared to existing MPI-IO with or without using collective I/O.

1.2.2 *Resonant I/O:* high-performance I/O with Data Layout Awareness

Collective I/O is a widely used technique to improve I/O performance in parallel computing. It can be implemented as a client-based or server-based scheme. The client-based implementation is more widely adopted in MPI-IO software such as ROMIO because of its independence from the storage system configuration and its greater portability. However, existing implementations of client-side collective I/O do not take into account the actual pattern of file striping over multiple data servers in storage systems. This can cause a significant number of requests for non-sequential data at data servers, substantially degrading I/O performance.

Investigating the surprisingly high I/O throughput achieved when there is an accidental match between a particular request pattern and the data striping pattern on the data servers, we reveal the resonance phenomenon as the cause. Exploiting readily available information on data striping from the metadata server in popular file systems such as PVFS2 and Lustre, we design a new collective I/O implementation technique, resonant I/O, that makes resonance a common case. Resonant I/O rearranges requests from multiple MPI processes according to the presumed data layout on the disks of data servers so that non-sequential access of disk data can be turned into sequential access, significantly improving I/O performance without compromising the independence of a client-based implementation. We have implemented our design in ROMIO. Our experimental results on a small- and medium-scale cluster show that the scheme can increase I/O throughput for some commonly used parallel I/O benchmarks such as mpi-io-test and ior-mpi-io over the existing implementation of ROMIO by up to 157%, with no scenario demonstrating significantly decreased performance.

1.2.3 *IOrchestrator:* Preserving Spatial Locality using Inter-Server Coordination

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 order to preserve the spatial locality of programs, 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 significant performance degradation.

1.2.4 *iTransformer:* Using SSD to Improve Disk Scheduling

The parallel data accesses inherent to large-scale data-intensive scientific computing require that data servers handle very high I/O concurrency. Concurrent requests from different processes or programs to hard disk can cause disk head thrashing between different disk regions, resulting in unacceptably low I/O performance. Current storage systems either rely on the disk scheduler at each data server, or use SSD as storage, to minimize this negative performance effect. However, the ability of the scheduler to alleviate this problem by scheduling requests in memory is limited by concerns such as long disk access times, and potential loss of dirty data with system failure. Meanwhile, SSD is too expensive to be widely used as the major storage device in the HPC environment.

We propose iTransformer, a scheme that employs a small SSD to schedule requests for the data on disk. Being less space-constrained than with more expensive DRAM, iTransformer can buffer larger amounts of dirty data before writing it back to the disk, or prefetch a larger volume of data in a batch into the SSD. In both cases high disk efficiency can be maintained even for concurrent requests. Furthermore, the scheme allows the scheduling of requests in the background to hide the cost of random disk access behind serving process requests. Finally, as a non-volatile memory, concerns about the quantity of dirty data are obviated. We have implemented iTransformer in the Linux kernel and tested it on a large cluster running PVFS2. Our experiments show that iTransformer can improve the I/O throughput of the cluster by 35% on average for MPI-IO benchmarks of various data access patterns.

1.3 Thesis Organization

The rest of the dissertation is organized as follows:

Chapter 2 provides an overview of existing work on execution modes, optimization techniques in MPI-IO middleware, scheduling of resource entities, and application of SSDs in computer systems.

Chapter 3 describes why data-driven execution mode is needed when performance of HPC system hinges on I/O resources. Then we present the design and implementation of DualPar, which enables the system to on-line adapt to computing-driven execution or data-driven execution according to its resource constraints. We then compare our solution with existing techniques using real implementation in MPI-IO middleware.

In Chapter 4, we first investigate the reason of I/O resonance phenomenon which results in surprisingly high I/O throughput. Then according to our findings, we design and implement resonant I/O, replacing the current implementation of collective I/O in MPI/IO library, to achieve a better match between request patterns and data striping pattern on data servers. Finally, we compare resonant I/O to collective I/O with representative MPI-IO benchmarks.

Chapter 5 presents the reason that spatial locality can be compromised when multiple data servers concurrently service I/O requests of different applications. Then we propose IOrchestrator scheduling framework, which uses inter-server coordination to preserve the spatial locality and improve disk efficiency.

In Chapter 6, we first use a motivation example to show why the current disk scheduling framework cannot handle high I/O concurrency for peta-scale computing. Then we describes the design and implementation of iTransformer using SSDs as queue extension of disk scheduler. In the end, we benchmark the system using iTransformer with both benchmarks and real-world workloads.

Finally, Chapter 7 summarizes our contributions and the limitations of the work, and we propose open questions and directions for future research.

Chapter 2

Related Work

This chapter reviews research literature on improving I/O performance of high-performance computing systems using parallel file systems in four categories: (1) data-driven execution mode and the pre-execution technique for improving I/O performance; (2) optimization techniques in MPI-IO middleware; (3) recovering lost spatial locality when running multiple processes/programs using disk scheduler; (4) application of SSD in the memory hierarchy; (5) quality of service(QoS) for end users of I/O-intensive HPC applications. We compare our work DualPar [117], resonant I/O [119], IOrchestrator [114], and iTransformer [116] to the related work in contexts, respectively.

2.1 Data-driven Process Scheduling and Pre-execution

In the parallel computing paradigm, the dataflow architecture has been investigated to expose the greatest concurrency. If the operations are encapsulated as processes, in the dataflow architecture process scheduling is explicitly affected by data availability. The concept is closely related to large-scale data-parallel computation infrastructures, including Google's MapReduce [53] and Microsoft's DryadLINQ [110]. In contrast, DualPar introduces a data-driven execution mode into the general-purpose computing environment to overcome the I/O bottleneck. To reduce I/O latency, Steere proposed a new operating system abstraction—Dynamic Sets—to allow the I/O system to access a group of data items in an order deemed most efficient to the system [98]. Accordingly the order of processing the data items follows the order in which they become available. To adopt the data-driven computation model, programmers need to disclose the set of data that can be processed concurrently. In another work, a set of language constructs for asynchronous IO are introduced into native languages such as C/C++ [59]. In this work long-latency I/O operations can be overlapped with computations with retaining a sequential style of programming. The potential performance advantage of these works is limited by the data concurrency that can be disclosed by programmers. In contrast, DualPar uses pre-execution to obtain the set of data to be requested by multiple processes without requiring any changes on the application source code.

Chang et al. proposed to exploit speculative execution (or pre-execution) to initiate I/O prefetching to reduce I/O stalls [44, 56]. Whenever the normal execution of a process is blocked by I/O requests, speculative execution takes the opportunity to run ahead and issue non-blocking prefetch requests. In these works the researchers made every effort to ensure that speculative execution uses only spare CPU cycles and normal processing always takes higher scheduling priority than the pre-execution process. Recently in the parallel computing arena a speculative prefetching approach was proposed to use program slicing to extract I/O-related code from the program, which is executed by a prefetch process [108]. All these works aim to hide I/O latency behind computation rather than improve I/O efficiency in serving prefetch requests, so prefetch requests are issued for service as soon as they are generated. For I/O-intensive programs, when there is a disparity between application access patterns and the physical data layout on disk, sequences of I/O requests that do not respect the physical data layout may induce a large I/O latency that is almost impossible to hide behind computation. In contrast, by being aware of the I/O bottleneck DualPar uses pre-execution for improving I/O efficiency instead of attempting to hide I/O time behind minimal computation time.

2.2 Optimization Techniques in MPI-IO Middleware

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 requests into a smaller number of larger contiguous requests. Data sieving [100] is one such technique wherein instead of accessing each portion of the data separately, a larger contiguous chunk that spans multiple requests is read/written. If the overhead for accessing additional unneeded data, called holes, is not excessive, its benefit can be significant. However, data sieving cannot ensure that its aggregated large requests from multiple clients are serviced at each data server in an order that minimized disk seeks, which is the objective of resonant I/O.

Datatype I/O [48] and list I/O [47] 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, ROMIO collective I/O [100] 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. However, collective I/O may adversely cause requests to arrive at each data server in an order inconsistent with data placement. In other work [63], an optimization is made to improve the ROMIO collective-I/O efficiency in a cluster where data is striped on the disks local to each compute node. The efficiency is achieved by making each agent process access only data on its local disks. In contrast, resonant I/O addresses I/O efficiency in a cluster with dedicated data servers that may service requests from multiple compute nodes.

Because the configuration of the storage subsystem of a cluster may be modified independently of the computing subsystem, it is desirable to implement I/O optimization techniques on the client side to keep them independent of configuration of storage subsystem. Collective I/O, as well as other commonly used techniques, are usually implemented on the client side. In contrast, server-side implementations such as server-directed collective I/O [93] are less adopted. Server-directed collective I/O was developed as a component of Panda, an I/O library for accessing multi-dimensional arrays, on the IBM SP2 supercomputer [93]. In this system I/O nodes are heavily involved in the re-arrangement of I/O requests by collecting request information from compute nodes and then directing them for sending/receiving data. Disk-directed I/O [69] is a strategy similar to server-directed collective I/O, with the addition of explicit scheduling of requests according to the data layout on disk. While these two techniques can provide performance benefits similar to resonant I/O, both of them compromise the independence of middleware on compute-side I/O, such as MPI-IO, from configuration changes on the data server side.

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.

2.3 I/O Request Scheduling

The weakened or even lost spatial locality with concurrent servicing requests can be recovered by disk schedulers [8, 82, 7]. However, in a multi-disk storage system, a higher-level coordination of I/O requests issued from different programs is needed in order to alleviate the disk head thrashing, leading to improvement in I/O performance.

Early work [9] on I/O request scheduling does no optimization. The scheduler is best used with devices that do not depend on mechanical movement like solid state devices, but not hard disks. Work-conserving disk schedulers, like the deadline scheduler [8] sorts incoming I/O requests in queues according to their logical block addresses(LBAs) and implements request merging for minimized hard disk seeks. Recently, non-work-conserving disk schedulers, such as the AS [64] and CFQ scheduler [7, 102], were designed to save the spatial locality with concurrent servicing of interleaved requests issued by multiple processes. 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 [82]. 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.

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 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 [84, 55]. A similar idea was used for disk spindle synchronization in a disk array to reduce platter rotation time in serving small requests [68]. 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 [51]. Wachs et al. proposed timeslice co-scheduling for cluster-based storage [103]. The objective of this latter work is better performance insulation quantified by *R-value* [102] 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.

2.4 SSD-based Memory and Storage Systems

Because hard disk performance is severely degraded by non-sequential accesses, SSD has a clear performance advantage [113, 21]. As a consequence it is widely used as cache between main memory and hard-disk-based storage in various systems. Flashcache [97], developed by Srinivasan et al. as a write-back block-level cache, is available in the latest Linux distributions. Any requests larger than 4KB are passed through the cache, which is managed using either FIFO or LRU-based cache replacement policies. There are similar tools in Sun Solaris [73] and Microsoft Windows [20] to reduce the perceived time to power up the disk, launch programs, and write data to the disk. To take account of relatively small SSD capacity, some systems selectively cache small files, metadata, executables, and shared libraries at the file level, such as Conquest [104], or at the data block level as does SieveStore [89]. In contrast, iTransformer uses a small SSD space only as the extension of the scheduling queue to more effectively exploit spatial locality. iTransformer does not rely on caching a large amount of data, or strong temporal locality in the workload, for high-performance. Instead, it leverages a relatively small cache space for improving spatial locality. This can be especially meaningful in the HPC environment, where strong temporal locality in storage access is not common.

SSD-based hybrid storage systems integrate a SSD and a hard disk as one block device. Users can partition the device and access data on it as an ordinary block-level storage device. Combo Drive uses a hardware-based solution to concatenate an SSD and a hard disk via a SATA-to-2xSATA chip [87]. Bisson et al. proposed to issue flash-backed I/O requests to reduce the number of I/O writes to the hard disk by maintaining two duplicated request queues in both main memory and SSD devices [38]. However, the large amount of memory required for maintaining the queue is usually undesirable for HPC systems. In the I-CASH work, the authors proposed a new hybrid storage architecture based on data-delta pairs to improve I/O performance for I/O-intensive workloads [90]. Chen et al. designed the Hystor kernel module, which provides a software-based solution to implement a hybrid storage device [46]. Hystor identifies performance-critical data blocks on the hard disk and stores them on SSD for future accesses. Unlike these works, iTransformer does not seek to use the SSD as fast storage for holding data. Instead, it buffers the data transferred between the memory and disk only for improving locality in the live request streams dispatched to the disk. For applications working with large data sets, the approach taken by iTransformer allows the SSD space to be used more cost-effectively.

2.5 QoS Support in Shared Storage Systems

I/O-intensive applications are becoming increasingly common on today's high-performance computing systems. And provision of QoS guarantees to high-performance applications, such as climate and weather forecasting [60] and modeling financial data [36], can be critical to the success of the services provided to their users. While the performance of compute-bound applications can be effectively guaranteed with techniques such as space sharing or QoS-aware process scheduling, it remains a challenge to meet QoS requirements for end users of I/O-intensive applications using shared storage systems because of the difficulty of differentiating I/O services for different applications with individual quality requirements. Furthermore, it is difficult for end users to accurately specify performance goals to the storage system using I/O-related metrics such as request latency or throughput [62, 57, 58]. As access patterns, request rates, and the system workload change in time, a fixed I/O performance goal, such as bounds on throughput or latency, can be expensive to achieve and may not provide performance guarantees such as bounded program execution time. Zhang et al. [115, 120] proposed a machine-learning based scheduling scheme for shared storage clusters to automatically guarantee end-uses's QoS goals, specified in terms of program execution time. QBox [95] was recently proposed to guarantee I/O performance for black box storage system using a utilization-based approach.

Chapter 3

Opportunistic Data-driven Execution of Parallel Programs

3.1 Introduction

The trend for high-performance computation to be increasingly data-intensive makes the storage system the performance bottleneck in important application areas such as astrophysics, climate, and earthquake simulations. In general, when a system resource becomes a parallel program's performance bottleneck a better scheduling policy is sought to alleviate it. If the resource is the processors this can be a process scheduling strategy for load balancing or co-scheduling [84]. If the resource is storage this could be an optimized I/O scheduler [64, 114]. In today's systems, when I/O service on the storage system becomes a program's bottleneck the process scheduler becomes less relevant. This is especially true when I/O requests are mostly synchronous because most of the time processes are idle waiting for the completion of their I/O requests and their scheduling is essentially a passive reaction to the progress of I/O operations.

Over decades the I/O stack, through which I/O requests pass and are serviced, has been significantly optimized, such as by forming larger sequential requests [100], hiding I/O time behind compute time with conservative I/O prefetching [108, 39], or increasing the parallelism of data access with parallel file systems [17, 19]. However, in these efforts the way in which processes are scheduled for execution is not examined for its effect on I/O efficiency. I/O requests are issued by processes and the requested data are consumed by processes. Therefore, the order in which the requests are issued and served can be significantly influenced by process scheduling. When a process is driven by its computation, the computation determines the request issuance order, which can directly affect the request service order and I/O efficiency. The throughput of a hard disk for serving sequential requests can be more than an order of magnitude greater than that for serving random requests. When I/O is the bottleneck the process scheduler is essentially in standby status and I/O request issuance order is a critical factor for improving I/O efficiency and alleviating the bottleneck.

We propose a data-driven execution mode for parallel programs that is enabled when I/O becomes the bottleneck and I/O efficiency is being compromised by request issuance order. In this mode a process is scheduled to resume its execution not when the request it is currently blocked on is completed, but when the data that it and its peer processes are anticipated to read has also been prefetched into the buffer cache, or the data to write are buffered in the cache. This allows the processes to run longer before they block on a new I/O request. In the data-driven mode we not only require requests of a process be served in a batch, but also coordinate the serving of requests from different processes, because requests from different processes may disruptively compete for disk service and degrade disk efficiency [76, 118]. Furthermore, this coordination creates an opportunity to further improve the request issuance order to increase access sequentiality and to reduce the number of requests.

In summary, we make the following contributions.

- We propose a new program execution mode in which the scheduling of processes can be explicitly adapted for I/O efficiency. To this end, we use pre-execution to predict data to be requested for prefetching and a client-side buffer to hold written data for efficient writeback. Thus the computation of the program can be decoupled from the issuance of requests for its needed data and the I/O bottleneck can be alleviated by having a large space for optimizing request issuance order for high disk efficiency, which cannot be achieved by conventional disk schedulers.

- We design algorithms, comprising DualPar, to detect the condition for enabling and disabling the data-driven mode and to coordinate data access and process executions.

- We implement these algorithms in the MPICH2 MPI-IO library for MPI programs. We evaluate it with representative benchmarks, including *mpi-io-test*, *ior-mpi-io*, *BTIO*, and *S3asim*. Experimental measurements on a large-scale cluster show that I/O efficiency can be significantly improved.

The rest of this paper is organized as follows. Section 3.2 provides a motivating example to reveal the potential and design issues of DualPar. Section 3.3 describes the design and implementation of DualPar. Section 3.4 describes and analyzes experimental results.

3.2 A Motivating Example

To investigate the potential performance benefit of data-driven execution, we experiment with three strategies imposing different relationships between data access

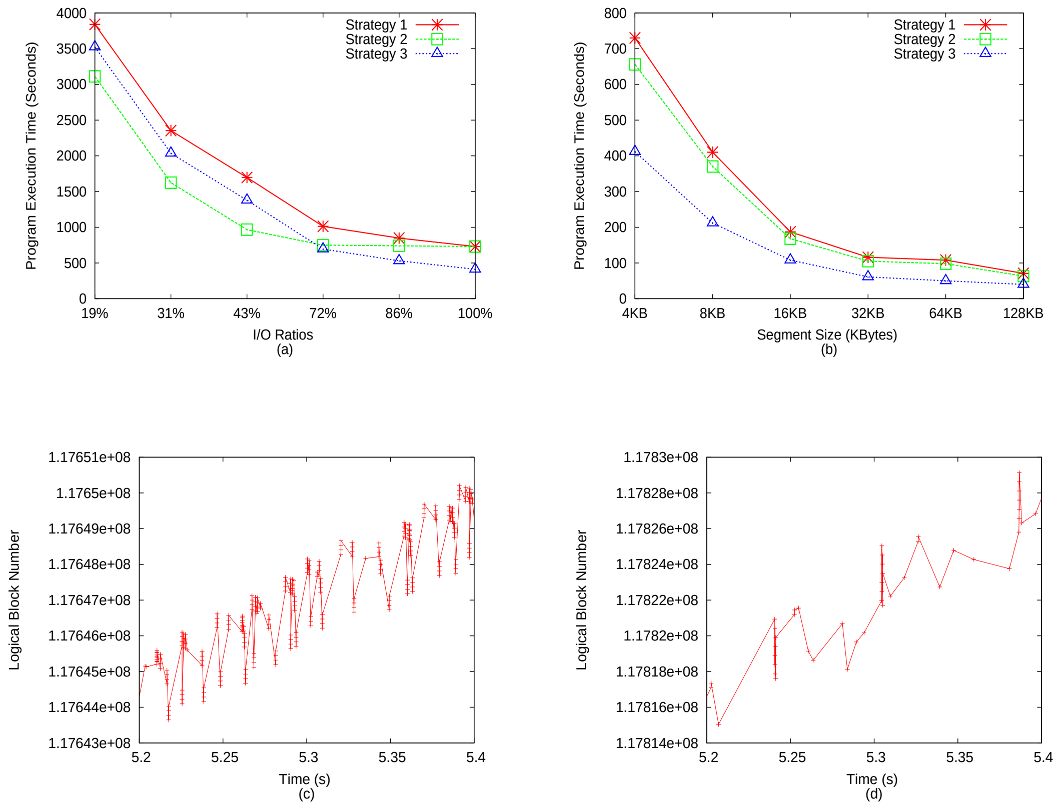


Figure 3.1: (a) Throughputs of the ***demo*** program with different I/O ratios. (b) Throughputs with different segment sizes (or request sizes). (c) Disk addresses (LBNs) of data access on the disk of data server 1 in the execution period 5.2 s to 5.4 s with ***Strategy 2***; (d) Disk addresses of data access on the same data server with ***Strategy 3***.

(using *read* access as example in this discussion) and process scheduling for an MPI program on a cluster. The first strategy is conventional computation-driven execution wherein process execution is fully coupled with data access. Without system-level prefetching triggered by fully sequential data access, a process issues its synchronous read requests one at a time and there is no overlap between computation and data access. This strategy serves as a baseline for evaluation of other strategies. The second strategy is application-level prefetching, which uses pre-execution to generate I/O requests ahead of those produced by the actual computation [108, 43]. A prefetch request is issued to the data servers immediately after it is generated. The objective of this strategy is to hide the I/O time behind the computation through data prefetching. If the computation time is large enough to cover the I/O time, the process can run without being blocked by its requests. However, when the process is highly I/O intensive, the I/O time cannot be hidden and the process must block at some point. The third strategy entails suspending all processes of a parallel program. Pre-executing the processes will generate a batch of I/O requests, which are collected and then sorted to issue to the data servers to prefetch. When the data is available in the buffer cache of the compute nodes where the program runs, the processes are released to consume the data. This comprises one cycle of data-driven execution of the program. In this strategy overlap of computation and I/O service times is not sought and the objective is to improve I/O efficiency through application-level request ordering. In this way the order of I/O request issuance is minimally affected by computation order and process scheduling is driven by the availability of the requested data in a cycle-by-cycle fashion. Table 3.1 summarizes the features and objectives of the three strategies.

	Strategy 1	**Strategy 2**	**Strategy 3**
Computation & I/O overlapped?	No	Yes	No
Correlation between Computation and I/O service order	Strong	Strong w/high I/O intensity	Minimal
Advantage to I/O	Baseline	Hides I/O time	Improves I/O efficiency

Table 3.1: Comparison of three strategies on how I/O request service is correlated to process scheduling.

To examine their I/O performance disparity, we designed an experiment in which we ran a synthetic MPI program *demo*. The program ran on a cluster of 120 nodes, nine of which are configured as data servers and managed by the PVFS2 parallel file system (More details of the platform are given in Section 3.4). In *demo* each process reads a number of noncontiguous data segments of a file in each MPI-IO function call. Specifically, we ran $N = 8$ processes to read a file of 10 GB from its beginning to its end. Each process, identified by its rank *myrank*, reads 16 data segments at offset $k * N + myrank$ ($0 \leq k < 16$), respectively, in each call using the derived *Vector* datatype. The size of the segment varies from 4 KB to 128 KB. The compute time in each process between consecutive I/O operations is adjustable to generate workloads of different I/O intensity, which is quantified as an I/O ratio, the ratio between a program's I/O time and its total execution time in the vanilla system. We compare the program's execution times with the three strategies as a function of I/O ratio and segment size. In the simulation of Strategy 2 we use the approach suggested by Chen et al. [108] for excluding unnecessary computation in the pre-execution. Accordingly we remove all the computation from the pre-execution for Strategy 2. Strategy 3 is employed to suggest the performance potential of *DualPar*, which performs computation in pre-execution for faithful emulation of normal execution for prediction accuracy and tolerance of unavailability of the program's source code. Therefore Strategy 3 also performs the computation in the pre-execution.

Figure 3.1(a) shows the execution times of *demo* with the I/O ratio varying from around 20% to nearly 100% with the segment size fixed at 4 KB. When the I/O ratio is small the I/O time with Strategy 2 is the shortest because it can nearly or completely hide I/O times. Strategy 3 increases the execution time because it blocks the main processes and runs the computation in pre-execution; the time saved by improved I/O efficiency cannot offset the loss due to redundant computation. However, when the I/O ratio increases beyond a certain value (around 70% in this example), the performance advantage of Strategy 2 diminishes and the advantage of Strategy 3 becomes significant. When the ratio is close to 100%, the execution time with Strategy 3 is 36% less than the others. The more I/O-intensive a program, the more critical I/O efficiency is and the more potential for this strategy to improve performance, as evidenced in Figure 3.1(a).

Figure 3.1(b) shows the execution times of the program with different segment sizes and a fixed I/O ratio of 90%. The smaller the segment is, the larger the program's execution time. When the segment size is 4 KB, throughput of the program with Strategy 2 is 16 MB/s, which is 64% of that with Strategy 3 at 25 MB/s. When the segment size is sufficiently large (beyond 32 KB), the advantage of Strategy 3 is less impressive. As we know, for workloads consisting of large requests, a disk's efficiency can be maintained even if these requests are not contiguous on the disk. However, when requests are small the disk scheduler plays a critical role in exploiting spatial locality among them by creating an efficient service order. An interesting observation is that the disk scheduler in the kernel is not effective in creating such an order, even though requests are issued to the scheduler as soon as they are generated by the pre-execution, as shown in Figure 3.1(b).

To reveal the actual service order generated by the disk scheduler, CFQ, under Strategies 2 and 3, we tracked the accessed disk addresses using the Blktrace tool [6]. Figure 3.1(c) and Figure 3.1(d) show sample access sequences reported by Blktrace on a particular data server during the execution period from 5.2s to 5.4s under Strategies 2 and 3, respectively. Strategy 2 produces more short sequences growing in opposite directions, implying back-and-forth movements of the disk head. In contrast, with Strategy 3 the disk scheduler does a much better job by efficiently moving the disk head mostly in one direction. The difference is due to the action taken in Strategy 3 to enable the disk scheduler to better exploit the spatial locality in the request stream.

However, why does the disk scheduler not create an efficient request dispatch order for Strategy 2? This is because there can still be time gaps between consecutive requests issued during the pre-execution of Strategy 2. Furthermore, requests from different processes can arrive at a data server in an essentially random order. Consequently the disk scheduler sees a limited number of outstanding requests to schedule and has difficulty identifying a long access sequence from them. In Strategy 3 a large number of requests from different processes is collected. They are then sorted and merged at the file-abstraction level and issued to the data servers together. In the experiment the average request size is 128KB for Strategy 3 and 12KB for Strategy 2. Usually there is a good correspondence between file-level and disk-level addresses, so the disk scheduler is more likely to identify strong locality in the large pre-sorted file-level requests.

3.3 The Design of DualPar

The objective of DualPar is to opportunistically change programs' execution to the data-driven mode when they are I/O bottlenecked and their performance is compromised by low I/O efficiency. In this section we describe DualPar's architecture as well as how it determines the execution mode, performs pre-execution, manages the cache for I/O data, and generates read or write requests to the data servers. In the design we target MPI programs [12].

3.3.1 Overview of the System Architecture

A cluster consists a number of compute nodes running MPI programs, and data servers providing I/O services to the programs. In addition, parallel file systems such as PVFS2 and Lustre [17, 19] have a metadata server to provide metadata service for data access. DualPar has three major system modules, Execution Mode Control (EMC), Process Execution Control (PEC), and Cache and Request Management (CRM). EMC resides on the metadata server and dictates the current execution mode for any programs that have registered for dual-mode execution. PEC is built into the MPI-IO library and runs with each MPI program to track the processes' I/O activities and enforce the data-driven mode, e.g. blocking/resuming the (main) process and creating pre-execution processes for generating prefetch requests. CRM is on each compute node for collecting, sorting, and dispatching prefetch requests as well as caching requested data. Figure 5.1 depicts DualPar's architecture. When it is in the normal computation-driven mode, PEC and CRM are in standby status and only EMC is actively monitoring the programs' execution and I/O efficiency on the data servers.

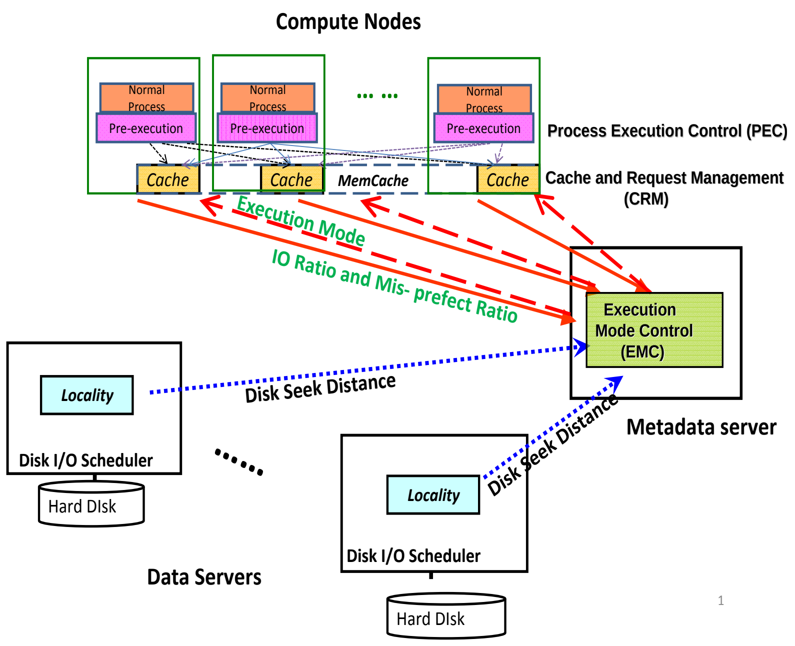


Figure 3.2: DualPar's architecture, in which EMC takes inputs from both the data servers and compute nodes to choose the execution mode for each program. If the mode is data-driven PEC is activated to implement the mode at each process of the program through coordination of process execution and request service. CRM manages the global cache to support application-level request scheduling and buffering requested data.

3.3.2 Determining Program Execution Mode

Whether a program should switch into the data-driven mode depends on its I/O intensity and current I/O efficiency. Only when the intensity is high and the efficiency is low can the data-driven mode help, rather than hinder, performance. As the decision is made for a parallel program rather than for its individual processes, we place the decision maker, the execution mode control daemon (EMC) at the metadata server, which constantly interacts with the compute nodes. To allow the processes to track I/O intensity and report to the EMC daemon, we instrument the ADIO functions in the MPI-IO library of MPICH2 [12], such as *ADIOI_PVFS2_Open*, *ADIOI_PVFS2_Close*, *ADIOI_PVFS2_ReadContig*, *ADIOI_PVFS2_ReadStrided*, *ADIOI_PVFS2_WriteContig*, and *ADIOI_PVFS2_WriteStrided*, enabling measurement of I/O times and computation times to calculate the I/O ratio. We treat the time between any two consecutive I/O-related function calls as computation time. Though the computation time may include the time a process is de-scheduled for running other processes, and communication time, the measurement serves the purpose for evaluating I/O intensity well by precisely measuring the I/O component of user-observed program execution time. Because the measurements are taken with the occurrence of expensive I/O operations, their overhead is negligible.

Another input to the EMC daemon is the current I/O efficiency. As data servers are shared to serve I/O requests from different programs, analysis of individual programs' access patterns to estimate I/O efficient is not accurate. In addition, analysis conducted only at the compute node cannot faithfully capture the interaction among requests served at data servers, especially the interference between requests from different programs. Therefore, we set up a *locality* daemon at each data server that tracks disk head seek distance, *SeekDist*, a parameter maintained in the Linux kernel for I/O request scheduling [82], and use it as a metric for quantifying I/O efficiency—the smaller *SeekDist*, the higher the I/O efficiency. At the same time we record requests observed at each of the compute nodes running programs in constant time slots, sort requests for data from the same file according to their file offsets, and calculate the average distance (*ReqDist*) between adjacent requests. *ReqDist* represents the highest I/O efficiency that a data-driven execution can possibly achieve. When the EMC daemon receives these two types of distance values from compute nodes and storage nodes, it uses the ratio of their respective averages ($aveSeekDist/aveReqDist$) as potential I/O efficiency improvement to determine whether it is larger than a predefined threshold $T_{improvement}$. I/O efficiency improvement is a system-wide metric. Once it is larger than the threshold, EMC will instruct the programs whose I/O intensities are sufficiently large (larger than 80% in our prototype) to switch to the data-driven mode. When the condition no longer holds, EMC reverts the program to the normal mode. The default $T_{improvement}$ value is 3 in our prototype system. Our experimental results show that system performance is not sensitive to this threshold.

3.3.3 Pre-execution for Predicting Future Requests

The key idea of data-driven execution is to decouple computation and I/O service into alternating time phases (computation phase and I/O phase) so that a batch of requests can be served in an optimized order and the computation is driven by the completion of the data service. We set up a buffer cache for each process on its compute node, and data service is either for the processes to write data to the cache or for the prefetcher to read data from the data servers into the cache. When caches assigned to every process of a program are filled, any dirty data will be first written back and then the program will be allowed to resume execution using the prefetched data, if any.

It is straightforward for write requests to leave dirty data in the buffer cache. For read requests DualPar must use pre-execution to predict a process's future read requests and prefetch them into the cache. When a program has been instructed to enter the data-driven mode and any of its processes calls a synchronous read function, the process's control passes to the MPI-IO library that we have instrumented. DualPar does not issue the corresponding read request to the data server if the request cannot be served by the local cache. Instead it suspends the function call and forks a ghost process to keep running on behalf the normal process. As DualPar knows the identity of the ghost process, the instrumented library can distinguish I/O calls issued by a ghost process from normal calls. DualPar records these calls without immediately turning them into requests to issue. It also tracks the total space that would be consumed by the calls if their corresponding requests were served. If the space reaches the reserved cache size for the process, pre-execution pauses. When the pre-execution of every process is paused, DualPar resumes the execution of the program's processes. In case that the I/O demands of different processes are different and some processes may take relatively longer to fill their caches, we use the processes' recent average I/O throughput and reserved cache size to calculate the expected time to fill the cache. When the time period expires, all unfinished pre-executions are stopped.

In DualPar a ghost process carries out all of the computations encountered in its execution, so DualPar does not require a modification to the program, which in most cases is practical only when its source code is available. Because a program in the data-driven mode has a high I/O ratio the overhead for the computation is limited. Though normal processes' communications and computations are retained, the requests generated by the pre-executions can still be wrong if their data locations depend on the values of data requested in the previous function calls, because requests in the pre-execution are not immediately served. This inaccuracy does not affect the correctness of the original program: the only consequence is mis-prefetching that will be detected when the processes are resumed to access the prefetched data. We define the mis-prefetch ratio as the fraction of prefetched but not used data in a cache when the next pre-execution begins. We pass this ratio to the EMC daemon. The daemon calculates the average value of the ratios reported by each program's processes and disables the data-driven mode if its average mis-prefetch ratio is larger than a threshold, 20% by default in the prototype.

3.3.4 Global I/O Cache for Serving Requests

DualPar maintains a cache for each process in data-driven mode. To make efficient use of the caches they must be shared to avoid redundancy and consistency issues. To achieve this all the caches form a global cache and are managed by Memcached [11]. Memcached provides a high-performance distributed cache infrastructure as an in-memory key-value store. A file is partitioned into chunks of equal size. Every data chunk is indexed by a unique key generated from the name of the file that stores the chunk and chunk address in the file. The caches assigned to individual processes on each compute node are actually managed by Memcached. A file's chunks are stored in the caches of different compute nodes in a round-robin fashion. In DualPar we set the chunk size to be the unit size used in the PVFS2 parallel file system for striping files across data servers, 64KB by default, so that a chunk can be efficiently accessed by touching only one server. Each chunk in the global cache has a tag to record the time of its most recent reference. A chunk will be evicted if it is not used for a certain period of time. For the I/O calls from normal processes in the data-driven mode, the instrumented I/O library will direct the requests to the global cache. A read miss on the cache will block the process and a pre-execution for the process will be initialized.

As mentioned, DualPar records read and write requests in its pre-execution phase without immediately serving them. Reads will be collectively served at the end of the pre-execution phase and writes will be collectively served at the end of normal execution. In the service of either read or write requests, requests from different processes belonging to the same program are sorted, and adjacent requests are merged. If there are small numbers of holes between the requests, which are not accessed by any of the current requests, for writes the data in the holes will be filled by additional reads before writing to disks, and for reads the data in the holes are added to the requests. This further helps form larger requests. In addition, we use list I/O [47] to pack small requests and issue them in ascending order of the requested data's offsets in the files to improve disk efficiency.

3.4 Performance Evaluation and Analysis

Most of the DualPar implementation is in the MPI-IO library of MPICH2 software [12] as instrumentation of ADIO functions. We place daemons on the data servers for measuring disk efficiency and on the metadata server for determining program execution mode. We evaluated the performance of DualPar on the Darwin cluster at Los Alamos National Laboratory. The cluster consists of 120 nodes, nine of which were configured as data servers, one of which was also configured as a metadata server of the PVFS2 parallel system (version 2.8.2) [17]. Of the 120 nodes, 116 are 48-core (12 core by 4 socket) 2GHz AMD Opteron 6168, and are the nodes on which our experiments were performed. Each node has 64 GB memory, a hardware-based RAID 0 consisting of two 7200-RPM disk drives (HP model MM0500FAMYT). Each server ran Fedora Linux, kernel-2.6.35.10, with CFQ (the default Linux disk scheduler). We used MPICH2-1.4 with ROMIO to generate executables of MPI programs. All nodes were interconnected with a switched Gigabit Ethernet network. Files were striped over data servers with a 64 KB unit in the PVFS2 file system. To ensure that all data were accessed from the disk, the system buffer cache of each compute node and data server was flushed prior to each test run. For write tests we forced dirty pages to be written back every second on each data server. PVFS2 does not maintain a client-side data cache. As the benchmarks used in the evaluation have no or little data reuse, selection of a different file system with client-side caching does not reduce the performance advantage of DualPar presented here. For the cache infrastructure the latest stable Memcached release v1.4.7 was used [11]. Unless otherwise specified, each process has 1MB quota in the cache.

3.4.1 Benchmarks

We use six benchmarks with distinct data access patterns.

mpi-io-test is an MPI-IO benchmark from the PVFS2 software package [17]. In our experiments we ran the benchmark to read or write a 20 GB file with request size of 16 KB. Process p_i accesses the $(i + 64j)^{th}$ 16 KB segment at call j ($j \geq 0$), for $0 \leq i < 64$. The benchmark generates a fully sequential access pattern.

hpio is a program designed by Northwestern University and Sandia National Laboratories to systematically evaluate I/O performance using a diverse set of access patterns [50]. We used it to generate contiguous data accesses by changing parameters such as *region_count*, *region_spacing*, and *region_size*. We set *region_count* to 4096 B, *region_spacing* to 1024 B, and *region_size* to 32 KB.

ior-mpi-io is a program in the ASCI Purple Benchmark Suite developed at Lawrence Livermore National Laboratory [4]. In this benchmark each MPI process is responsible for reading its own 1/64 of a 16 GB file. Each process continuously issues sequential requests, each for a 32 KB segment. The processes' requests for the data are at the same relative offset in each process's access scope of 256 MB. The program's access pattern as presented to the storage system is random.

noncontig is a benchmark developed at Argonne National Laboratory and included in Parallel I/O Benchmarking Consortium [15]. This benchmark uses MPI processes to access a large file with a vector-derived MPI data type. If we consider the file to be a two-dimensional array, there are 64 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* (MPI_INT). If collective I/O is used, in each call the total amount of data read by the processes is fixed, which is 4 MB in our experiments. We also set the data access pattern of *noncontig* to be non-contiguous for both memory access and file access.

S3asim is a program designed to simulate sequence similarity search in computational biology [18]. In the experiments, the sequences in the database are divided into 16 fragments. We set the minimal size of each query and database sequence to 100 B and the maximal size to 10,000 B. The amount of data accessed depends on the counts of queries. In the experiments the amount of data that is written during execution is up to 6.4 GB.

BTIO is an MPI program from the NAS parallel benchmark suite [14] designed to solve the 3D compressible Navier-Stokes equations using the MPI-IO library for its on-disk data access. We choose to run the program with an input size coded as *C* in the benchmark, which generates a data set of 6.8 GB. The program can be configured to use either non-collective or collective I/O functions for its I/O operations.

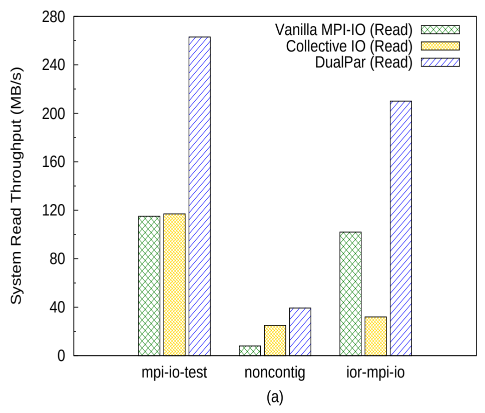
Each of the benchmarks was run with 64 MPI processes unless otherwise specified. These benchmarks cover a large spectrum of access behaviors, from sequential access among processes (e.g. *mpi-io-test* and *hpio*) to non-sequential access among processes (e.g. *noncontig* and *BTIO*), from read access to write access, from requests that are well-aligned with the 64 KB striping unit size (e.g. *mpi-io-test* and *ior-mpi-io*) to requests of different sizes (e.g. *S3asim* and *BTIO*).

3.4.2 Performance with a Single Application

We first evaluate how DualPar can improve I/O performance when only one MPI program is running so that the I/O resource is not competed among multiple programs. In this experiment we use one program with sequential access, *mpi-io-test*, and two programs with non-sequential access, *noncontig* and *ior-mpi-io*, to benchmark performance of *vanilla MPI-IO*, *collective IO*, and DualPar. We choose *collective IO* for comparison as it shares a similar principle with DualPar by holding processes and re-arranging requests at the application level. For execution with DualPar, programs stay in the data-driven mode. As the programs can be configured to issue either read or write requests, both of read and write accesses are tested. Because DualPar aims to increase a program's performance by improving I/O efficiency, we use the system's I/O throughput as the metric to demonstrate the differences among the schemes. The experiment results are shown in Figure 3.3.

The sequential access pattern of *mpi-io-test* can potentially make efficient use of the disks if many I/O requests arrive at the disks together. However, because a barrier routine is frequently called in its execution, and each barrier operation takes a relatively long time with a large number of processes, the I/O requests cannot reach the disk scheduler at data servers in large numbers for it to effectively schedule. When *DualPar* is enabled, requests from prefetch threads quickly saturate the storage device, resulting in a 263 MB/s throughput, which nearly doubles the 115 MB/s throughput with *vanilla MPI-IO* and the 117 MB/s throughput with *collective IO*. For the program using write requests, DualPar significantly increases the throughput by having many writeback requests issued from the cache together for which the disk scheduler can effectively exploit access locality.

For program *noncontig*, which accesses noncontiguous data, both *DualPar* and



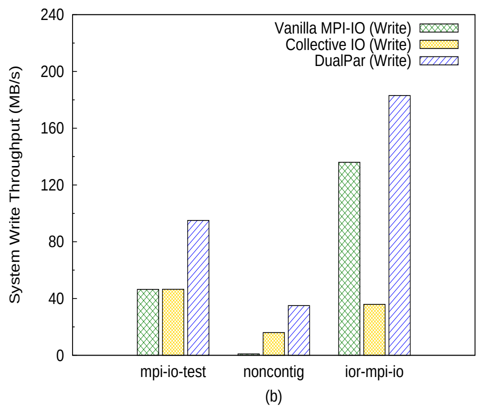


Figure 3.3: System I/O throughput when a single program instance is executed with *vanilla MPI-IO*, *collective IO*, and DualPar, respectively. In (a) the programs issue read requests, and in (b) they issue write requests.

collective IO achieve significantly higher throughput than *vanilla MPI-IO*, as both can transform overlapped noncontiguous accesses into larger contiguous accesses for efficient disk access. Furthermore, DualPar sorts requests and forms large requests among data in the buffer cache, while *collective IO* does its optimization in the data domain covered by the requests in a single collective IO routine. As the data domains are usually much smaller than the buffer cache, DualPar can be more effective in producing requests of strong locality. As such, the benchmark with DualPar achieves 39 MB/s read throughput, a 57% improvement over that with *collective IO*.

For program *ior-mpi-io*, the advantage of *collective IO* is lost even though the program issues random requests. This is because of a mismatch between the data request pattern and the file striping pattern, which keeps only one or two data servers busy serving requests in each collective call [119]. While *collective IO* synchronizes processes at every collective call, the number of outstanding requests is very limited and the load imbalance on the data servers cannot be corrected. In comparison, *DualPar* achieves a much higher throughput as it actually carries out the service of a large number of requests accumulated at each process's cache together. By collectively serving the requests at the end of a pre-execution phase (for read) or at the end of a normal execution phase (for write), a large number of requests spread over all of the data servers keeping every disk well utilized. The improvement of DualPar over *vanilla MPI-IO* is 105% for reads and 35% for writes because disk efficiency is greatly improved with workloads with sequential accesses and larger requests.

3.4.3 Performance with Concurrent Applications

The I/O efficiency of a program changes when it runs with other I/O-intensive programs sharing the same data servers because the interference caused by concurrently serving their requests can reduce disk efficiency. In this section we evaluate system performance with DualPar in such scenarios with *BTIO*, *S3asim*, and *mpi-io-test*.

We first concurrently run three *BTIO* programs, each accessing its own 6.8 GB file on the data servers. Figure 3.4 shows system throughput when the number of processes increases from 16 to 256. As shown in Figure 3.4, throughput with *collective IO* and DualPar outperforms that with vanilla MPI-IO by up to 24X and 35X, respectively. This is because without using techniques for I/O optimization, the request size of the benchmark is only 400 B when 256 processes are used, which is too small for the disks to be efficiently used. Both DualPar and *collective IO* can help transform noncontiguous requests into much larger ones. The figure shows that the performance advantage of *collective IO* gradually decreases when more processes are used. This is because the size of data domain accessed by one *collective I/O* routine does not increase with more processes, making *collective I/O* increasingly expensive because more data exchanges are needed among the processes. DualPar has better scalability than *collective IO* as more requests from a larger number of processes can

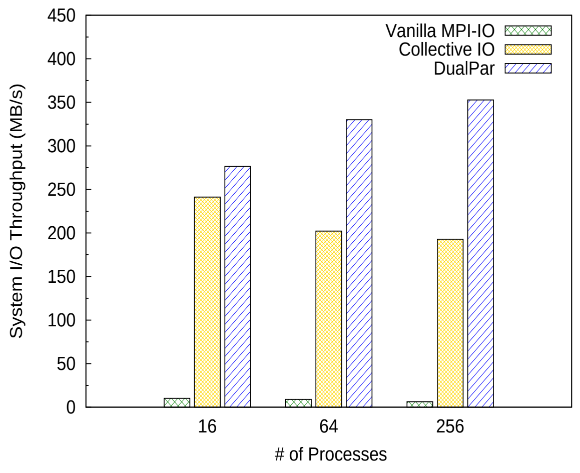


Figure 3.4: System I/O throughput with three concurrent instances of *BTIO*. We compare throughput with DualPar to those with *vanilla MPI-IO* and *collective IO* as we increase process parallelism from 16, 64, to 256.

quickly fill up the buffer cache in the data-driven mode, resulting in a better use of the storage system.

Next we executed three concurrent instances of *S3asim* to benchmark the system with *vanilla MPI-IO*, *collective IO*, and DualPar, with different numbers of queries. As shown in the Figure 3.5, the total I/O times with DualPar reported by the program are smaller by up to 25%, and by 17% on average, than those of the other two schemes. *DualPar*'s performance advantage for this benchmark is much smaller than that for *BTIO* because *S3asim*'s requests are much larger than *BTIO*'s requests.

Finally, we ran two instances of *mpi-io-test* with 16 KB segment size. Each instance accesses its own 20 GB file. Both read and write tests were carried out with *vanilla MPI-IO*, *collective IO*, and *DualPar*. Table 3.2 shows system I/O throughput under different scenarios. The measurements demonstrate consistent performance advantages of DualPar over the other two schemes. This is because of its ability to

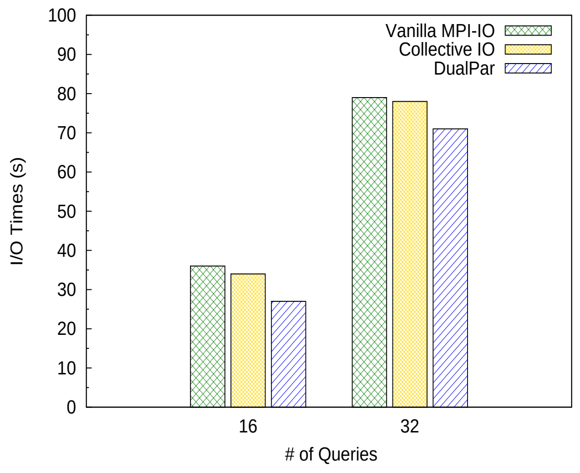


Figure 3.5: I/O times with three concurrent instances of *S3asim*. We compare I/O times with DualPar to those with *vanilla MPI-IO* and *collective IO* as we increase number of queries from 16 to 32.

	Vanilla MPI-IO	Collective IO	DualPar
Throughput (Read)	160 MB/s	168 MB/s	284 MB/s
Throughput (Write)	54 MB/s	67 MB/s	127 MB/s

Table 3.2: Aggregate I/O throughput when two *mpi-io-test* program instances are concurrently executed with *vanilla MPI-IO*, *collective IO*, and DualPar, respectively.

accumulate, sort, and merge requests at the application level so that requests can arrive at data servers in a bursty manner and with an optimized order for efficient disk access. To confirm this analysis, we profile disk addresses in terms of logical block numbers in the order that they are accessed at a particular data server (Server 1) in a randomly selected 1 s execution period with 16 KB requests. The result is shown in Figure 3.6 for *vanilla MPI-IO* and DualPar. With *vanilla MPI-IO*, the disk head must frequently move between regions of data of the two files, frequently and at long

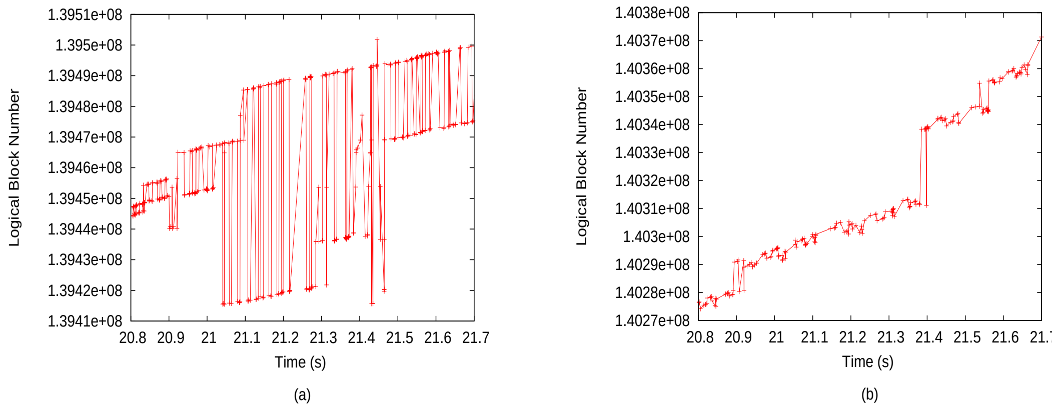


Figure 3.6: Disk addresses (LBNs) of data access on the disk of Server 1 in a 1 s sampled execution period. (a) Two *mpi-io-test* instances are executed using *vanilla MPI-IO* reads in the vanilla system; (b) Two *mpi-io-test* instances are executed with DualPar. Note that because the instances ran faster in DualPar, the disk positions shown in the two figures can be different during the same execution period.

distance, significantly reducing disk efficiency. The disk scheduler does not help in this case because the number of requests outstanding in its I/O queue is usually not large enough for it to effectively sort and merge requests. DualPar helps reduce the average disk seek distance by up to 10X.

3.4.4 Performance with Varying Workload

DualPar is designed to opportunistically take advantage of the data-driven mode according to current I/O efficiency and intensity. In this experiment we change the workload to evaluate how DualPar responds to the dynamic changes as well as its impact on the system performance. We first run *mpi-io-test* at time 0s to read a 20 GB file with 16 KB requests, and at time 5 s *hpio* joins to read a 2GB file with the same request size. We run the experiment with either *vanilla MPI-IO* or DualPar. Figure 3.7(a) shows the I/O throughput during the execution. When *mpi-io-test* is

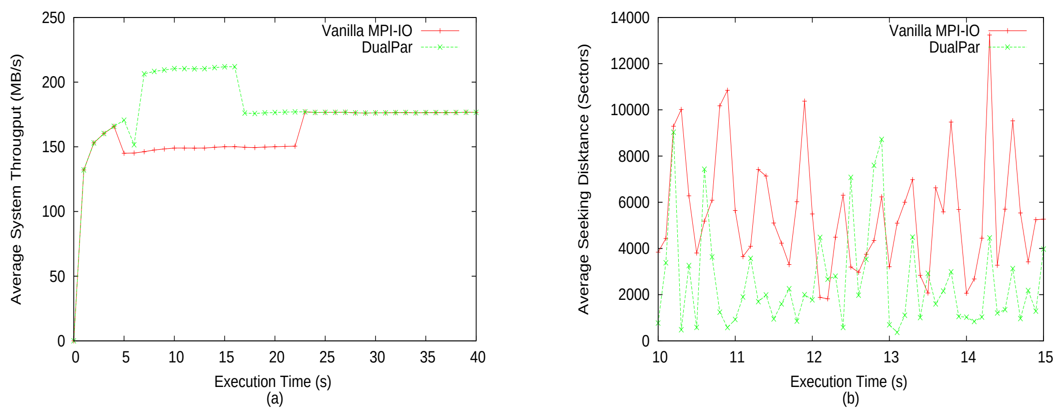


Figure 3.7: (a) System throughputs measured on an execution window of 1 s. (b) Average distances of diskhead seeks on Server 1 when both *mpi-io-test* and *hpio* run concurrently.

the sole workload on the system, the average throughput is 178 MB/s and there is no I/O interference among programs. Since *mpi-io-test* issues sequential requests and I/O efficiency is not an issue before the fifth second, DualPar keeps the program in computation-driven mode. When *hpio* starts to issue requests, the system throughput is reduced with *vanilla MPI-IO*, even though the I/O demand is increased and both programs issue sequential requests, because of the interference between requests from different programs. When DualPar is applied it detects the interference-induced I/O efficiency degradation and instructs both programs to enter data-driven execution mode, which improves the throughput by 46% until *hpio* completes its execution. DualPar measures the I/O efficiency by tracking average disk seek distance. Figure 3.7(b) shows a sample of the distances (in the unit of disk sectors) that are collected on Server 1—seek distances are reduced by DualPar.

3.4.5 Performance with Varying Cache Size

In DualPar each process in data-driven mode is assigned a cache for holding prefetched data and write-back data. To evaluate the effect of cache size on DualPar's performance we ran benchmark *BTIO*, which stays in the data-driven mode in its execution with its non-sequential access pattern. We ran *BTIO* with 64 processes and varied the cache size from 0 KB to 1024 KB. The measured I/O throughputs are shown in Figure 5.9. When the cache size is 0 KB DualPar is effectively disabled and the throughput (2.7 MB/s) is almost the same as that of *vanilla MPI-IO*. When we increase the cache size to 64 KB the throughput is increased by almost 43X. The reason that such a small cache gives such large improvement is that *BTIO*'s original request size is very small at 800 B. Further increasing the cache size brings diminishing returns. While it is true that the larger the cache the larger the requests that can be formed, a too-large cache could consume excessively memory. Fortunately, for most programs a cache of several MB for each process should be sufficient. By default, the size is set to 1 MB in DualPar.

3.4.6 Overhead Analysis

There are two major sources of overhead in DualPar. One is the redundant computation conducted in the pre-execution, and the other is the miss-prefetched data. As the data-driven mode is enabled only for highly I/O-intensive activity the computation overhead is limited. Because of the possibility of data dependency, in the extreme case all prefetched data would be useless, and this would not be detected until normal processing has misses in the cache. To investigate performance effects of the possible overhead, we wrote an MPI program that reads 20 GB data, with the requested data addresses depending on the data read in the previous I/O call. Because

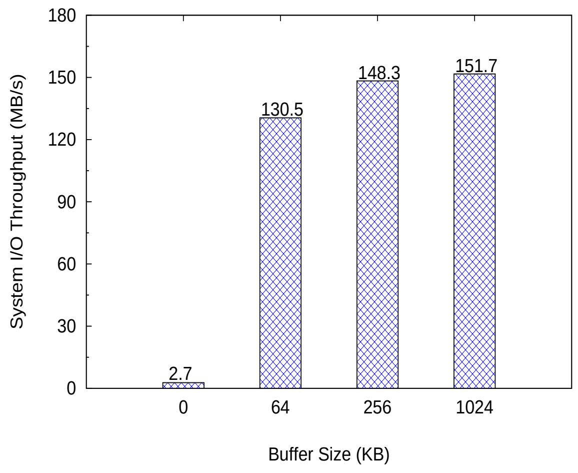


Figure 3.8: System I/O throughput of program *BTIO* measured when the size of the cache for one process varies from 0 KB to 1024 KB.

Cache Size (KB)	No DualPar	1024	2048	4096
Execution Time (s)	138	140	142	148

Table 3.3: The execution times of a program whose future requests cannot be predicted by pre-execution. The times with DualPar and without *DualPar* are presented.

of this dependency all data loaded into the cache are mis-prefetched. We measured the program's execution times without and with DualPar with varying cache size. The results are shown in Table 3.3. Even in the worst scenario the increase of the execution time is very limited: when the cache size is 4 MB, which represents a substantial amount of mis-prefetching, the performance is decreased by only 7.2%. The reason is that a large mis-prefetching miss ratio will turn off the data-driven mode, so this is a one-time overhead.

Chapter 4

Collective I/O with Data Layout Awareness

4.1 Limitations of Current Implementation of Collective I/O

As large-scale scientific applications running on clusters become increasingly I/O intensive, it is important to have effective system support for efficient I/O between the processes on the compute nodes issuing I/O requests, and the disks on the data servers servicing the requests [37, 52, 83]. A problematic situation in I/O performance is the issuance of requests for many small non-contiguous I/O accesses, because unoptimized servicing of these requests results in low disk efficiency and high request processing cost. Many techniques have been proposed to address this problem, including data sieving [100], list I/O [47], datatype I/O [48], and collective I/O [100]. Of these, collective I/O is one of the more commonly used techniques and usually yields the greatest improvement in I/O performance. This is because collective I/O rear-

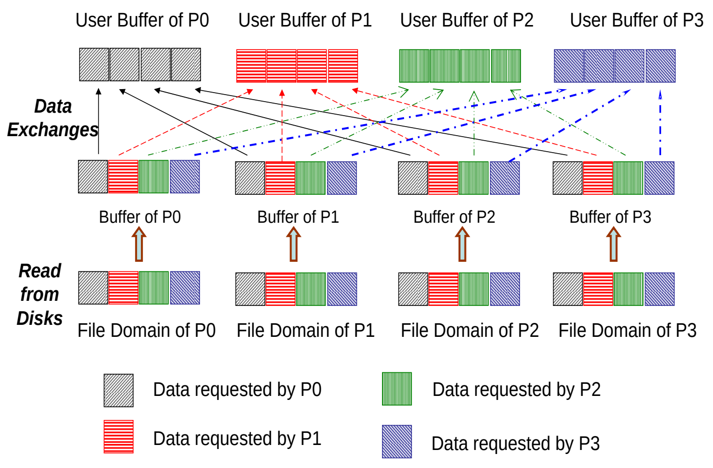


Figure 4.1: Illustration of the ROMIO implementation of two-phase collective I/O. Data is read by each process (the aggregator), P_0, P_1, P_2, or P_3, which is assigned a contiguous file domain in the logical file space, first into its temporary buffer in phase I and then to the user buffer of the process that actually requested them in phase II.

ranges requests from multiple processes (global optimization), rather than optimizing requests from each individual process (local optimization).

4.1.1 Transforming Non-contiguous Access into Contiguous Access

A common technique used in the aforementioned schemes for optimizing I/O performance is to transform small requests of non-contiguous access into large requests of contiguous access. Let us first see how the read operation can benefit from collective I/O. As depicted in Figure 4.1, four processes, P_0, P_1, P_2, and P_3, each requests four segments that are not adjacent in the logical file space. Because an I/O request must be issued for logically contiguous data, each process issues four requests. Without collective I/O there would be 16 small requests from the compute nodes to the data servers, with each data server receiving and servicing four requests in a random order.

With collective I/O, all the requested data is divided into four file domains, each consisting of four contiguous segments, and each process issues a single request to read data belonging to a single file domain into its buffer. After the reads complete, each process retrieves its respective data from the others' buffers via inter-node message passing. As an example of a widely used collective-I/O implementation, ROMIO [100] adopts a two-phase strategy. In the first phase, each process serves as an *aggregator*, with process P_k $(k \geq 0)$ responsible for reading the kth file domain into its buffer. In the second phase, data is exchanged among the processes to satisfy their actual requests. The rationale for this implementation of collective I/O is two-fold. First, both the number of requests issued to the data servers, and the request processing overhead, are reduced. Second, contiguous access is expected to be more efficiently serviced on the disk-based I/O servers than non-contiguous access because contiguous access requires fewer disk head movements, which can account for more than an order of magnitude disparity in disk throughput. Clearly, for collective I/O to improve rather than degrade performance, the gains must outweigh the communication overhead incurred in this second phase that does not exist in the traditional non-collective I/O scheme.

4.1.2 The Resonance Phenomenon

To analyze how collective I/O performs in a typical cluster computing environment, we set up an experimental platform consisting of eight nodes, four configured as compute nodes, and the other four as data servers, managed by a PVFS2 parallel file system [17]. File data was striped over the data servers. We used the default

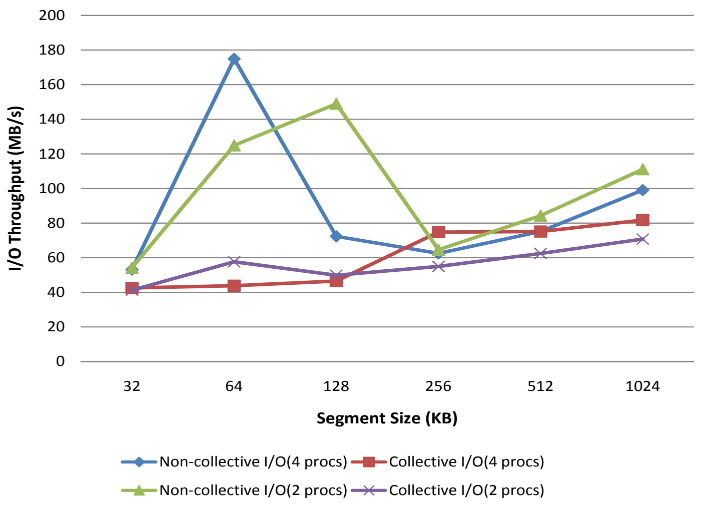


Figure 4.2: Throughput of data servers when running a demonstration MPI program with two and four processes, varying the segment size from 32KB to 1024KB, with and without collective I/O. Throughput peaks at 64KB with *non-collective-4*, and at 128KB with *non-collective-2*, exhibiting *resonance* between the data request pattern and the striping pattern.

PVFS2 striping unit size of 64KB. (More details of the experimental platforms are given in Section 4.3.)

In our experiment we ran N-process MPI programs, where N was 2 or 4, one process per compute node, that read data from a 10GB file striped over the four data servers. The access pattern was generally the same as that illustrated in Figure 4.1. Specifically, the processes repeatedly call collective I/O to read the entire file from beginning to end. In each call, process i, $i \in \{0, 1, \ldots, N-1\}$, reads segments $k*N+i$, $k \in \{0, 1, 2, 3\}$, in the file range specified by the call. The size of the segment was varied from 32KB to 1024KB (powers of two times 32KB) over different runs of the program. Figure 4.2 shows the I/O throughput of the system using collective I/O with N processes and the various segment sizes, denoted as *collective-I/O-N*, where N is 2 or 4. The graph also shows the throughput with N processes when each process makes four distinct I/O calls for each of its four segments of contiguous data, denoted as *non-collective-I/O-N*. As we expect, with *collective-I/O-N*, increasing segment size (amount of requested data) gives increasing throughput. This is consistent with the fact that the disk is more efficient with large contiguous data access because of better amortized disk seek time. Surprisingly, however, we see that the throughput for *non-collective-I/O-4* reaches a peak value of 175MB/s at 64KB segment size, which is much higher than the 42MB/s throughput for *collective-I/O-4* at the same segment size. Similarly, for *non-collective-I/O-2* there is a peak of 149MB/s at 128KB, versus 48MB/s for *collective-I/O-2*. This appears to be inconsistent with the assumption that requests for larger contiguous data would be more efficiently serviced.

The reason for these throughput peaks lies in the order in which the requests arrive at each data server. Figure 4.3 illustrates the different orders when collective I/O is used (Figure 4.3(a)) and is not used (Figure 4.3(b)) in the case of four processes. When collective I/O is used, four contiguous segments are assigned to a process as a file domain. Because both segment size and striping unit size are 64KB, the four requests to a particular data server, each for 64KB data, come from four concurrent processes and arrive in an order determined by the relative progress of the processes, which is unpredictable. In an operation manual for the Lustre cluster file system this issue is raised as a disadvantage of striping a file into multiple objects (the portion of file data on one I/O server). Consider a cluster with 100 clients and 100 I/O servers. Each client has its own file to access. The manual [19] states: *"If each file has 100 objects, then the clients all compete with one another for the attention of the servers, and the disks on each node seek in 100 different directions. In this case, there is needless contention."* This exactly describes the situation with collective I/O when multiple processes access the same file on multiple data servers simultaneously. We note that while the I/O scheduler at the data server can re-order the requests for a sequential dispatching order, this re-ordering operation will rarely occur unless the I/O system is saturated and many requests are pending.

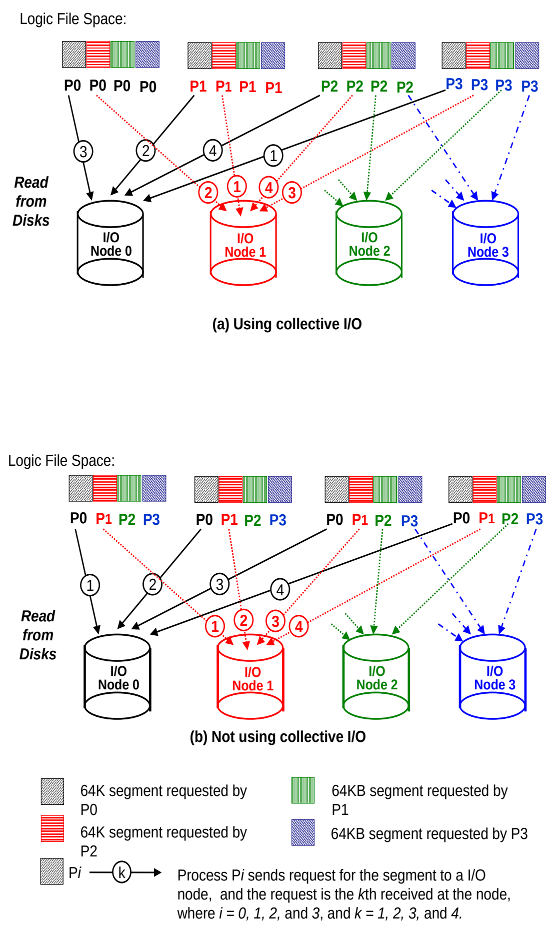

Logic File Space:
P0 P0 P0 P0
P1 P1 P1 P1
P2 P2 P2 P2
P3 P3 P3 P3
Read from Disks
I/O Node 0
I/O Node 1
I/O Node 2
I/O Node 3
(a) Using collective I/O
Logic File Space:
P0 P1 P2 P3
Read from Disks
I/O Node 0
I/O Node 1
I/O Node 2
I/O Node 3
(b) Not using collective I/O
64K segment requested by P0
64K segment requested by P2
64KB segment requested by P1
64KB segment requested by P3
Pi
k
Process Pi sends request for the segment to a I/O node, and the request is the kth received at the node, where i = 0, 1, 2, and 3, and k = 1, 2, 3, and 4.

Figure 4.3: Illustration of how a *resonance* is developed: when collective I/O is used (a), each process reads four contiguous segments, but each data server receives requests from four processes in an unpredictable order. When collective I/O is not used (b), a process sends four requests for four non-contiguous segments to one I/O node, making the service order of the requests at the node consistent with the offsets of the requested data in the file domain.

Therefore, a data server usually serves requests in the order that they are received—in random order from the viewpoint of data server— which degrades disk performance. In contrast, when collective I/O is not used, all four requests to a data server are from the same process, which sends them one by one in the order of their offsets in the logical file space. Because the file system generally allocates data on the disk in an order consistent with their offsets in the file domain, the consequent sequential service order at a data server leads to an effective prefetching at the data server [74]. We name this scenario, in which an accidental match between data request pattern and data striping pattern produces sequential disk access and peak disk performance, *resonance* in the distributed I/O service, a term borrowed from the physics field. A similar resonance exists with *non-collective-I/O-2* with 128KB segment size, in which two data servers are dedicated to service requests from one process (one segment is striped on two nodes), and no data server receives requests from multiple processes that cause random disk accesses. We also observe that *non-collective-I/O-2* with 64KB segment size generates a resonance, though with a throughput (125MB/s) lower than the one (149MB/s) at 128KB segment size. The lower throughput is a result of under-utilized data servers, because at any time only two of the four data servers are servicing requests from the two processes.

Analyzing the conditions for resonance to occur, we see that the key factor for high I/O throughput is not simply accessing a contiguous file domain, rather, it is ensuring sequential access of data on a data server. When data is striped over multiple data servers, collective I/O, which designates one contiguous file domain to a process, allows requests for data on a data server to be from multiple processes, which introduces the indeterminacy that leads to non-sequential access. If we can rearrange requests involved in collective I/O such that all the requests for data on a data server come from one process, then resonance would be a common case when each process requests its data in ascending order of file offset. This is one of the techniques used in our proposed implementation of collective I/O, called *resonant I/O*.

4.2 The Design of Resonant I/O

The design objective of resonant I/O is to ensure that requests arrive at each data server in ascending order of file offsets for requested data from the same file. While data layout on disk usually matches offsets in the logical file space, the design allows the disk to service the requests in its preferred order, i.e., from small disk addresses to high addresses (possibly sequential), to achieve high disk throughput.

4.2.1 Making Collective I/O Aware of Data Layout

To induce resonance the compute node must know on which data server requested data is stored. Because an important design goal for the compute-node-side middleware is keeping the middleware independent of the data server side's configuration to ensure portability and system flexibility, explicitly requesting this information from the data servers is undesirable.

Fortunately, the configuration information that is needed in resonant I/O is readily available on the compute node side in many commonly used parallel file systems, including PVFS2 [17, 71], Lustre [2, 10], and GPFS [92]. In these systems meta-data service is separate from data service to avoid bottlenecks in data transfer. As such, a compute node needs to first communicate with the meta-data server to acquire the locations of its requested data on the data servers before it can access data on data servers. In fact, we only need to know the striping unit size and number of data servers, from which we can determine which requested data is on the same data server. We are aware that these two parameters may be set by users when the file is created in some file systems such as Lustre. However, to keep the design general and the interfaces of collective I/O unchanged, we do not assume that users would provide these parameters when they call collective I/O functions.

4.2.2 Process Access Set and data server Access Set

Because resonant I/O is an implementation of collective I/O, it does not make any changes to the function interfaces seen by programmers. As usual, each participant in a resonant-I/O operation needs to call the same collective-I/O function to specify one file segment or multiple non-adjacent file segments in a request. To execute the function call the processes are first synchronized to exchange information on the requested file segments so that every process knows all the file data requested in the collective I/O. After that, a collective-I/O implementation strategy needs to decide, for each process, which data the process is responsible for accessing. We call the set of data that is assigned to a process its *access set*. Once a process knows its access set it generates one or multiple requests to the data servers to access the data specified by the access set. In ROMIO collective I/O all file data to be requested are evenly partitioned into contiguous file domains. Each file domain is the access set of a process, which then uses only one request to access the data. Because this method of forming access sets based on contiguity in the logical file seeks to reduce the number of requests as well as their processing overhead, the resulting pattern of requests does not necessarily help improve disk efficiency, as described in Section 4.1.

To achieve disk efficiency in the implementation of collective I/O, we define a data server's access set as the set of data that is requested in a collective I/O *and* is stored on the data server. One of the objectives of resonant I/O is to ensure that a data server's access set is accessed by requests arriving in the ascending order of the offsets of the data in the logical file domain. Note that it is the LBNs (logical block numbers)* of the data that represent the on-disk locations of the data and directly determine the disk efficiency, and there is a mapping from the logical file offsets to on-disk LBNs by file systems. Therefore, in theory, ascending file offsets do not necessarily correspond to ascending LBNs, but in practice the correspondence generally holds, especially for file systems managing large files. Furthermore, our objective is that client-side optimization, such as resonant I/O, not require detailed configuration information on the server side. Using file offsets for this purpose fulfills this objective. Because the striping unit size and the number of data servers are available, processes on the compute nodes can easily calculate the access set of each data server.

The reasons that a data server's access set might be requested in a random order are that (1) data in the data server's access set belongs to multiple processes' access sets; *and*, (2) these processes send their requests in random order because of their unpredictable relative progress. To produce an ascending access order at a data server, resonant I/O can take either of two actions: (1) make *one* process' access set be a data server's access set; or, (2) make multiple processes send their requests in a pre-defined order. In the following we describe how resonant I/O takes the first action as its basic approach to produce an ascending access order, and takes the second action to make an optimization for a particular request pattern.

*If the data server is attached to a disk array the LBN refers to the address in the array.

4.2.3 Designating Agent Processes According to the Data Server's Access Set

If a process' access set is the same as a data server's access set, and the process sends its requests to the data server in ascending order of offset, then the data server will receive all of its requests in the preferred order. We call such a process the data server's *agent process*. Assuming each data server needs one agent process, for a given data server we select the process that requests the largest amount of data from the data server and has not been selected as another data server's agent process. If more than one such process exists, we arbitrarily choose the one with the lowest rank in the MPI process group as the agent process. As some data requested by an agent process may belong to other processes and need to be transferred between the agent process and their owner processes, this strategy minimizes the data to be transferred. The data transfer takes place before access to the data servers in the write operation, and after access to the data servers in the read operation. This data transfer is similar to the inter-process communication phase in ROMIO collective I/O. However, we make a special optimization for the read operation in this phase to minimize the transfer cost, as follows.

Synchronization is usually required after each agent has read data from data servers into its buffer and before the inter-process data transfer starts. This synchronization can degrade I/O performance by forcing all processes to wait for the slowest process to read its data; moving the synchronization ahead of the read operation would obviate this. To this end, we let all agent processes send their requests for their access sets in a non-blocking fashion in the first phase of the read operation, assuming non-blocking I/O is supported, and synchronize their progress immediately after sending requests instead of after the data has been read. Then each process proceeds to read *directly* from the data servers the data that it needs but has not requested in the first phase. If the process is not an agent, the data is actually all that it needs to access. This step replaces inter-process data transfer to eliminate synchronization immediately before the second phase. In this arrangement, we actually make many requests issued in the first phase serve as prefetching hints for the requests issued in the second phase. By performing the synchronization we ensure that requests in the second phase arrive after the data servers receive requests from the agents in the first phase. Thus the request service order at a data server is determined by the arrival order of requests in the first phase. When data is read from the disk, the requests of the second phase would be satisfied in the buffer cache of the data server. Usually the buffer cache is large enough to hold the data when the requests in the second phase arrive. By using the prefetching-like method, the two phases in resonant I/O can be overlapped to achieve higher efficiency.

Because an agent process may send many requests to a data server in resonant I/O, compared with one request in the ROMIO collective I/O, the request processing cost can be substantially higher. To reduce this cost resonant I/O uses list I/O to pack small requests for non-contiguous data segments into one or a few large requests to minimize request processing overhead. For the ROMIO implementation in MPICH2, one list I/O can accommodate up to MAX_ARRAY_SIZE (64) non-contiguous data segments, which can significantly reduce the cost.

4.2.4 Timing Requests from Different Processes

Because the second phase in the conventional implementation of collective I/O is the additional cost that does not exist in the non-collective I/O scheme, we seek to eliminate it subject to the condition that the access pattern satisfies a *non-overlapping* *condition.* This condition requires that in a collective I/O call the file offsets of the data requested by process i are smaller than those of data requested by process $i+1$ ($i = 0, 1, \cdots, N-2$; N is the number of processes). If a collective I/O call satisfies the condition for all the requests in the call to a given data server, those from process i will be for data with offsets smaller than those from process j ($i < j$). If we place the processes into sets according to the data servers to which they send their requests such that processes in different sets do not share data servers, and ensure that for all processes in a set, a process with lower rank always sends its request earlier than a process of higher rank, then the data servers will receive the requests in the preferred order. For this particular request pattern, by timing the sending of requests in different processes, we can produce the same effect on request arrival order as by using process agents. Then we can eliminate the second phase in which data is transferred to their owner processes, because each process requests its own data.

When the non-overlapping condition is satisfied, in each process set the process with lowest rank sends its request(s) first, and after a short delay it sends a synchronous message to the process with the next higher rank in the set, which then repeats the procedure. The delay is introduced to ensure that requests arrive at data servers in the preferred order. Our study has shown that because disk access time is usually much higher than message passing time, this delay can be chosen from a relatively large range, such as from 0.1ms to 1ms, with little effect on I/O performance, especially in a system supporting non-blocking I/O where a process can send its message without waiting to receive its requested data. (We note that the choice of delay does not affect the *correctness* of the protocol, only performance.) If non-blocking I/O is not supported no delay would be imposed and I/O access among processes would be fully serialized.

Because we coordinate request sending among processes, the benefits of improved disk efficiency will outweigh the penalty of reduced concurrency of I/O operations if the number of processes is comparable to the number of data servers. Otherwise, the serialization could become a performance bottleneck. To maintain balance, we set up n process groups in each process set sharing a common set of data servers, where n is the number of the data servers. We place the ith process in a set, sorted by rank, into group k, where $k = i/n$. Then processes in the same group send their requests without coordination, and the timing (or serialization) is carried out between process groups.

This timing technique can also be applied to make the approach using agent processes more scalable. When the number of processes in a collective I/O is much larger than the number of data servers, and the amount of data to be requested is very large, resonant I/O can designate more than one process agent for each data server for higher network bandwidth. This is made possible by timing the request sending in these multiple agent processes.

4.2.5 Putting it All Together

Figure 4.4 summarizes the design of resonant I/O. The objective in the design is to make requests served at each data server arrive in the preferred order. This is achieved by either allowing requests to one data server to be from the same agent process or by coordinating the issuance of requests from multiple processes. In achieving this objective, several optimizations were applied, including minimization of the cost of synchronization and elimination of the second phase of a conventional implementation of collective I/O.

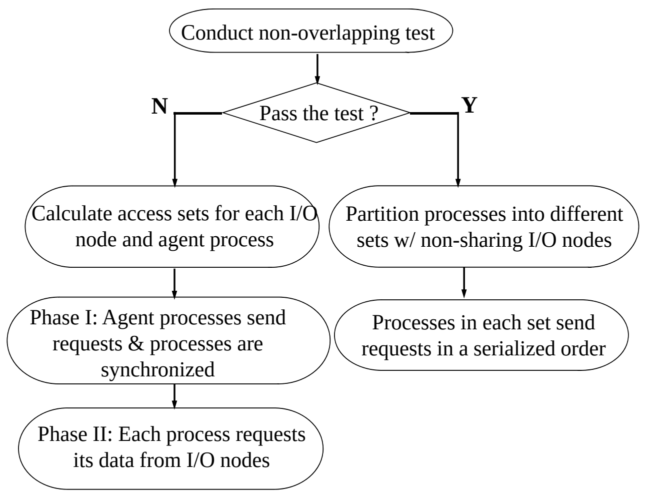

Conduct non-overlapping test
Pass the test ?
N
Y
Calculate access sets for each I/O node and agent process
Partition processes into different sets w/ non-sharing I/O nodes
Phase I: Agent processes send requests & processes are synchronized
Processes in each set send requests in a serialized order
Phase II: Each process requests its data from I/O nodes

Figure 4.4: Algorithmic Description of Resonant I/O

4.3 Effectiveness of Resonant I/O

To evaluate the performance of resonant I/O and compare it with the widely used collective I/O implementation in ROMIO, we used two different experimental platforms. The first is our own dedicated system, an eight-node cluster. All nodes are of identical configuration, each with dual 1.6GHz Pentium processors, 1GB memory, and an 80GB SATA hard disk. The cluster uses the PVFS2 parallel virtual file system (version 2.6.3), in which four nodes were configured as compute nodes and the other four as data servers. Each node runs Linux 2.6.21 and uses GNU libc 2.6. One of the data servers is also configured as the meta-data server of the file system. We used MPICH2-1.0.6 with ROMIO for our MPI programs. All nodes are connected through a switched Gigabit Ethernet network. The default striping unit size, 64KB, is used to stripe file data over the data servers. The second platform, used to evaluate how the performance of resonant I/O scales in a shared production environment, is described in the section on scaling.

Our resonant I/O is implemented in ADIO on top of PVFS2. The current version of ADIO does not provide genuine support for non-blocking I/O functions [72]. Because of this limitation our implementation of resonant I/O makes some compromises: (1) for the read operation, the second phase is not initiated until the data requested in the first phase has been received by the agent processes, which nullifies much of the benefit of using prefetching-like data access in the second phase; and, (2) the I/O operations among process groups are serialized. The consequence of these compromises is that experimental results for resonant I/O presented here are conservative, and potential performance advantages may not be fully revealed.

In addition to the demonstration program we used in Section 4.1 to exhibit the resonance scenario, we used five well-known benchmark programs for the evaluation: *coll_perf* from the MPICH2 software package [12], *mpi-io-test* from the PVFS2 software package [17], *ior-mpi-io* from the ASCI Purple benchmark suite developed at Lawrence Livermore National Laboratory [4], *noncontig* from the Parallel I/O Benchmarking Consortium at Argonne National Laboratory to test I/O characteristics with noncontiguous file access [15, 16], and *hpio*, designed by Northwestern University and Sandia National Laboratories, to systematically evaluate performance with a diverse set of I/O access patterns [49, 3].

All presented measurements represent arithmetic means of three runs. The variation coefficients—the ratio of the standard deviation to the mean—are less than 5% for the experiments on the dedicated cluster and less than 20% on the production system. To ensure that all data was accessed from the disk, we flushed the system buffer caches of the compute nodes and data servers before each test run.

4.3.1 Revisiting the Demonstration Program

We first revisit the demonstration program presented in Section 4.1. Figure 4.5 shows the I/O throughput observed when running the program with ROMIO collective I/O and resonant I/O with two and four MPI processes. The figure shows that resonant I/O can significantly improve I/O performance. It produces its peak throughput for segment size of 64KB with four processes and for segment size of 128KB with two processes, the two scenarios where resonance takes place when I/O requests are not collectively issued (c.f. Figure 4.2). In these two scenarios, resonant I/O increases I/O throughput by 151% and 75% over their counterparts in ROMIO collective I/O, respectively. However, the throughput of resonant I/O in these two scenarios is less than those of non-collective I/O shown in Figure 4.2. This is because resonant I/O needs synchronization in each call, which slows the faster processes. In fact a collective call is not necessary when a data server is dedicated to a process. For a segment size of 32KB and with two processes, ROMIO collective I/O coincidentally requests data in the same pattern as resonant I/O, so it has almost the same throughput as that of resonant I/O.

4.3.2 Results on the Dedicated Cluster

We ran benchmarks *coll_perf*, *mpi-io-test*, *ior-mpi-io*, *noncontig*, and *HPIO* on the dedicated cluster to measure their achieved aggregate I/O throughput when resonant I/O, and ROMIO collective I/O, were used. Because the I/O operation in all but *coll_perf* can be set as either file *read* or file *write*, and *coll_perf* can be divided into read and write phases, we measured the read and write throughputs separately.

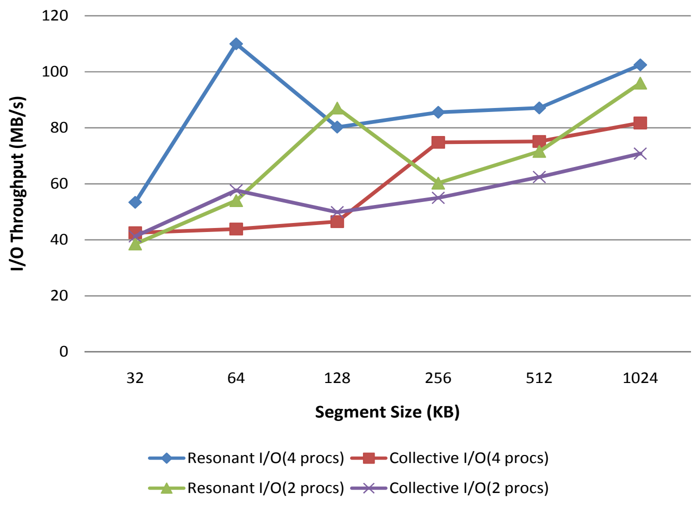


Figure 4.5: I/O throughput of the demonstration program with varying segment sizes and number of processes.

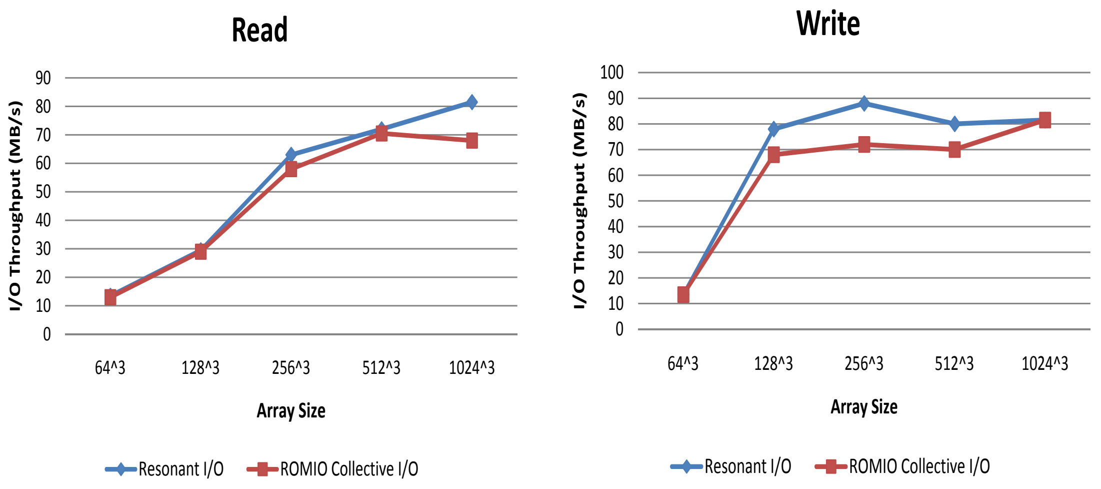


Figure 4.6: I/O throughput of benchmark *coll_perf* with varying scale of arrays.

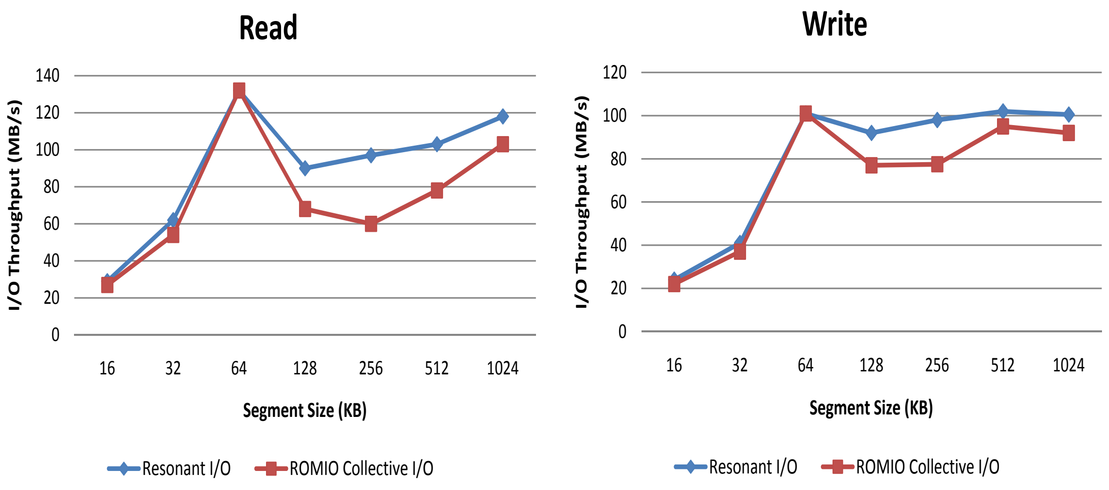


Figure 4.7: I/O throughput of benchmark *mpi-io-test* with varying segment sizes.

Benchmark *coll_perf*

The benchmark *coll_perf* comes from the MPI source package. Using collective I/O, this benchmark first writes a 3D block-distributed array to a file which resides on the parallel file system corresponding to the global array in row-major order and then reads it back, and checks if the data is consistent with the written data [12]. We scaled the array size from 64^3 to 1024^3 elements to test the effect of storage throughput. We isolated read and write phases with memory flushing instead of read-after-write used in the original implementation. Figure 4.6 shows the read and write throughput for both resonant I/O and ROMIO collective I/O. Because the I/O request size is proportional to the array size, as the array size increases the disk becomes very efficient in servicing individual requests, and the benchmark quickly achieves its peak throughput in the system (around 80MB/s). Therefore, while resonant I/O produces higher throughput, the improvements over ROMIO collective I/O are modest.

Benchmark ***mpi-io-test***

In the *mpi-io-test* benchmark we used four MPI processes, one on each compute node, to read a 10GB file. Each process reads one segment of contiguous data at a time. In each collective call, four processes read four segments in a row, respectively. In the next call, the next four segments are read. Figure 4.7 shows the throughput of the benchmark when resonant I/O and ROMIO collective I/O are used. As expected for this benchmark we see an I/O resonance (a spike in I/O throughput) at segment size 64KB. This resonance occurs with resonant I/O for both the read and write versions of the benchmark. Interestingly, we found that the ROMIO collective I/O also exhibits these resonances. Because there is no overlapping of processes' access ranges, ROMIO collective I/O does not re-arrange requests, and executes its I/O as non-collective I/O does. However, for other segment sizes, ROMIO collective I/O allows each data server to receive requests from multiple processes, and resonant I/O is able to order request arrivals and substantially increases the throughput by up to 61%. The figures also show that the write throughput is higher than read throughput when the segment size is larger than 64KB; this is mainly due to delayed write-back.

Benchmark ***ior-mpi-io***

In benchmark *ior-mpi-io* each of the four MPI processes reads one quarter of a 1GB file: process 0 reads the first quarter, process 1 reads the second quarter, and so on. The reads are executed as a sequence of collective calls. In a call, each of the four processes reads a segment with the same relative offset in their respective access scope, starting at offset 0. Figure 4.8 shows the throughput with different segment sizes. When the segment size is less than 64KB only one data server is involved in servicing requests in each call, so the throughput is low. The difference

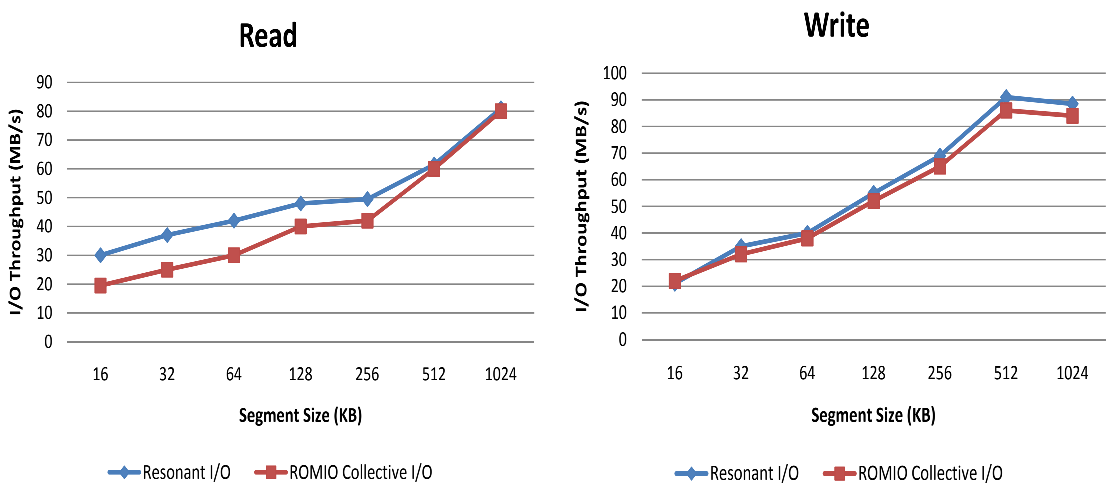


Figure 4.8: I/O throughput of benchmark *ior-mpi-io* with varying segment sizes.

is that requests are from one agent process in resonant I/O and from four processes in ROMIO collective I/O, which explains their performance difference in the read version of the benchmark. The performance advantage of resonant I/O diminishes with increasing segment size because increasingly amortized disk seek time reduces the penalty of non-sequential disk access in collective I/O.

Benchmark *noncontig*

Benchmark *noncontig* uses four MPI processes to read a 10GB file using the *vector* derived MPI datatype. If the file is considered to be a two-dimensional array, there are four 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*sizeof(MPI_INT)*. In each collective call, the total amount of data read by the processes is fixed, determined by the buffer size, which is 16MB in our experiment. Thus the larger *elmtcount* the more small pieces of non-contiguous data are accessed by each process.

When *elmtcount* is small, such as 4096, resonant I/O would need to send requests

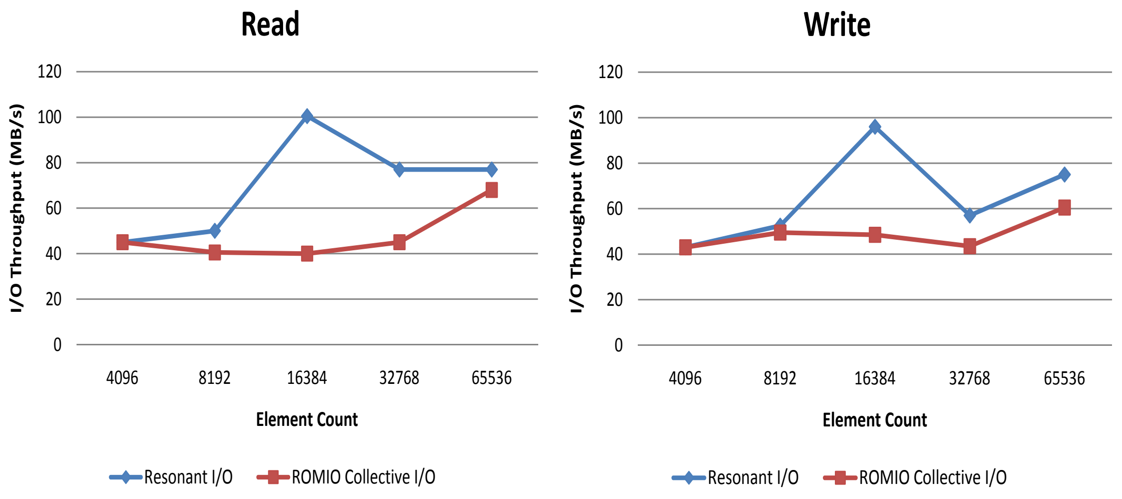


Figure 4.9: I/O throughput of benchmark *noncontig* with varying segment sizes.

for a large number of non-contiguous data segments. Because each list I/O can contain at most 64 non-contiguous segments using the default list I/O parameter, multiple list-I/O requests must be made by each agent process. This creates extra overhead for resonant I/O as ROMIO collective I/O uses only four requests. Figure 4.9 shows that the I/O throughput of resonant I/O is a little lower than that of ROMIO collective I/O when *elmtcount* is 4096. However, when *elmtcount* is increased, resonant I/O yields higher throughput. Both read and write throughput peaks at *elmtcount* of 16K when the segment size equals the striping unit size and all the data requested by an agent process is for itself. For read the peak throughput is 101MB/s, an improvement of 157% over that of ROMIO collective I/O, and for write the peak throughput is 96MB/s, an improvement of 97% over that of ROMIO collective I/O.

Benchmark ***HPIO***

The benchmark *HPIO* can generate various data access patterns by changing three parameters: *region_count*, *region_spacing*, and *region_size* [49]. In our experiment, we set *region_count* to 4096, *region_spacing* to 0, and vary *region_size* from 2KB to

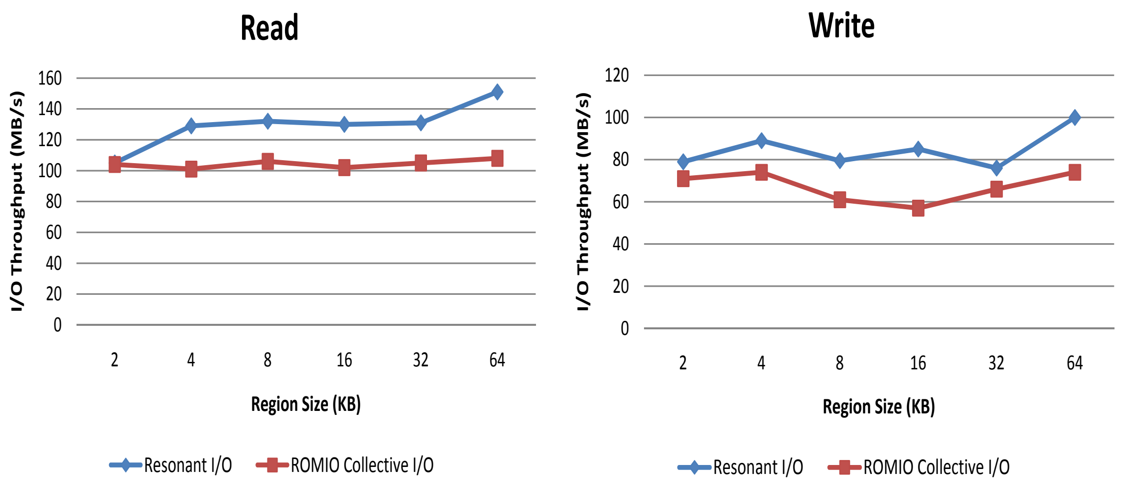


Figure 4.10: I/O throughput of benchmark *HPIO* with varying segment sizes.

64KB. Using four MPI processes, the access pattern is similar to the one described for benchmark *noncontig*. Here the length of a column is fixed as *region_count* (4096) and the width of a column varies from 2KB to 64KB (powers of two times 2KB). Each process reads its designated column with a collective call. Only one collective call is made in the benchmark.

Compared with the 16MB data requested in one collective call in *noncontig*, *HPIO* accesses much more data in one collective call, from 32MB to 1GB. This helps the benchmark to achieve a higher throughput and the high throughput is consistent across different region sizes, as we compare Figure 4.9 and Figure 4.10. Resonant I/O provides even higher throughput by rearranging requests to a data server, and produces a resonance peak at a region size of 64KB.

4.3.3 Resonant I/O Under Interference

In this section we study the impact of interference due to external competing I/O requests on the performance of resonant I/O. For comparison we also show the impact of interference on ROMIO collective I/O. We run two programs, the demonstration

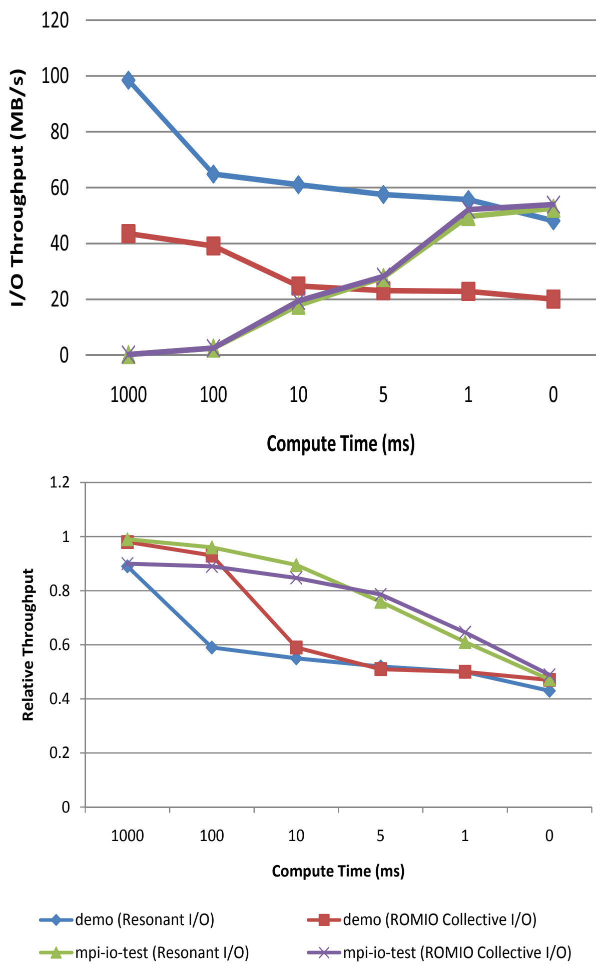


Figure 4.11: Absolute and relative throughput of resonant I/O and ROMIO collective I/O under different interference intensity, represented by length of the compute time between two consecutive I/O calls in *mpi-io-test*.

program, denoted by *demo*, and *mpi-io-test*, which concurrently access their respective files that are striped over the same set of data servers. We use four parallel processes for each program with 64KB segment size. In this experiment we consider *mpi-io-test* to be the source of interference with *demo*. To control intensity of interference we insert a period of compute time between two consecutive I/O requests in *mpi-io-test*. Thus the interference intensity is quantitatively represented by the magnitude of the compute time—the smaller the compute time the higher the interference. We also define a metric called *relative throughput* as the ratio of the throughput of the program under interference and the throughput of the program with exclusive access to the same storage system. We measure both absolute throughput and relative throughput of *demo* and *mpi-io-test* with inter-call compute time ranging from 1 second to 0 seconds using resonant I/O and ROMIO collective I/O, respectively (see Figure 4.11).

For the *demo* program, the relative throughput of resonant I/O drops from 0.90 to 0.43 as the compute time decreases from 1 second to 0. In contrast, the relative throughput of ROMIO collective I/O drops from 0.98 to 0.47. The relative throughput of resonant I/O drops at a greater relative rate, which demonstrates that resonant I/O is more sensitive to interference because sequential request-service order is more difficult to retain with increasingly high interference from concurrently I/O requests. However, even when there is no compute time between two consecutive I/O calls (and so the highest interference intensity) in *mpi-io-test*, resonant I/O still achieves an *absolute* throughput of 48MB/s for *demo*, which is more than twice the throughput of ROMIO collective I/O (22MB/s). Meanwhile, when the interference intensity is the highest, *mpi-io-test* could potentially reduce the throughput of *demo* by at least half. From this perspective, the relative throughput of resonant I/O for *demo*, which is 0.43, can be deemed quite acceptable. This result shows that the effort at the

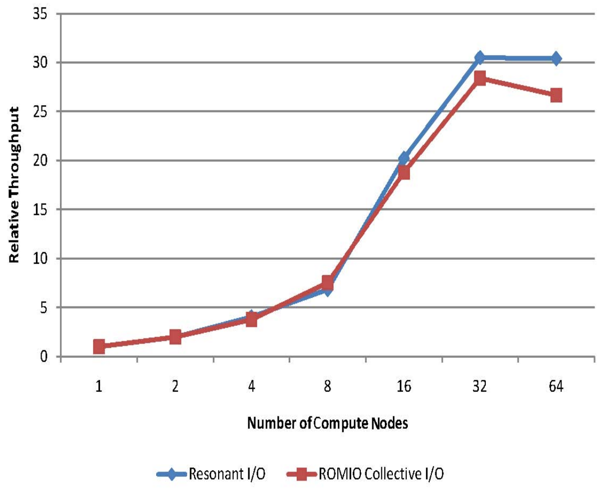


Figure 4.12: I/O throughput as a function of the number of compute nodes, relative to a single node, for benchmark *mpi-io-test*.

application/run-time level to maintain preferred request arrival order still helps to improve disk scheduling efficiency even when the competing load on the disk system is high and there are many pending requests for the disk scheduler to reorder.

For *mpi-io-test*, the relative throughput also drops but at a relatively moderate rate with the increase in interference intensity. Higher interference intensity means more I/O time in the program's run time, and the I/O time could be at least doubled when *mpi-io-test* runs concurrently with *demo* in comparison to when it has exclusive use of the I/O subsystem. Here the relative throughput of resonant I/O is slightly higher than that of ROMIO collective I/O. The rising curves representing absolute throughput of *mpi-io-test* are due to its increasing I/O demand as its compute time is reduced.

4.3.4 Scalability of Resonant I/O

In this section we study the scalability of resonant I/O in a production system environment, the Itanium 2 Cluster at Ohio Supercomputer Center, which has 110 compute nodes and 16 storage nodes, each with 4 GB of memory, running the PVFS2 file system. We ran benchmark *mpi-io-test* with 10GB file size and 1MB segment size with different numbers of processes, each on a different processor, to a maximum of 64. Figure 4.12 shows I/O throughput as a function of the number of compute nodes, relative to the throughput achieved on a single node, for benchmark *mpi-io-test*, for both resonant I/O and ROMIO collective I/O. As shown, resonant I/O is as scalable as ROMIO collective I/O. Because the quantity of data requested in a collective-I/O call is proportional to the number of processes, the I/O throughput increases with the number of processes to the limit of the storage system at 32 processes. When the performance of the storage system becomes a bottleneck, efficient use of the disk-based system becomes critical, which explains the performance advantage of resonant I/O over the ROMIO collective I/O when the number of processes is larger than 32. In general, both resonant I/O and ROMIO collective I/O scale well in the experiment. In addition, we note that the program shared the data servers with other concurrently running programs during its execution. As the measurements show, the concurrent I/O requests from other programs do not negate the effects of resonant I/O arranging a preferred access order for a higher I/O throughput. This is because the requests belonging to a collective I/O, implemented as resonant I/O, still arrive at the I/O system in a bursty fashion and so retain their preferred order.

Chapter 5

Inter-Server Coordination in a Storage Cluster

5.1 Lost of Spatial Locality

To provide adequate I/O support parallel file systems such as PVFS2 [17, 71], Lustre [2], and GPFS [92] 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.

5.1.1 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 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) [64] and Completely Fair Queuing (CFQ) [7] 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 process in a non-work-conserving fashion is greater than the cost paid for idle waiting.

5.1.2 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, 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's 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.

5.1.3 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 [100] and data sieving [100], 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, the reuse distance by a program may be much shorter than the reuse distance by an individual process, so it may be profitable for disk head to wait for 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 individually to exploit the spatial locality of each, when sufficient spatial locality exists.

5.2 Design and Implementation of IOrchestrator

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.

5.2.1 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 [17]. 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 a *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 created and its member processes are spawned, IOrchestrator sends unique identifiers for the running program (*job* in MPI) and its processes to the *pf* daemon.* 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

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

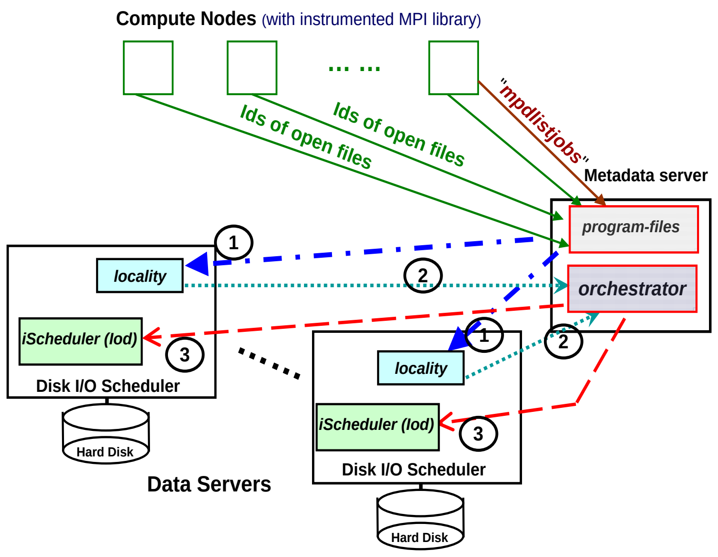


Figure 5.1: The IOrchestrator software architecture: the *pf* daemon collects the program-to-files mapping information from the compute nodes, and uses it to determine the program-to-local-file mapping information, which is passed to the *locality* daemons at the data servers (Step 1); the *locality* 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).

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, a component added to the PVFS2 *iod* daemon at each data server. The *iScheduler* daemon at each data server sits above the local disk scheduler, to which it relays requests. Figure 5.1 illustrates the architecture of IOrchestrator.

5.2.2 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 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. [61], 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 [1] for a similar purpose:

$$SL_i(k) = \frac{1}{8} * SL_i(k-1) + \frac{7}{8} * LBA\_Gap_i(k) \tag{5.1}$$

$$SL_{ij}(k) = \frac{1}{8} * SL_{ij}(k-1) + \frac{7}{8} * LBA\_Gap_{ij}(k) \tag{5.2}$$

$$RD_{ij}(k) = \frac{1}{8} * RD_{ij}(k-1) + \frac{7}{8} * ReuseDistance_{ij}(k) \tag{5.3}$$

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.

5.2.3 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}, 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} would usually imply a large reuse distance, which automatically leads to a small scheduling time slice.

5.3 Effectiveness of IOrchestrator

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.

5.3.1 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 [17]. 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 [4]. 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 [15] 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 [3]. 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, *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 [13]. 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*).

5.3.2 Performance of Homogenous Workloads

In this experiment we run two instances of each benchmark on the PVFS2 parallel file system and measure the aggregate 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 5.2 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

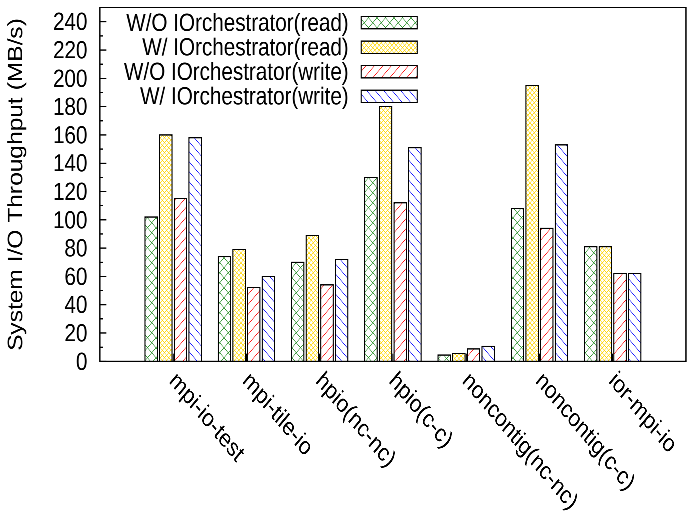
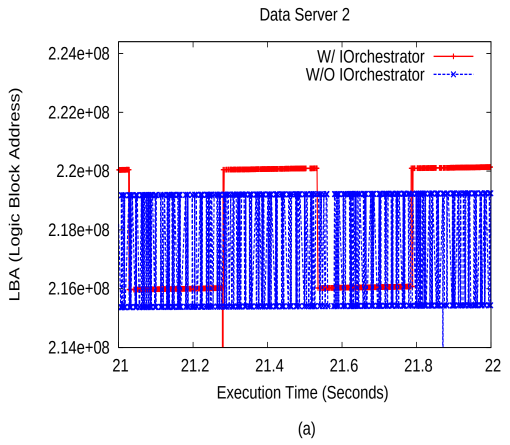


Figure 5.2: Aggregate I/O throughput with running of two instances of the same program when only *barrier* is used. For each program, the data access is set as either *read* or *write*. For *hpio* and *noncontig* data access pattern is set as either contiguous (c) or non-contiguous (nc) for memory access 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.

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. Figure 5.3(a) and Figure 5.3(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 1 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



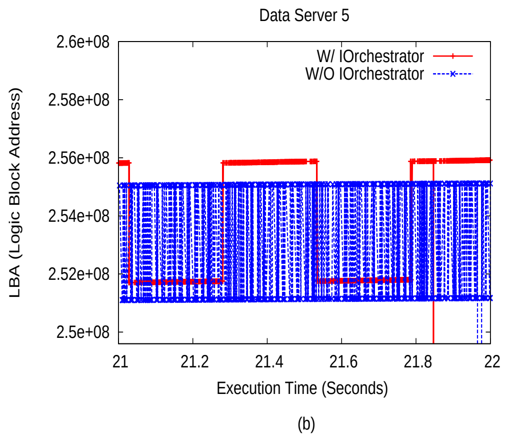
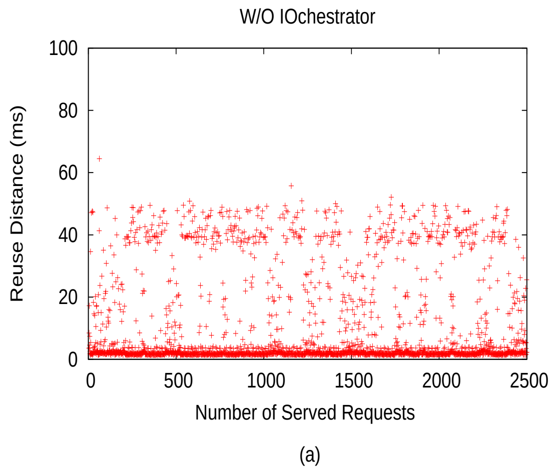


Figure 5.3: (a) and (b) show the disk addresses (LBAs) of data access on the disks of data servers 2 and 5 in a sampled 1-second execution period, when two *mpi-io-test* programs ran together with and without using IOrchestrator. Note that because the programs run much faster with IOrchestrator, they access disk positions somewhat different from those accessed by the programs without IOrchestrator during the same execution period.



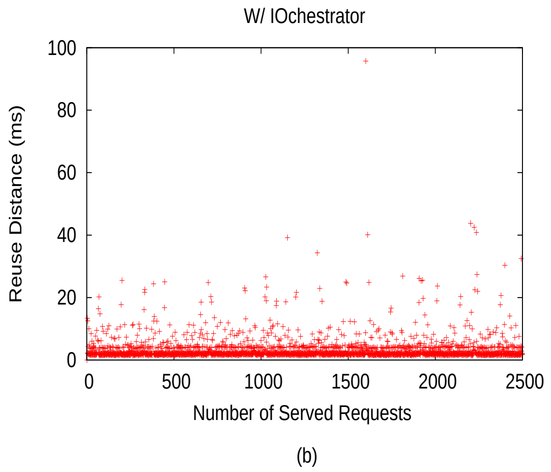


Figure 5.4: Reuse distances of requests served at data server 5 measured when two instances of *mpi-io-test* benchmark ran together without and with using IOrchestrator.

one running program at data server 5 during certain time period and show them in Figure 5.4(a) (without IOrchestrator) and Figure 5.4(b) (with IOrchestrator). Without IOrchestrator, there are many very large reuse distances (between 20ms and

50ms).[†] 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 5.3. 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 5.4(b)). This helps exploit the strong locality inherent in the program into efficient disk access (see the lines showing access with IOrchestrator in Figure 5.3). We can make similar observations for other benchmarks with sequential access patterns, such as *hpio(c-c)* and *noncontig(c-c)*.

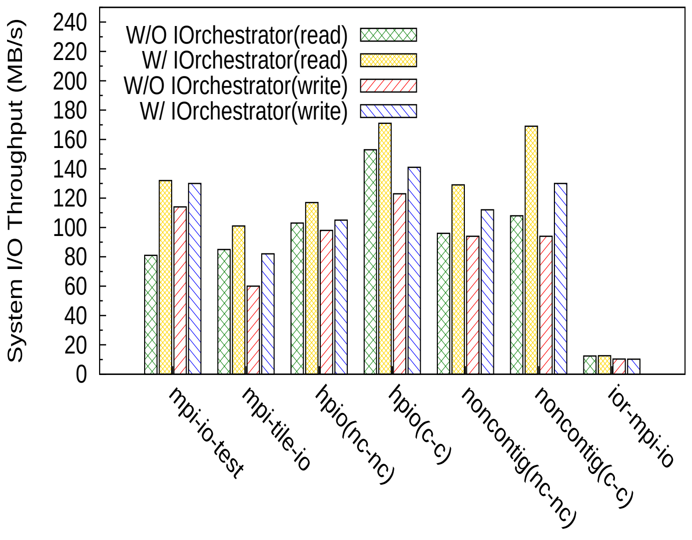


Figure 5.5: Aggregate I/O throughput running two instances of the same program when *barrier* and *collective I/O* are used. For each program, the data access is set as either *read* or *write*. For *hpio* and *noncontig* the data access pattern is configured as either contiguous (*c*) or non-contiguous (*nc*)

For *mpi-tile-io* I/O throughput increased by 11% for *read* and 15% for *write*. The

[†]Those very small reuse distances shown in Figure 5.4 are mostly produced by requests from different processes of the running program, and can be exploited by the CFQ scheduler.

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 5.5 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.

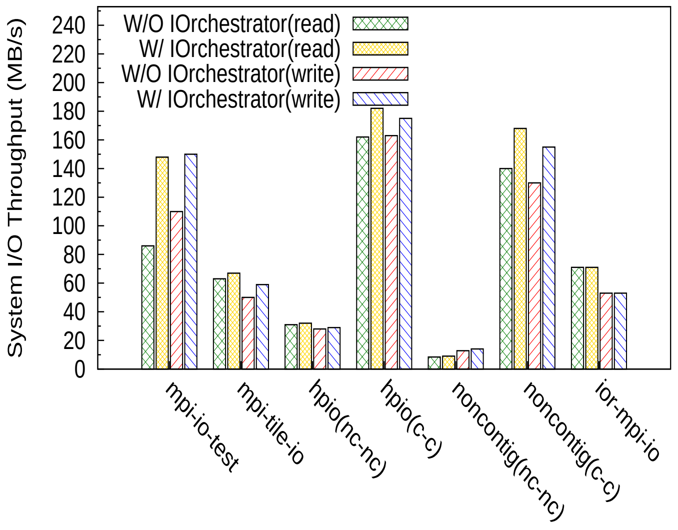


Figure 5.6: Aggregate I/O throughput with running of two instances of the same program when both *barrier* and collective I/O are not used. For each program, the data access is set as either *read* or *write*. For *hpio* and *noncontig*, data access pattern is configured as either contiguous (*c*) or non-contiguous (*nc*)

Performance without *barrier* or collective I/O. Figure 5.6 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* (Figure 5.2 and Figure 5.6). 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 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 5.6. 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.

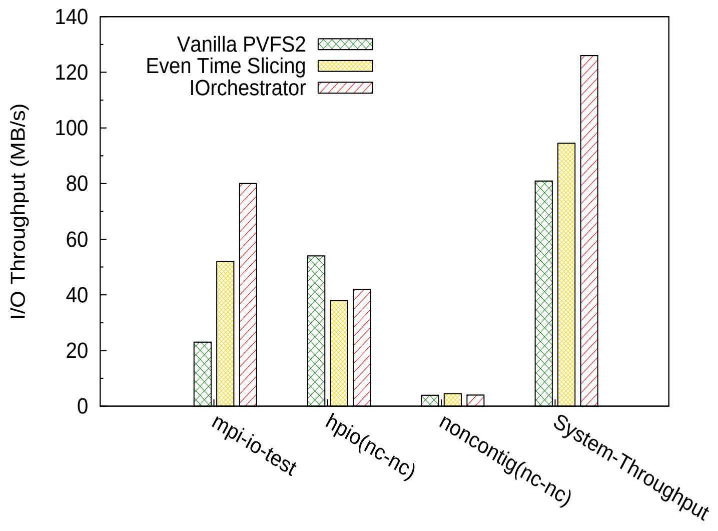


Figure 5.7: I/O throughputs of three different programs, *mpi-io-test*, *noncontig(nc-nc)*, and *hpio(nc-nc)*, when they are running together to read three data files of 10GB, respectively. The entire system's throughputs are also shown. Three systems are tested in the experiment: the vanilla PVFS2, PVFS2 with IOrchestrator with even time slicing, and PVFS2 with IOrchestrator.

5.3.3 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 5.7 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.

5.3.4 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.[‡] Figure 5.8 shows the

[‡]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.

system's I/O throughputs. The results are consistent with our hypothesis: when the distance 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.

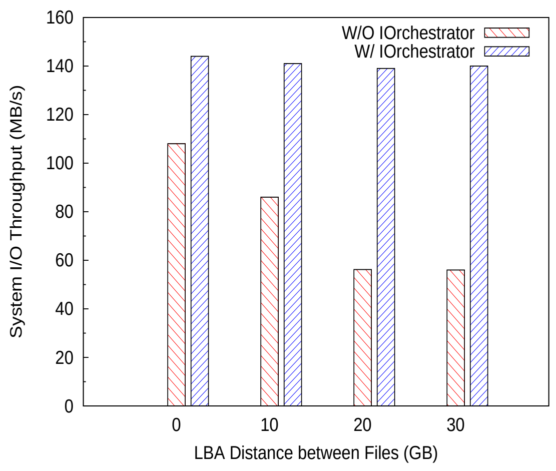


Figure 5.8: Aggregate I/O throughputs measured when we increase on-disk file distances from 0GB to 30GB.

5.3.5 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 5.9 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.

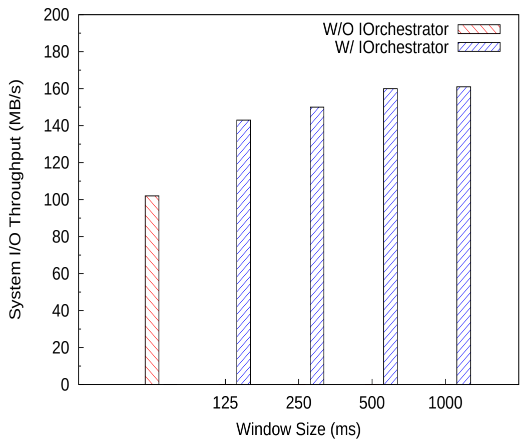


Figure 5.9: Aggregate I/O throughputs measured when we increase the scheduling window size from 125ms to 1000ms. The I/O throughput without using IOrchestrator is also shown for comparison.

Chapter 6

Using SSD to Improve Disk Scheduling for High-Performance I/O

6.1 Introduction

Data-intensive scientific computing applications are producing increasingly high I/O demands on the storage devices of high-performance computing systems. Request concurrency, or the number of processes concurrently issuing requests, can be very high at data servers serving requests from applications running on a large-scale cluster. Besides the potentially large volume of requested data, this concurrency can significantly compromise the efficiency of a hard-disk-based storage system: data on a disk that are requested by different processes or programs are usually separated on the disk, and concurrently accessing them can cause the disk head to frequently seek from track to track, potentially delivering I/O throughput an order of magnitude lower (or worse) than that for sequential disk access.

6.1.1 Limitations of Existing SSD Solutions

The emerging solid-state-drive (SSD) is largely unaffected by random access because it does not contain any moving parts—it is basically a uniform memory access device. However, it is currently not an economical option in high-performance computing (HPC) to use it as the main storage in a large-scale installation, the existence of Gordon, a SSD-based cluster made possible through $20 million funding from the US government (National Science Foundation), notwithstanding [41, 86]. More cost-effective and practical options are either to use an SSD as buffer cache between main (DRAM) memory and the hard disk and exploit workloads' locality for data caching [97, 89], or use it with the hard disk to form a hybrid storage device such that frequently accessed data is stored on the SSD [87, 46].

These schemes for SSD usage, however, do not effectively address the problem of concurrent requests. Unlike the consumer or enterprise workloads that have relatively small working data sets and exhibit relatively strong locality, the characteristics of workloads from data-intensive parallel programs are not accommodated well. First, the data accessed in a single run of a data-intensive parallel program can be larger than the capacity of the SSD. When a program processes a large data set, data are rarely accessed multiple times from the disk and the accesses therefore exhibit weak temporal locality, which is hard to exploit by a relatively small SSD for effective caching. Second, requests to a disk are usually interleaved from different processes of one or multiple programs. Most existing SSD-based schemes exploit spatial locality, i.e., attempt placement of randomly accessed data on the SSD such that the hard disk serves requests of sequential, or at least well-ordered, data. However, when the request concurrency is high, it is highly likely that most requests from different processes are presented to the disk as random access and have to be handled by the

SSD. This would overwhelm SSD as a cache, or as a small storage device for random data, and make these schemes ineffective.

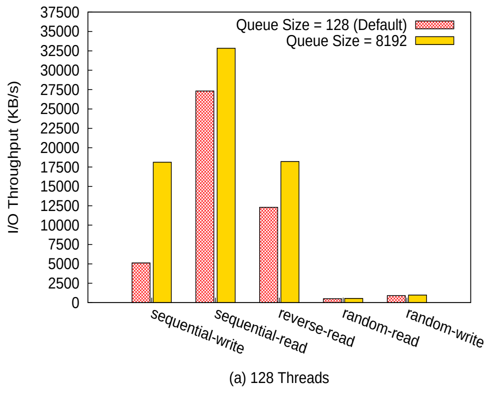
6.1.2 Limitations of Existing Software Strategies

Traditionally the problem of concurrent requests is addressed in middleware using techniques such as collective I/O and data sieving [100], or in the system using buffer cache and the I/O scheduler. The middleware approach is more concerned with reducing the number of requests than request concurrency. To use it I/O requests in a parallel program must be presented via specific interfaces such as MPI collective-I/O function calls or MPI derived data types. Moreover, the high request concurrency due to processes belonging to different programs cannot be reduced by this approach. The system buffer cache is usually even smaller than the SSD, and therefore shares its concerns when handling requests of high concurrency in a large-scale system. In the operating system, the I/O scheduler is the last opportunity to exploit spatial locality in the presence of high request concurrency. For example, CFQ, the default Linux disk scheduler, reduces random data accesses by merging and sorting outstanding requests according to their logical block addresses (LBAs) [7]. The outstanding requests are usually placed in a data structure called a dispatch queue. The larger the queue, the more requests can be collected for sorting and the greater the chance to exploit spatial locality. The default queue depth in Linux's CFQ is 128, i.e., the queue can hold at most 128 outstanding requests.

To investigate the effect of queue size on I/O performance in the presence of request concurrency, we ran IOzone [5], a commonly used filesystem benchmark generating and measuring a variety of file operations, with a varying queue size on a data server running Linux with CFQ to access data on a hard disk (details of the server's configuration are given in Section 6.3). The benchmark ran in its *throughput* mode, in which we can vary the number of threads to control request concurrency. Each thread generates POSIX *asyn* I/O requests with at most 32 outstanding requests. Each thread accesses its own file, and the total amount of accessed data is 8GB in the experiments. Figure 6.1 shows I/O throughputs reported by the benchmark for access patterns *Sequential Read/Write*, *Reverse Read*, and *Random Read/Write*, and queue sizes 128 and 8192, with either 128 threads (Figure 6.1(a)) or 256 threads (Figure 6.1(b)). *Reverse Read* sequentially reads a file from its end to its beginning.

As demonstrated, increasing queue size can significantly improve performance for *Sequential Read/Write* and *Reverse Read.* In the original configuration even for the case where each thread issues sequential requests (*Sequential Read/Write*), the I/O throughputs are consistently low. In particular, the throughputs with 256 threads, 2.3MB/s for *Sequential Read* and 1.8MB/s for *Sequential Write*, are substantially lower than those with 128 threads, 5.1MB/s for *Sequential Read* and 26.2MB/s for *Sequential Write.* This indicates that it is concurrently serving multiple requests from different threads that weakens spatial locality and hurts performance. When the queue size is increased to 8192 the throughputs are significantly increased (by 42% to 650%), and the improvements are especially dramatic in the case of 256 threads. This demonstrates that a large queue can effectively recover spatial locality if it exists in requests from the same thread. However, when individual threads issue fully random requests, the I/O throughputs are very low and the improvements made by the increased queue size are also small. This clearly demonstrates that random requests are at best difficult to schedule for efficient service by hard disk.

While increasing the size of the dispatch queue in memory can improve access locality for higher disk efficiency, the approach has limitations. First, having a large queue would allow many write requests to be outstanding in volatile DRAM. This



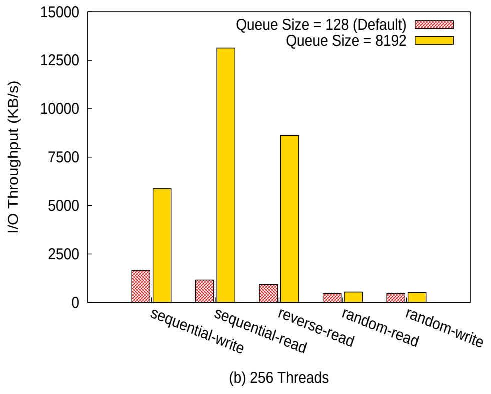


Figure 6.1: I/O throughput of a data server of the experimental cluster as the IOzone filesystem benchmark run with (a) 128 threads; and, (b) 256 threads. In each figure I/O throughputs with differing dispatch queue sizes and differing data access patterns are shown.

runs the risk of losing a large amount of data as frequent system failures in a large-scale system can be the norm [96]. Second, although a long queue usually improves throughput, it can allow requests to remain in the queue for an extended period of time without being completed, which may result in excessive response times for those requests; for applications with strict QoS requirements a long queue can be problematic. Third, as we showed in the experiments, simply increasing the queue size may not be sufficient, especially for addressing the issue of concurrency among streams of random requests.

6.1.3 Our Solution: Use SSD for Disk Scheduling

We propose to extend the scheduler's dispatch queue and place the extension on SSD. In our design the in-memory queue is only responsible for dispatching requests to disk if relatively strong locality can be identified in the queue. Otherwise the requests are sent to the queue extension on the SSD for further scheduling. As the SSD is less expensive than DRAM in terms of both capacity and energy consumption, the trade-off for greater capacity is justifiable. In addition, because SSD is non-volatile, dirty data in the queue extension need not be lost because of system failure. For the same reason, a write request can be considered complete once it is sent to the queue extension. As such a large queue extension does not cause excessive response times. Because SSD performance is not sensitive to random access, random write requests can be quickly serviced. For the data transfer between SSD and the disk, we schedule large-sized and well-sorted write-back requests and prefetch requests, and in the background of serving process request if possible. Leveraging the non-volatility and large size (relative to DRAM) of SSD, our design enables the decoupling of serving process requests and disk operation in request scheduling.

In summary, we make the following contributions.

- We propose a new disk scheduling architecture that uses SSD to facilitate exploitation of spatial locality in I/O requests and to hide random access latencies on the hard disk resulting from serving process requests.

- We design an algorithm for intelligently using the in-memory queue and an in-SSD queue extension, and for effectively scheduling background write-back and prefetch requests to minimize the negative effects of concurrent requests.

- We implement the scheduling architecture and the scheduling algorithm, collectively called iTransformer, as a stand-alone Linux kernel module. The implementation is transparent to the software above the generic block layer in the kernel memory hierarchy and is therefore portable across different parallel file systems.

- We evaluate iTransformer with representative benchmarks, including *ior-mpi-*

io, *Hpio*, *BTIO*, and *S3aSim* on a large cluster. Experimental measurements show that it significantly reduces random access on the hard disk and increases I/O throughput of storage system by up to 3X, and 35% on average, compared to the stock system, for these benchmarks.

The rest of this paper is organized as follows. Section 6.2 describes the design of iTransformer. Section 6.3 describes and analyzes experiment results.

6.2 The Design of iTransformer

iTransformer is designed to exploit characteristics of SSD, including non-volatility and large size (compared to DRAM memory of similar cost or power consumption) and insensitivity to random access (compared to hard disk) to make the I/O scheduler function more effectively. As existing I/O schedulers such as CFQ and Deadline have received years of design, implementation, evaluation, and tuning, iTransformer is not intended to be a new scheduler for hard disk or SSD. Instead, it acts as a scheduling framework to direct requests into dispatch queues and relies on the existing scheduler to decide their actual service order on disk.

6.2.1 Scheduling Architecture

In iTransformer the role of the SSD is to enhance locality. Without changing the existing disk scheduler, iTransformer monitors the locality exploited by the scheduler over its regular in-memory request dispatch queue, and evaluates how much improvement it could make if the requests were further scheduled by the scheduler over the extended dispatch queue in the SSD. To justify the cost associated with SSD operations, only if the improvement is predicted to be sufficiently large are requests sent to the in-SSD queue extension for further processing. Otherwise the requests are directly issued to the disk as current systems do. Note that by the queue extension in the SSD, we refer to the data for reading or writing that are in the SSD. Their metadata, or data structures describing the requests, are resident in memory and are managed by the standard disk scheduler operating on the regular dispatch queue. The size of the queue, or the number of requests the queue can hold, is determined by the amount of data accessed by the requests, which is bounded by the SSD space allocated for iTransformer.

To determine the potential locality improvement the SSD queue extension could achieve even if the SSD scheduling is not in use, we maintain a separate *ghost* queue to hold the metadata of any requests dispatched out of the regular queue. The size of this ghost queue is the same as that of the SSD queue extension. We run the standard disk scheduler over the requests in the ghost queue and monitor the locality of requests released from the queue. The purpose of the ghost queue is solely to evaluate potential locality improvement, so the requests out of the queue are never actually dispatched.

6.2.2 Determining the Use of SSD

To quantify the locality of a stream of requests we use an approach similar to the one adopted in the Linux kernel for a similar purpose [82, 7]. The locality of the stream of requests $\{R_0, R_1, ..., R_n\}$ is defined by a function $L(n)$. The size of data requested by R_k is denoted by S_k, and the distance between two consecutive requests R_k and R_{k+1}, is denoted by D_k, $0 \leq k < n$, and is the absolute value of difference between logical block address (LBA) of R_k's last data block and LBA of R_{k+1}'s first data block. Initially, $L(0) = 1/S_0$. When $L(n)$ is obtained for request stream $\{R_0, R_1, ..., R_n\}$ and R_{n+1} arrives, $L(n+1)$ is defined by

$$L(n+1) = L(n)/8 + (7/8) * (D(n)/S_{n+1}). \tag{6.1}$$

The weights 1/8 and 7/8 for the last locality value and the new locality value, respectively, are used to produce a decay effect so that more recent requests are better represented. These two weight values are chosen according to the formula used in the Linux kernel for a similar purpose in its implementation of anticipatory scheduling [7]. Here a smaller locality value (L) indicates stronger locality. We continuously measure the locality of the requests dispatched out of the regular in-memory queue (L_{in_mem}). When SSD is not in use for scheduling, the ghost queue is receiving requests and we measure the locality for the requests out of the queue (L_{ghost}). The potential locality improvement is calculated as $H = L_{in\_mem}/L_{ghost}$.

When H is larger than a threshold, SSD scheduling is enabled and the requests dispatched from the in-memory queue enter the in-SSD queue for scheduling instead of going to disk. When the in-SSD queue is in use, iTransformer monitors its dispatched requests and calculates their locality (L_{in_ssd}) as well as $H = L_{in\_mem}/L_{in\_ssd}$. If H becomes smaller than a threshold, SSD scheduling is disabled. The two default thresholds are 4 and 2, respectively, in our implementation.

6.2.3 Dispatching Random Write Requests via Out-of-band Writeback

One of the advantages of incorporating SSD into request scheduling is decoupling the dispatching of write requests from processes' progress: we may delay the service of write requests as long as the SSD queue is not full and schedule them when the hard disk is not busy. In such a scenario even random write requests whose locality cannot be effectively exploited in the SSD queue, such as the random requests shown in Figure 6.1, can benefit from rerouting requests to the SSD. However, the H value calculated as before does not take this effect into account and so can be too pessimistic to trigger the use of the SSD. To take advantage of behind-the-scenes request service opportunities we modify the calculation of L_{in_ssd} and L_{ghost}. In the case that the SSD queue is in use, when a write request is dispatched to the disk and during its service time period no new requests arrive in the SSD queue, the distance gap D between this request and the request dispatched immediately before it is set to 0 in the updating of L_{in_ssd}. In the case that the SSD queue is not in use, we need to modify the calculation of L_{ghost}. However, the requests dispatched from the queue are not sent to the disk for actual service. Therefore, we cannot use the queue to estimate the disk idle period. Instead we conservatively use the in-memory queue for this purpose. When a write request is dispatched from the queue and the queue does not receive new requests in the request's service period, we treat the distance D between the request and the one before it in the scheduling of the ghost queue as 0 to calculate L_{ghost}. In both cases a less-occupied disk will help produce larger H and encourage the use of SSD for scheduling random write requests.

6.2.4 Servicing Read Requests via Data Prefetching

Compared with write requests, the service of read requests is harder to speed up if they are random, and even the SSD queue cannot effectively exploit their locality. This can be a serious concern because their servicing can be on the critical path of process execution, especially if they are synchronous requests. Furthermore, for synchronous requests at most one request can be outstanding for scheduling so the locality in individual process's requests is not visible to the scheduler. Even non-work-conserving schedulers such as anticipatory [64] and CFQ [7] cannot help because the schedulers at the servers do not know from which process a request is issued [106]. This can make in-SSD scheduling ineffective.

To address this problem we monitor the read requests to identify data worth prefetching and set up a prefetch area on the SSD for caching prefetched data. This monitoring is performed whether or not in-SSD scheduling is enabled. Read requests are checked against the SSD prefetch area and dispatched only for data that is not present. Prefetch candidates are only dispatched when the disk is idle or their disk locations are close to the location accessed by the most recently dispatched request. Prefetching thereby does not consume memory space, nor does it aggressively compete for disk service time with process requests. The only concern is to determine the data of maximum value to prefetch. To this end we fix a prefetch unit size and partition the disk address space into slots of unit size. The prefetching scheme identifies slots that read requests have moved into and out of multiple times. When such patterns repeat there would be high value in having prefetched these slots, thereby avoiding the long service times of random reads.

We have developed an efficient algorithm for this purpose. We maintain an LRU stack to hold the metadata of slots, including slot number, access counter, and a flag recording whether a slot is prefetched. When a read request is dispatched, we place the metadata of the slot that the request accesses at the top of the stack. If the slot was not already in the stack, its access counter is set to one; if the slot was already in the stack, but not at the top, it is removed from its previous position and its access counter incremented; if the slot was already at the top then the current request is not considered a random request, so its counter is not incremented. Thus the counter tracks number of notional random accesses to a slot. When a slot's counter value is greater than a threshold (default 2), the slot becomes a prefetchable slot. When the disk is idle iTransformer searches the stack from the top to find the first prefetchable but not-yet-prefetched slot and issues a read request to load it into the SSD prefetch area. When the prefetch area becomes full, the data in the bottom-most prefetched slot in the stack is removed. The size of the stack is twice the number of slots the prefetch area can hold. When a slot is not accessed for a long time it will be removed from the bottom of the stack and lose its history access information, as well as the prefetched data if it had been prefetched. For every prefetched slot whose data is removed, we calculate the percentage of its data that was prefetched but not yet actually requested. If average of the percentages for recently replaced slots is below a threshold (default 40%) the loading of data into prefetchable slots is suspended until the average is above a second threshold (default 60%). In this way the accuracy of prefetching is maintained.

6.3 Performance Evaluation

6.3.1 Implementation of iTransformer

We have prototyped iTransformer with Linux kernel 2.6.35.10 with instrumentation of the Linux device mapper, a part of the Linux storage infrastructure software stack. iTransformer is implemented as a stand-alone kernel module in the generic block layer for request monitoring and rerouting, and data prefetching. To activate iTransformer in a cluster system, one need only load the module into the Linux kernel on each of the data servers. To maintain data consistency iTransformer writes dirty data on the SSD back to disk on unloading of the module. During initialization iTransformer checks if there are any dirty data left in the SSD because of system failure, and rebuilds a mapping table for describing the contents of the SSD. Whenever requested data is found in the SSD via the mapping table, whether written dirty data or prefetched data, the requests are rerouted to the SSD so that up-to-date data is efficiently accessed. Because the SSD cannot directly write to or read from the hard disk, we use via-memory read and write to simulate data transfer between SSD and hard disk. These requests to the disk bypass the in-memory dispatch queue and are sent directly to the disk to avoid affecting the behavior of the standard scheduler.

6.3.2 Experimental Setting

We conducted an extensive experimental evaluation of iTransformer on the Darwin cluster at Los Alamos National Laboratory. Darwin consists of 120 compute nodes, a head node, and two admin nodes. Of the 120 nodes, 116 are 48-core (12-core by 4 socket) 2GHz AMD Opteron 6168, and are the nodes on which our experiments were performed. Each node has 64GB memory, a hardware-based RAID 0 consisting of two 500GB 7200rpm disk drives (HP model MM0500FAMYT), and a 120GB SSD drive (HP model MK0120EAVDT). The nodes are connected by both 1GB Ethernet and Infiniband networks. Each node runs Fedora Linux with kernel 2.6.35.10. CFQ [7] and NOOP [9] were used as the disk I/O scheduler for the HDD and SSD devices, respectively. NOOP simply dispatches a request as soon as it is received and does nothing beyond merging contiguous requests. Its performance is usually better than other schedulers for SSD devices [45]. We configured nine of 116 AMD nodes as data servers using PVFS2 parallel file system [17], one of which was also configured as the meta-data server. We used MPICH2-1.4 [12], compiled with ROMIO, to generate executables of MPI programs. The iTransformer kernel module was installed on every data server. To provide fair and reproducible throughput measurements we removed any cached data from system buffers at each data server before each test, and periodically (every second) flushed dirty pages in the system buffers to their respective disks. In the experiments, unless otherwise specified, the SSD allocation for holding data accessed by requests in the SSD queue extension was 8GB, the size of the SSD prefetch area was 8GB, and the prefetch unit size was 4MB.

Table 6.1 shows the basic throughput measurements of the HDD and SSD devices on a data server with fully sequential and fully random requests and with a uniform request size of 4KB. For random requests the SSD's throughput is much higher than that of the hard disk. For sequential requests the disk device provides slightly higher throughput than the SSD, as a 2-disk RAID 0 is used. We selected four benchmarks, chosen from different application fields and representing different access patterns, for the evaluation. Following we present and analyze the experiment results of running the benchmarks individually and concurrently. The throughputs of the system with iTransformer module are compared against those on the stock Linux system.

	SSD	Hard-disk RAID
Capacity	120GB	1TB
Interface	SATA	SAS
Sequential Read	160MB/s	170MB/s
Random Read	60MB/s	15MB/s
Sequential Write	140MB/s	160MB/s
Random Write	30MB/s	5MB/s

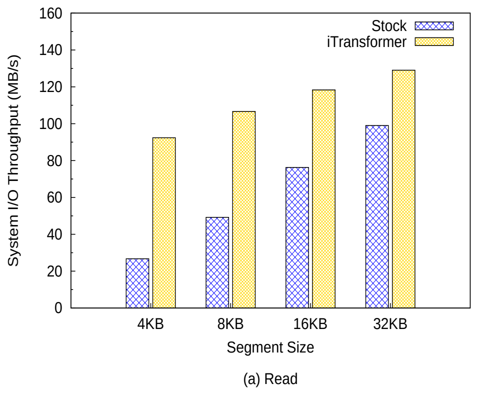
Table 6.1: Comparison of basic performance of the SSD and HDD devices used in the experiments, 4KB request size.

6.3.3 The *ior-mpi-io* Benchmark

Ior-mpi-io is a program in the ASCI Purple Benchmark Suite developed at Lawrence

Livermore National Laboratory [4]. In this benchmark each of n MPI processes is responsible for reading its own $1/n$ of a 20 GB file. Each process continuously issues requests of fixed segment size with random offsets. The program's access pattern as presented to the storage system is effectively random. We ran three concurrent instances of the program in the system, each with 64 processes and accessing its own file. The requests are concurrently sent to the data servers. Figure 6.2(a) shows the aggregate I/O throughput produced by the system with and without iTransformer with segment sizes ranging from 4KB to 32KB. The figure shows that iTransformer can dramatically increase I/O throughput up to 2.4X that of the stock system. With increasing segment size the improvement becomes smaller because the spatial locality within each request becomes stronger and disk access efficiency correspondingly improves. Changing each read request to the corresponding write request yields results shown in Figure 6.2(b). The improvements with iTransformer are not as great as those for reads, but are still substantial, from 47% to 56%. For reads the enabled prefetching allows data to be retrieved from the disk in large chunks (4MB) while for writes the SSD scheduling only produces better-sorted random write sequences.

To better characterize the reasons for the performance improvement we tracked the accessed addresses on the HDD and SSD using Blktrace [6] and show in Figure 6.3 the accesses at a particular data server during the one-second execution period starting from the 100th second of execution using a 4KB-segment read requests. The addresses are presented as logical block addresses (LBAs). Figure 6.3(a) shows that the accessed locations with the stock system are random over a large disk region and that the disk I/O scheduler, CFQ, does not effectively exploit spatial locality among them. Figure 6.3(b) and Figure 6.3(c) show the accessed locations on the HDD and SSD, respectively, using iTransformer. The hard disk mostly sees sequential or well-sorted accesses while random accesses mostly take place on the SSD. This is because random



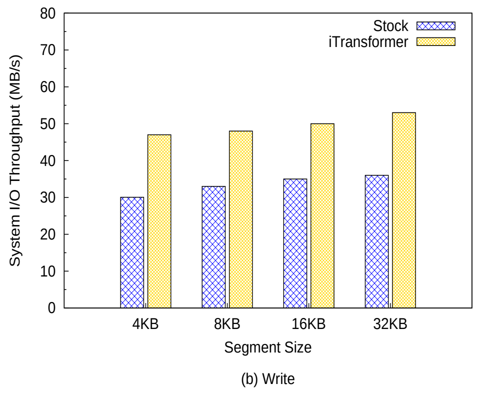
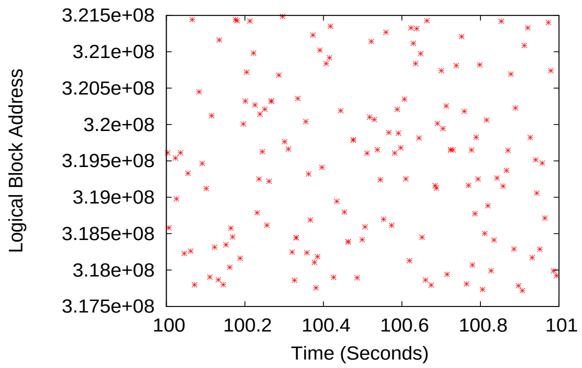


Figure 6.2: System I/O throughputs for the *ior-mpi-io* benchmark using read requests (a), and using write requests (b), for the stock system and system using iTransformer.

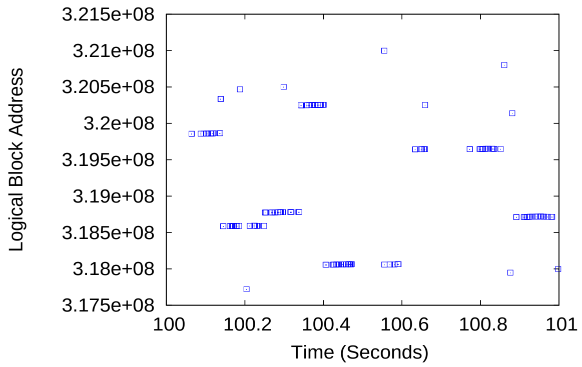
read requests triggered prefetching of 4MB data chunks into the SSD, resulting in many random read requests being hits on the SSD.

There are two important factors that affect iTransformer's effectiveness in handling read requests: prefetch area size and prefetch unit size. To study their effects on I/O performance we re-ran the *ior-mpi-io* experiment with 4KB segment size using different prefetch area and unit sizes. Figure 6.4 shows the system I/O throughputs with prefetch area size ranging from 0GB to 8GB. With a area size of 0GB the prefetching function is effectively disabled. As shown in the figure, even with a relatively small prefetch area of 1GB, the throughput can be improved by 117%. Increasing the size increases the throughput because it allows prefetched data to stay cached longer and so increases the likelihood of serving read requests from the SSD.

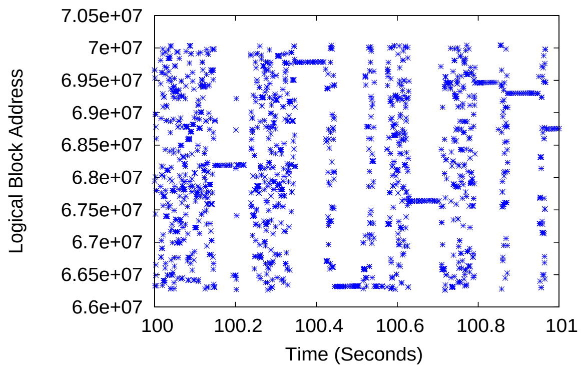
Table 6.2 shows the I/O throughputs with different prefetch unit sizes and their improvement ratios over the throughput in a system without prefetching enabled. When the unit size is 64KB, prefetch requests are too small and cannot be efficiently served. This shows that the performance benefit of prefetching can be outweighed by



(a)



(b)



(c)

Figure 6.3: Accessed locations when running *ior-mpi-io* using read requests in a sampled one-second execution period. (a) The locations are on the hard disk and the stock system is used. (b) The locations are on the hard disk and the system with iTransformer is used. (c) The locations are on the SSD and the system with iTransformer is used.

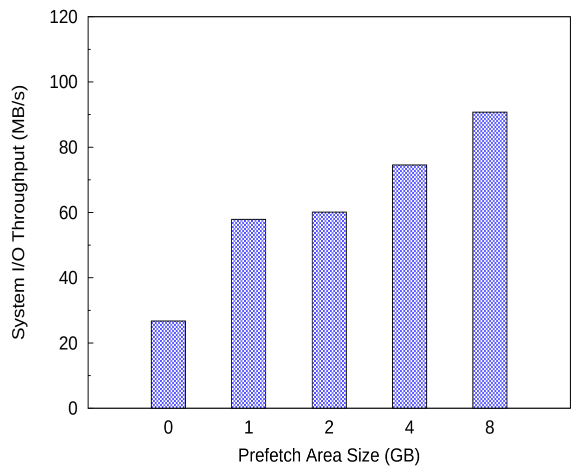
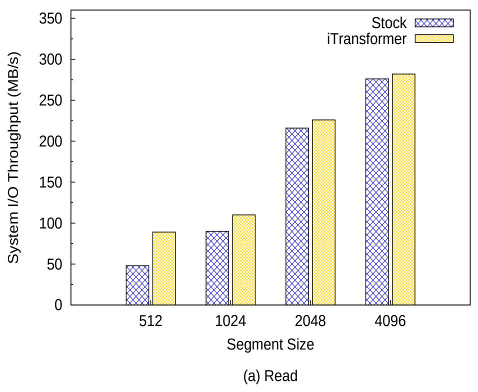


Figure 6.4: I/O throughput of system running *ior-mpi-io* with different prefetch area sizes. With a prefetch area of 0GB prefetching is effectively disabled.

its cost, here resulting in a 10% reduction in throughput. As the size of the prefetch unit increases, the benefit of prefetching increases while the cost of prefetching can be well amortized by the data in a large request. However, once the prefetch unit size is sufficiently large (such as 4MB in this experiment), further increasing it may lead to over-prefetching, in which excessive prefetched data may not be actually requested, thus diminishing the performance return.

Prefetch Unit	Throughput (MB/s)	Speedup (%)
0KB	26.7	0
64KB	23.9	-10
256KB	47.2	76
1MB	53.0	98
4MB	90.8	239
16MB	92.4	245

Table 6.2: I/O throughputs and their improvement ratios in the system running *ior-mpi-io* with different prefetch unit sizes. The ratios are calculated against the case of prefetch unit size 0KB wherein prefetching is effectively disabled.



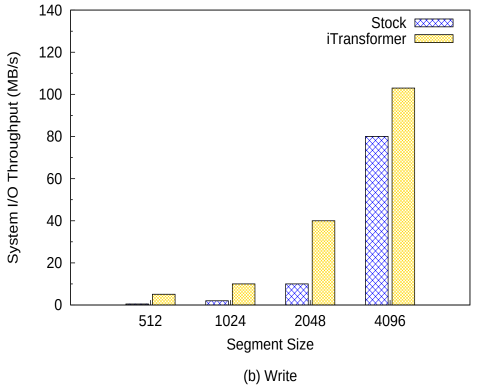


Figure 6.5: System I/O throughput with the *Hpio* benchmark for I/O reads (a) and I/O writes (b).

6.3.4 The *Hpio* Benchmark

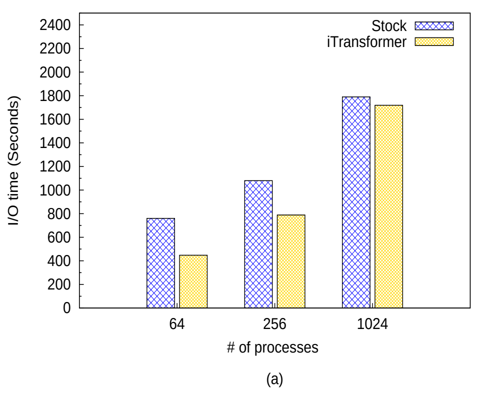
Hpio is a program designed by Northwestern University and Sandia National Laboratories to systematically evaluate I/O performance using a diverse set of access patterns [49]. This benchmark generates different data access patterns according to three parameters: *region_count*, *region_spacing*, and *region_size*. In the experiment we set *region_count* to 4096B, *region_spacing* to 8192B, and vary *region_size*, or access segment size, between 512B and 4096B. Access can be configured to be either read or write. We ran the benchmark to measure the throughput of the storage system for noncontiguous data accesses on disk. Three instances of the benchmark were concurrently executed in the experiment, each with 64 processes. Figure 6.5(a) and Figure 6.5(b) show the aggregate I/O throughputs for read and write requests, respectively.

In the experiment with read requests iTransformer increases system I/O throughput by 85%, 22%, 5.1%, and 2.2% for segment sizes 512B, 1024B, 2048B, and 4096B, respectively. The I/O pattern within each process is strided access, with a *re-*

gion_spacing between two consecutive requests. Though its access is noncontiguous, it is not as random as the pattern exhibited by the *ior-mpi-io* benchmark. This helps with throughputs in the stock system. On the other hand the space gaps between requests results in prefetched data not being fully used. This explains why the improvements for read requests with iTransformer are not as significant as those for *ior-mpi-io*. When the benchmark uses write requests the throughputs are much lower than those for the corresponding read requests and the throughput improvements made by iTransformer are 9X, 4X, 3X, and 30%, respectively, for segment sizes from 512B to 4096B. In the execution of the benchmark with write requests, we issued a sync command every second to flush dirty data in the system buffer cache to the disk and free filesystem data structures such as pagecache, dentries and inodes (*echo* 3 > */proc/sys/vm/drop_caches*). This creates a large number of small writes as well as small reads for recovering system metadata in the memory, significantly increasing randomness in the workload presented to the storage system and causing the throughput to plummet, especially when the segment is small. Leveraging SSD as a buffer, iTransformer absorbs small writes, and its prefetching also helps reads because the system metadata are usually co-located on the disk.

6.3.5 The *BTIO* Benchmark

BTIO is a Fortran MPI program designed to solve the 3D compressible Navier-Stokes equations using the MPI-IO library for its on-disk data access [14]. We ran the program using various numbers of processes with the input size coded as C in the benchmark, which generates a data set of about 6.8GB using non-collective I/O operations. We ran the program with the number of processes ranging from 64 to 1024. Three instances of the program were executed concurrently, each accessing its



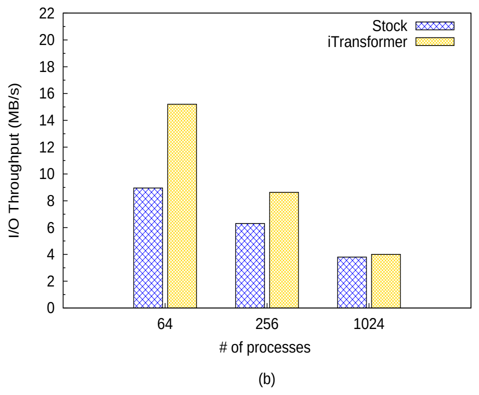


Figure 6.6: (a) I/O times, and (b) system I/O throughput, when running the BTIO benchmark with varying numbers of processes.

own 6.8GB file. Most I/O operations of the program are small random writes. The total I/O times reported by the program are shown in Figure 6.6(a), and the system I/O throughputs are shown in Figure 6.6(b).

When each instance used 64 processes I/O time was reduced by 41% and throughput was increased by 70% with iTransformer. However, when increasingly more processes are used, request sizes become smaller and iTransformer's performance advantage becomes smaller. For example, when the number of processes is 1024 for each instance, the request size is reduced to 200B and the total number of processes increases to 3072. With very small requests and very high access concurrency, the ability of iTransformer to form high-efficiency request streams is increasingly constrained.

We also used the *BTIO* benchmark to study the impact of the size of the in-SSD queue extension on the I/O throughput. In the experiment we ran one instance of the benchmark with 64 processes and varied the queue size between 1GB and 8GB. As shown in Figure 6.7, the I/O throughput is accordingly increased by 32%, 35%, 38%, and 40%, respectively, compared to that the stock system. Because the

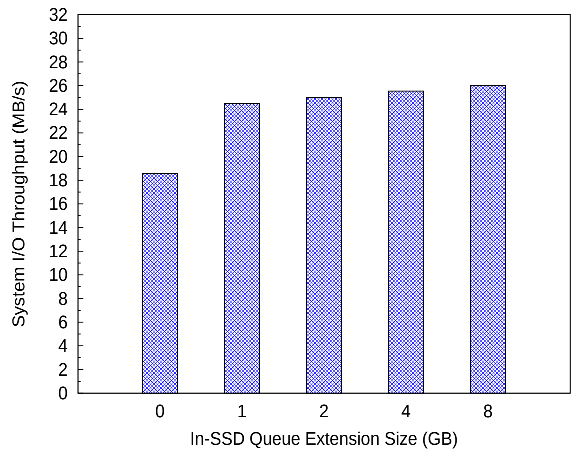


Figure 6.7: I/O throughput of *BTIO* using iTransformer with different SSD queue extension sizes. A queue of 0GB refers to the stock system.

data set of the program is only 6.8GB, having a 8GB queue size is sufficient to buffer the entire written data set. In addition, because of detected random access iTransformer reroutes write requests to the SSD. Therefore, the I/O throughput with the 8GB queue reflects SSD access speed. Interestingly, with a queue size as small as 1GB the throughput is only 8% lower than the optimal value. In the program there is substantial computation time (around 50% depending on I/O speed) between I/O activities. This results in periodic disk idle times, which gives iTransformer opportunities to write back its in-SSD dirty data to the disk during the idle periods, thereby hiding the disk operations behind the program's execution.

6.3.6 The *S3aSim* Benchmark

S3aSim is a computational biology program designed to simulate sequence similarity search [18]. In the program query sequences are compared against a sequence database. In this experiment each sequence in the database is divided into 16 frag-

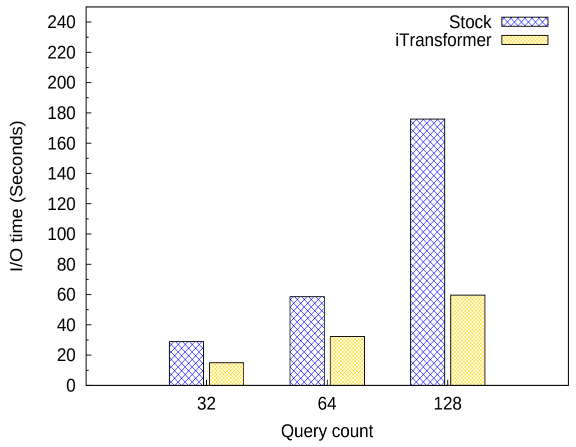


Figure 6.8: I/O times of the *S3aSim* benchmark with varying numbers of queries.

ments. For the parameters of the program, we set the minimum size of each query and database sequence to 100B, and set the maximum size to 10,000B. We ran three concurrent instances of the program, each with 64 processes. The amount of accessed data depends on the number of queries, up to 6.4GB for each instance in our experiments. Figure 6.8 shows the I/O times reported by the program when we ran it with the number of queries ranging from 32 to 128. Major accesses of the program are random writes of search results with various request sizes. Compared to the stock system, iTransformer reduces I/O times by up to 66%. The improvement is greater with larger query count. With more queries, write requests are scattered into a large disk space and the access locality becomes weak, which gives iTransformer greater opportunity to improve throughput.

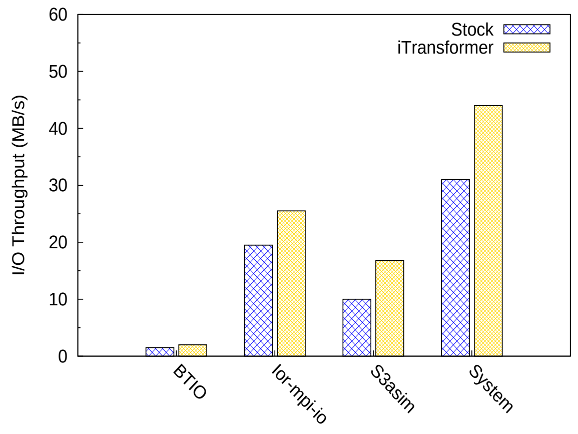


Figure 6.9: I/O throughput of *ior-mpi-io*, *BTIO*, and *S3aSim* as well as aggregate system throughput, when run concurrently.

6.3.7 Heterogeneous Workloads

Next we study the performance of iTransformer with different programs running concurrently. We select three programs with different access patterns, *ior-mpi-io*, *BTIO*, and *S3aSim*, and run one instance of each concurrently to read from (*ior-mpi-io*) or write to (*BTIO* and *S3aSim*) three different files. Each program runs with 64 processes.

Figure 6.9 shows the I/O throughput of each program as well as the aggregate system throughput with and without iTransformer. Because of the random access pattern that makes *ior-mpi-io* scatter its reads among several disk regions, iTransformer enables prefetching to serve its requests from SSDs. However, because of the concurrent random write requests from the other two benchmarks, the hit ratio of reads of *ior-mpi-io* in the prefetch area is 16% lower than when the system only serves read requests from one *ior-mpi-io* instance, with the I/O throughput of the program increasing by only 30% compared to the stock system. For *BTIO* I/O throughput is increased by 33% to 2MB/s, which is still very low because of its small request size (about 800B). *S3aSim* greatly benefits from the buffering effect of the in-SSD queue, and its I/O throughput is increased by 68%. Compared with the aggregate I/O throughput of the stock system, iTransformer improves the system's I/O performance by 42%.

Chapter 7

Conclusions and Future Work

In this chapter we first conclude this dissertation with a summary of our major contributions in four layers of the software stacks for parallel I/O. Then we discuss the limitations in the implementation and evaluation of the proposed solutions. In the end, we suggest several directions for future work.

7.1 Contributions

Leveraging *operating system process management*, we proposed a scheme, *DualPar*, to allow a parallel program to alternate between two execution modes, the normal computation-driven mode, and when justified by performance considerations, a data-driven mode in which process execution and I/O service are coordinated to improve I/O efficiency. As such the processes' execution is determined by data availability. While the timing of data access is no longer constrained by the timing of individual I/O function calls, access locality can be well exploited for optimal disk efficiency. DualPar has been implemented in the MPICH2 MPI-IO library to support dual-mode execution of MPI programs. Our experimental evaluation using representative benchmarks on the PVFS2 file system with various access locality and I/O intensity shows that DualPar can significantly improve I/O efficiency in various scenarios, whether or not collective I/O is used.

In *MPI/IO middleware layer*, we proposed, designed, and implemented a new collective I/O scheme, *resonant I/O*, that makes resonance—a phenomenon describing the increase in performance when there is a match between request patterns and data striping patterns—a common case. Resonant I/O makes the client-side implementation of collective I/O aware of the I/O configuration in its rearrangement of requests without compromising the portability of client-side middleware and the flexibility of server-side configuration. Our experimental results show significant increases—up to 157%—in I/O throughput for commonly used parallel I/O benchmarks. Resonant I/O demonstrated advantages both at scale, and in the presence of competition for I/O services. Finally, resonant I/O has not been observed to substantially degrade performance (relative to ROMIO collective I/O) in any test scenario.

For *parallel file system layer*, we 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.

In *disk I/O scheduling layer*, we proposed the *iTransformer* scheme to use a relatively small SSD space to facilitate the scheduling of disk requests in response to the increasingly large I/O-request concurrency and its correspondingly serious challenge to hard-disk-based storage systems. In the design we exploit SSD's large size and low power consumption, relative to DRAM, to more thoroughly exploit spatial locality in the requests for high storage system performance. Taking advantage of SSD's non-volatility, we decouple data servicing by the hard disk from process execution by squeezing the data write-back to the disk, and prefetching from the disk, into the background, or when the disk is idle. In addition, iTransformer takes effect in an opportunistic fashion, enabling the SSD's involvement only it is expected to enhance the locality and its cost is justified. We have implemented iTransformer in the Linux kernel as a module and extensively evaluated it on a large PVFS2 cluster of 120 nodes/5860 cores. The experimental measurements from running representative benchmarks with greatly varying access patterns, such as *BTIO* and *S3aSim*, demonstrate significant I/O performance improvements by up to 3X, and 35% on average.

7.2 Limitations

While our experiments have shown that the proposed approaches are promising techniques for alleviation of increasingly serious I/O bottleneck in high-performance computing, there are some limitations in their implementation and evaluation that will be addressed in future work. First, we will use asynchronous I/O to fully exploit the performance potential of resonant I/O. As current ADIO does not support real asynchronous I/O, we will use additional threads to implement asynchronous I/O. Second, the dedicated cluster used for the evaluation of IOrchestrator is of relatively small scale, and the larger cluster at OSC was not available for dedicated use for the evaluation of resonant I/O. Our plan includes evaluating both of them on the Darwin cluster at Los Alamos National Laboratory to obtain more insights into its performance characteristics. Third, for evaluation of this thesis work only the PVFS2 parallel file system is used. Not to lose the generality, we plan to benchmark the modified system on top of other state-of-the-art parallel file systems, such as Lustre, to further evaluate their potential.

7.3 Future Work

In this section, we present research directions for future work, for which we will concern both performance improvement and performance guarantee in shared storage clusters. We identify several difficulties and challenges in high-performance computing and provide possible solutions in some cases.

7.3.1 Improving Unaligned Parallel File Access

Unaligned data access has been identified as one of the I/O bottlenecks for high-performance computing [109, 70, 105]. When files are striped over multiple data servers in a parallel I/O system, requests to the files are decomposed into a number of sub-requests, each served by one server. If a request is not well aligned with the striping pattern such decomposition can make the first and last sub-requests much smaller than the striping unit. Because hard-disk-based servers can be much less efficient in serving small requests than for large ones, the system exhibits heterogeneity in serving different segments. Furthermore, a request is not considered complete until its slowest sub-request is. Thus the throughput of the entire system can be bottlenecked by the inefficiency of serving the smaller requests, or fragments, especially when synchronous requests are assumed. To make the situation worse, the larger the request, or the more data servers the requested data is striped over, the larger the detrimental performance effect of serving fragments can be. This effect can become the Achilles heel of a parallel I/O system performance seeking scalability with large sequential accesses. We believe that a hybrid storage system which uses solid-state disks to serve the fragments and uses hard disks to serve larger units can help solve the bottleneck, especially in the environment where unaligned parallel file access accounts for a significant amount of I/Os.

7.3.2 Design of Collective I/O for Multi-core Clusters

Collective I/O [99] is widely used on compute servers to improve spatial locality of data access on disks through file domain repartition for I/O aggregators. While the existing implementation of collective I/O in ROMIO/MPI-IO library only concerns increasing request size, the communication overhead is underestimated and could become performance bottleneck when large amount of data has to be transferred among processes in all-to-all communication, after data become ready in main memory of compute servers. We believe that it is possible to reduce communication overhead for an HPC cluster installation using multi-core multi-socket processors by partitioning the file domain according to process affinity. For example, if an aggregator is assigned to access file domain which is only needed by processes on the same socket, both inter-node and inter-socket communication can be avoided. If the requested data is only needed by processes on a node, inter-node communication can be avoided. In fact, we are trading spatial locality for communication efficiency. And the problem is if we can find an optimized partition, using which both communication overhead and negative impact on spatial locality can be minimized. We argue that this is possible because file access domain on a single socket or node could be quite large because of highly increased process parallelism on multi-core HPC clusters. Therefore collective I/O domain partition algorithm needs to be redesigned to consider the impact of reduced request size on I/O efficiency. Moreover, we also want to study the impact of process-core affinity on efficiency of MPI collective I/O [111, 42].

7.3.3 Improving Performance of Parallel Writes

Memory references are usually handled at a much smaller granularity, usually align with processor cache line size, than the size of disk I/Os. When referenced data do not present in memory, the entire page (4KB or 8KB), which contains the requested data, must be fetched into memory by the operating system. This is a blocking read operation in the current implementation of Linux operating system, which could potentially cause a performance bottleneck for small synchronous and asynchronous writes to disks [80, 101, 94], which is an order of magnitude slower than DRAM memory. We call this read-before-write problem, which can significantly compromise write performance. We believe because of this issue the write performance in high-performance computing could be even worse since many scientific and engineering applications process a large amount of data set with weak temporal locality in the memory of data servers. Useche et al. [101] proposes to replace the blocking reads with asynchronous non-blocking reads by constructing temporary memory-buffers for first-time writes and their updates. However, we argue that using DRAM based approach can not only cause the risk of losing data during system failure, but also incurs stiff competition for memory resource with other system components for data caching and prefetching. A possible solution is that creating the buffer on solid state disks to take advantage of its much better read performance, compared to disks. In addition, we can also consider serve such read requests in batch or collectively to further improve I/O efficiency by exploring existing device parallelism of solid state disks and spatial locality of data access.

7.3.4 I/O-intensive Scientific Workflow Streaming and In-situ Execution

Because of an order of magnitude performance difference between DRAM memory and disks, even parallel file systems for high-performance computing are not able to keep up with increased data rates in the future. Thanks to the observation that scientific applications for computation, simulation, and visualization are frequently coupled to form scientific workflows, a large body of recent research has focused on I/O efficiency of scientific workflows [34, 76, 33, 112] during its streaming in compute servers and I/O staging area. Lofstead et al. proposes ADIOS [75] to facilitate selection of efficient I/O methods for users and maintain an optimized intermediate file format for workflows. Abbasi et al. [33] showed that manageability of intermediate results in streaming is performance-critical for high performance computing. Most recently, Zhang et al. [112] designed and implemented a novel in-situ execution environment to further reduce the amount of data which are transferred in workflow streaming in communication networks. However, scientific applications have their own I/O characteristics, such as request size, I/O burstiness, temporal locality, and requirements for quality of services. We believe it is still an open question that how to leverage application-specific knowledge for scheduling multiple concurrent scientific workflows to achieve optimal utilization of limited system resource, e.g. DRAM size and networking bandwidth.

REFERENCES

[1] A. morton, linux code on anticipatory scheduling. `http:////lxr.linux.no/#linux+v2.6.28.5/block/cfq-iosched.c`.

[2] Cluster file systems, inc. “lustre: A scalable, robust, highly-available cluster file system”. `http://www.lustre.org/`.

[3] Hpio i/o benchmark. `http://cholera.ece.northwestern.edu/~aching/research-webpage/io.html`.

[4] Interleaved or random (ior) benchmarks. `http://www.cs.dartmouth.edu/pario/examples.html`.

[5] Iozone filesystem benchmark. `http://www.iozone.org`.

[6] J. axboe, blktrace. `http://linux.die.net/man/8/blktrace`.

[7] J. axboe, completely fair queueing (cfq) scheduler. `http://en.wikipedia.org/wiki/CFQ`.

[8] J. axboe, deadline scheduler. `http://en.wikipedia.org/wiki/Deadline_scheduler`.

[9] J. axboe, noop scheduler. `http://en.wikipedia.org/wiki/Noop_scheduler`.

[10] Lustre: Building a file system for 1000-node clusters. `http://off.net/cv/papers/ols2003.ps`.

[11] Memcached. `http://memcached.org/`.

[12] Message passing interface (mpi) standard, “mpich2”. http://www.mcs.anl.gov/research/projects/mpich2/.

[13] Mpi-tile-io benchmark. http://www-unix.mcs.anl.gov/thakur/pio-benchmarks.html.

[14] Nas parallel benchmarks, nasa ames research center. http://www.nas.nasa.gov/Software/NPB/.

[15] Noncontig i/o benchmark. http://www-unix.mcs.anl.gov/thakur/pio-benchmarks.html.

[16] Parallel i/o benchmarking consortium. http://www-unix.mcs.anl.gov/pio-benchmark/.

[17] Parallel virtual file system, version 2, “pvfs2”. http://www.pvfs.org/.

[18] S3asim i/o benchmark. http://www-unix.mcs.anl.gov/thakur/s3asim.html.

[19] Sun microsystems. lustre 1.6 operations manual. http://docs.sun.com/app/docs/doc/820-3681.

[20] Microsoft windows readyboost. 2008. http://www.microsoft.com/windows/windows-vista/features/readyboost.aspx.

[21] Solid-state drive. 2011. http://en.wikipedia.org/wiki/Solid-state-drive.

[22] 10-gigabit ethernet technical brief. 2012. http://www.extremenetworks.com/libraries/techbriefs/TB10GbE_1536.pdf.

[23] Asynchronous i/o. 2012. http://msdn.microsoft.com/en-us/library/windows/desktop/aa365683(v=vs.85).aspx.

[24] Computational fluid dynamics. 2012. http://www.lanl.gov/orgs/t/t3/combustion.shtml.

[25] Ext3 file system. 2012. http://www.redhat.com/resourcelibrary/whitepapers/ext3.

[26] Infiniband architecture. 2012. http://www.intel.com/content/www/us/en/data-center/data-center-management/infiniband-architecture-general.html.

[27] K computer, riken advanced institute for computational science. 2012. http://www.aics.riken.jp/en/kcomputer/system-2.html.

[28] The large hadron collider. 2012. http://lhc.web.cern.ch/lhc/.

[29] Mira, argonne leadership computing facility. 2012. http://www.alcf.anl.gov/resource-guides/intrepid-and-surveyor-guide.

[30] Ntfs file system. 2012. http://www.ntfs.com/.

[31] Nuclear weapon simulations show performance in molecular detail, purdue university. 2012. http://www.purdue.edu/newsroom/research/2012/120605BagchiWeapons.html.

[32] Sequoia, advanced simulation and computing at llnl. 2012. https://computing.llnl.gov/?set=resources&page=OCF_resources#sequoia.

[33] H. Abbasi, G. Eisenhauer, M. Wolf, K. Schwan, , and S. Klasky.

[34] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng.

[35] A. V. Aho, P. J. Denning, and J. D. Ullman. Principles of optimal page replacement.

[36] S. Barua, R. Thulasiram, and P. Thulasiraman. High performance computing for a financial application using fast fourier transform. In *EURO-PAR 2005 PARALLEL PROCESSING Lecture Notes in Computer Science*, 2005.

[37] S. Baylor and C. Wu. Parallel i/o workload characteristics using vesta. In *Workshop on Input/Output in Parallel and Distributed Systems (IPPS'95)*, 1995.

[38] T. Bisson and S. Brandt. Reducing hybrid disk write latency with flash-backed I/O requests. In *the 15th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2007.

[39] S. Byna, Y.Chen, X. Sun, R. Thakur, and W. Gropp. Parallel i/o prefetching using mpi file caching and i/o signatures. In *The Conference on High Performance Computing Networking, Storage and Analysis (SC'08)*, 2008.

[40] P. Cao, E. Felten, and K. Li. Application-controlled file caching policies. In *the USENIX Summer 1994 Technical Conference(USTC'94)*, 1994.

[41] A. Caulfield, L.M.Grupp, and S. Swanson. Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. In *Internatonal Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*, 2009.

[42] K. Cha and S. Maeng.

[43] F. Chang. Using speculative execution to automatically hide i/o latency. In *Carnegie Mellon Ph.D Dissertation CMU-CS-01-172*, 2001.

[44] F. Chang and G. A. Gibson. Automatic i/o hint generation through speculative execution. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI'99)*, 1999.

[45] F. Chen, D. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memroy based solid state drives. In *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'09)*, 2009.

[46] F. Chen, D. Koufaty, and X. Zhang. Hystor: making the best use of solid state drives in high performance storage systems. In *International Conference on Supercomputing (ICS'11)*, 2011.

[47] A. Ching, A. Choudhary, K. Coloma, and W. Liao. Noncontiguous i/o accesses through mpi-io. In *the IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid'03)*, 2003.

[48] A. Ching, A. Choudhary, W. Liao, R. Ross, , and W. Gropp. Efficient structured data access in parallel file systems. In *the IEEE International Conference on Cluster Computing (ICCC'03)*, 2003.

[49] 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 and Distributed Processing Symposium (IPDPS'96)*, 1996.

[50] A. Ching, A. Choudhary, W. Liao, L. Ward, and N. Pundit. I/o characteristics and methods for storing structured scientific data. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS'96)*, 1996.

[51] A. Ching, A. Choudhary, W. Liao, L. Ward, and N. Pundit. Improving response time in cluster-based web servers through coscheduling. In *Proceedings of IEEE Internatinal Parallel and Distributed Processing Symposium (IPDPS'04)*, 2004.

[52] P. Crandall, R. Aydt, A. Chien, and D. Reed. Input/output characteristics of scalable parallel applications. In *Supercomputing (SC'95)*, 1995.

[53] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI'04)*, 2004.

[54] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang. Diskseen: Exploiting disk layout and access history to enhance i/o prefetch. In *USENIX Annual Technical Conference(USENIX'07)*, 2007.

[55] D. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, pages 306–318.

[56] K. Fraser and F. Chang. Operating system i/o speculation: How two invocations are faster than one. In *USENIX Annual Technical Conference*, 2003.

[57] A. Gulati, A. Merchant, and P. Varman. pclock: an arrival curve based approach for qos guarantees in shared systems. In *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'07)*, 2007.

[58] A. Gulati, G. Shanmuganathan, X. Zhang, and P. Varman. Demand based hierarchical qos using storage resource pools. In *USENIX Annual Technical Conference*, 2012.

[59] T. Harris, M. Abadi, R. Isaacs, and R. McIlroy. AC: Composable asynchronous io for native languages. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2011.

[60] G.-R. Hoffmann. High performance computing and networking for numerical weather prediction. In *the International Conference and Exhibition on High-Performance Computing and Networking*, 1994.

[61] H. Huang, W. Hung, and K. Shin. Fs2: Dynamic data replication in free disk space for improving disk performance and energy consumption. In *ACM Symposium on Operating Systems Principles (SOSP'05)*, 2005.

[62] L. Huang, G. Peng, and T. Chiueh. Multi-dimensional storage virtualization. In *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'04)*, 2004.

[63] J. Ilroy, C. Randriamaro, and G. Utard. Improving mpi-i/o performance on pvfs. In *Euro-Par'01*, 2001.

[64] S. Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous i/o. In *ACM Symposium on Operating Systems Principles (SOSP'01)*, 2001.

[65] S. Jiang and X. Zhang. Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'02)*, 2002.

[66] T. Johnson and D. Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *International Conference on Very Large Data Bases(VLDB'94)*, 1994.

[67] M. Kandemir, S. Son, and M. Karakoy. Improving i/o performance of applications through compiler-directed code restructuring. In *the 6th USENIX Conference on File and Storage Technologies (FAST'08)*, 2008.

[68] M. Kim. Synchronized disk interleaving. *IEEE Transactions on Computers*, pages 978–988.

[69] D. Kotz. Disk-directed i/o for mimd multiprocessors. *ACM Transactions on Computer Systems*, 15.

[70] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock. I/o performance challenges at leadership scale. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis(SC'09)*, 2009.

[71] R. Latham, N. Miller, R. Ross, and P. Carns. A next generation parallel file system for linux clusters. *LinuxWorld Magazine*, 2004.

[72] T. Latham, W. Gropp, R. Ross, and R. Thakur. Extending the mpi-2 generalized request interface. In *the 14th European PVM/MPI Users' Group Meeting*, 2007.

[73] A. Leventhal. Flash storage memory. In *Communications of ACM*, 2008.

[74] S. Liang, S. Jiang, and X. Zhang. Step: Sequentiality and thrashing detection based prefetching to improve performance of networked storage servers. In *International Conference On Distributed Computing Systems (ICDCS'07)*, 2007.

[75] J. Lofstead, S. Klasky, M. Booth, H. Abbasi, F. Zheng, M. Wolf, and K. Schwan. Petascale i/o using the adaptable i/o system. In *Cray User's Group*, 2009.

[76] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf. Managing variability in the io performance of petascale storage systems. In *The Conference on High Performance Computing Networking, Storage and Analysis (SC'10)*, 2010.

[77] T. Ma, G. Bosilca, A. Bouteiller, and J. Dongarra. Hierknem: An adaptive framework for kernel-assisted and topology-aware collective communications on many-core clusters. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS'12)*, 2012.

[78] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. 9(2):78–117, 1970.

[79] J. May. Parallel i/o for high performance computing. pages 16–17, 2001.

[80] M. Mckusick, K. Bostic, M. Karels, and J. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System.* Addison Wesley, 1996.

[81] N. Megiddo and D. S. Modha. Arc: a self-tuning, lowoverhead replacement cache. In *the 2nd USENIX Conference on File and Storage Technologies (FAST'03)*, 2003.

[82] A. Morton. Linux: Anticipatory i/o scheduler. `http://kerneltrap.org/node/567`.

[83] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. Ellis, and M. Best. File-access characteristics of parallel scientific workloads. *IEEE Transactions on Parallel and Distributed Systems*, 7:1075–1089, 1996.

[84] J. Ousterhout. Scheduling techniques for concurrent systems. In *International Conference on Distributed Computing Systems (ICDCS'82)*, 1982.

[85] R. Pai, B. Pulavarty, and M. Cao. Linux 2.6 performance improvement through readahead optimization. In *the Linux Symposium*, 2004.

[86] A. Patrizio. Ucsd plans first flash-based supercomputer. 2009. `http://www.internetnews.com/hardware/article.php/3847465`.

[87] H. Payer, M. A. Sanvido, Z. Bandic, and C. M. Kirsch. Combo Drive: Optimizing cost and performance in a heterogeneous storage device. In *the 1st Workshop on integrating solid-state memory into the storage hierarchy*, 2009.

[88] R. Prabhakar, S. Vazhkudai, Y. Kim, A. Butt, M. Li, and M. Kandemir. Provisioning a multi-tiered data staging area for extreme-scale machines. In *International Conference On Distributed Computing Systems (ICDCS'11)*, 2011.

[89] T. Pritchett and M. Thottethodi. SieveStore: a highly-selective, ensembel-level disk cache for cost-performance. In *37th International Symposium on Computer Architecture*, 2010.

[90] J. Ren and Q. Yang. I-CASH: intelligently coupled array of ssd and hdd. In *The 17th IEEE Symposium on High Performance Computer Architecture*, 2011.

[91] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. In *The 13th ACM Symposium on Operating Systems Principles(SOSP'92)*, 1992.

[92] F. Schmuck and R. Haskin. Gpfs: A shared-disk file system for large computing clusters. In *the 1st USENIX Conference on File and Storage Technologies (FAST'02)*, 2002.

[93] K. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective i/o in panda. In *Supercomputing (SC'95)*, 1995.

[94] D. Simha, M. Lu, and T. Chiueh. An update-aware storage system for low-locality update-intensive workloads. In *Internatonal Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*, 2012.

[95] D. Skourtis, S. Kato, and S. Brandt.

[96] H. Song, C. Leangsuksun, and R. Nassar. Availability modeling and analysis on high performance cluster computing systems. In *International Conference on Availability, Reliablity and Security*, 2006.

[97] M. Srinivasan and P. Saab. Flashcache: a general purpose writeback block cache for linux. 2011. `https://github.com/facebook/flashcache`.

[98] D. Steere. Exploiting the non-determinism and asynchrony of set iterators to reduce aggregate file i/o latency. In *ACM Symposium on Operating Systems Principles (SOSP'97)*, 1997.

[99] R. Thakur, W. Gropp, and E. Lusk. An abstract-device interface for implementing portable parallel-i/o interfaces. In *the 6th Symposium on the Frontiers of massively Parallel Computation*, 1996.

[100] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective i/o in romio. In *the 7th Symposium on the Frontiers of massively Parallel Computation*, 1999.

[101] L. Useche, R. Koller, R. Rangaswami, and A. Verma. Truly non-blocking writes. In *3rd USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'11)*, 2011.

[102] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. Ganger. Argon: Performance insulation for shared storage servers. In *the 5th USENIX Conference on File and Storage Technologies (FAST'07)*, 2007.

[103] M. Wachs and G. Ganger. Co-scheduling of disk head time in cluster-based storage. In *International Symposium on Reliable Distributed Systems (SRDS'09)*, 2009.

[104] A. Wang, P. Reiher, G. Popek, and G. Kuenning. Conquest: better performance through a disk/persistent-ram hybrid file system. In *USENIX Annual Technical Conference*, 2002.

[105] L. Ward. A brief parallel i/o tutorial. In *SANDIA REPORT, SAND2010-1915*, 2010.

[106] Y. Xu and S. Jiang. A scheduling framework that makes any disk schedulers non-work-conserving solely based on request characteristics. In *the 9th USENIX Conference on File and Storage Technologies (FAST'11)*, 2011.

[107] J. Yang, D. Minturn, and F. Hady. When poll is better than interrupt. In *10th USENIX Conference on File and Storage Technologies(FAST'12)*, 2012.

[108] Y.Chen, S. Byna, X. Sun, R. Thakur, and W. Gropp. Hiding i/o latency with pre-execution prefetching for parallel applications. In *The Conference on High Performance Computing Networking, Storage and Analysis (SC'08)*, 2008.

[109] H. Yu, R. Sahoo, C. Howson, G. Almasi, J. Castanos, M. Gupta, J. Moreira, J. Parker, T. Engelsiepen, R. Ross, R. Thakur, R. Latham, and W. Gropp. High performance file I/O for the bluegene/l supercomputer. In *The 12th Inter-*

national Symposium on High-Performance Computer Architecture (HPCA'06), 2006.

[110] Y. Yu, P. K. Gunda, and M. Isard. Distributed aggregation for data-parallel computing: Interfaces and implementations. In *ACM Symposium on Operating Systems Principles (SOSP'09)*, 2009.

[111] C. Zhang, X. Yuan, and A. Srinivasan. Processor affinity and mpi performance on smp-cmp clusters. In *IEEE International Parallel and Distributed Processing, Workshops, and PhD Forum*, 2010.

[112] F. Zhang, C. Docan, M. Parashar, S. Klasky, N. Podhorszki, and H. Abbasi. Enabling in-situ execution of coupled scientific workflow on multi-core platform. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS'12)*, 2012.

[113] X. Zhang, F. Chen, and S. Jiang. Smartsaver: Turning flash drive into a disk energy saver for mobile computers. In *International Symposium on Low Power Electronics and Design*, 2006.

[114] X. Zhang, K. Davis, and S. Jiang. Iorchestrator: Improving the performance of multi-node i/o systems via inter-server coordination. In *The Conference on High Performance Computing Networking, Storage and Analysis (SC'10)*, 2010.

[115] X. Zhang, K. Davis, and S. Jiang. Qos support for end users of i/o-intensive applications using shared storage systems. In *The Conference on High Performance Computing Networking, Storage and Analysis (SC'11)*, 2011.

[116] X. Zhang, K. Davis, and S. Jiang. itransformer: Using ssd to improve disk scheduling for high-performance i/o. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS'12)*, 2012.

[117] X. Zhang, K. Davis, and S. Jiang. Opportunistic data-driven execution of parallel programs for efficient i/o services. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS'12)*, 2012.

[118] X. Zhang and S. Jiang. Interferenceremoval: Removing interference of disk access for mpi programs through data replication. In *International Conference on Supercomputing (ICS'10)*, 2010.

[119] 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 *IEEE International Parallel and Distributed Processing Symposium (IPDPS'09)*, 2009.

[120] X. Zhang, Y. Xu, and S. Jiang. Youchoose: A performance interface enabling convenient and efficient qos support for consolidated storage systems. In *The 27th IEEE Symposium on Massive Storage Systems and Technologies (MSST'11)*, 2011.

ABSTRACT

RETHINKING THE DESIGN AND IMPLEMENTATION OF THE I/O SOFTWARE STACK FOR HIGH-PERFORMANCE COMPUTING

by

XUECHEN ZHANG

August 2012

Advisor: Dr. Song Jiang

Major: Computer Engineering

Degree: Doctor of Philosophy

Current I/O stack for high-performance computing is composed of multiple software layers in order to hide users from complexity of I/O performance optimization. However, the design and implementation of a specific layer is usually carried out separately with limited consideration of its impact on other layers, which could result in suboptimal I/O performance because data access locality is weakened, if not lost, on hard disk, a widely used storage medium in high-end storage systems. In this dissertation, we experimentally demonstrated such issues in four different layers, including operating system process management layer and MPI-IO middleware layer on compute server side, and parallel file system layer and disk I/O scheduling layer on data server side.

This dissertation makes four contributions towards solving each of the issues. First, we propose a data-driven execution model for *DualPar* to explore opportunity of effective I/O scheduling to alleviate I/O bottleneck via cooperation between the I/O and process schedulers. Its novelty is on the ability to obtain a pool of presorted requests to I/O scheduler in its data-driven execution mode by using process pre-execution and prefetching techniques.

Second, realizing that well-formed locality for an MPI program by using collective I/O can be seriously compromised by non-determinism in process scheduling, we proposed *Resonant I/O*, to match the data request pattern with the pattern of file striping over multiple data servers to improve disk efficiency.

Third, since the conventional practice for I/O parallelism using file striping may compromise on-disk data access locality, we proposed *IOrchestrator* scheduling framework which is implemented in PVFS2 parallel file system 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.

Fourth, we developed *iTransformer*, a scheme that employs a small SSD to schedule requests for the data on disk. Being less space constrained than with more expensive DRAM, iTransformer can buffer larger amounts of dirty data before writing it back to the disk, or prefetch a larger volume of data in a batch into the SSD. In both cases high disk efficiency can be maintained for highly concurrent requests.

AUTOBIOGRAPHICAL STATEMENT

XUECHEN ZHANG

Xuechen Zhang is a graduate student of Department of Electrical and Computer Engineering at Wayne State University. He received his B.S. degree in computer science and technology from Liaoning Normal University, Dalian, Liaoning, China in 2006, and M.S. degree in computer engineering from Wayne State University, Detroit, MI, USA in 2010.

His research interests include operating system, file and storage system, and parallel file systems, parallel I/O and MPI-IO library, and storage QoS. He has published more than 10 research papers in premier international conferences of supercomputing. His "YouChoose" paper was selected as the sole best student paper at *IEEE Conference on Massive Data Storage*, 2011. For his research excellence, he is a recipient of the *Summer Dissertation Fellowship* (2012), *Thomas C. Rumble University Graduate Fellowship* (2011), and *Olbrot Travel Award for Excellence in Graduate Student Research* (2011).

He has been a lab instructor for an undergraduate course Digital Circuits for five consecutive semesters. And he worked as a teaching assistant for Prof. Cheng-zhong Xu and Prof. Harpreet Singh for teaching graduate courses Computer Networking and Programming in 2008, and Switching Circuits in 2010, respectively. He has been receiving outstanding evaluations each semester from his students.