LiU: Hiding Disk Access Latency for HPC Applications with a New SSD-Enabled Data Layout

Dachuan Huang[†] Xuechen Zhang[‡] Wei Shi[♮†] Mai Zheng[†] Song Jiang[*] Feng Qin[†]

[†]Dept. of Computer Science and Engineering
The Ohio State University

[‡]School of Computer Science
Georgia Institute of Technology

[♮] Dept. of Computer Science and Technology
Tsinghua University

[*]Dept. of Electrical and Computer Engineering
Wayne State University

***Abstract*—Unlike in the consumer electronics and personal computing areas, in the HPC environment hard disks can hardly be replaced by SSDs. The reasons include hard disk's large capacity, very low price, and decent peak throughput. However, when latency dominates the I/O performance (e.g., when accessing random data), the hard disk's performance can be compromised. If the issue of high latency could be effectively solved, the HPC community would enjoy a large, affordable and fast storage without having to replace disks completely with expensive SSDs.**

In this paper, we propose an almost latency-free hard-disk-dominated storage system called *LiU* for HPC. The key technique is leveraging limited amount of SSD storage for its low-latency access, and changing data layout in a hybrid storage hierarchy with low-latency SSD at the top and high-latency hard disk at the bottom. If a segment of data would be randomly accessed, we lift its top part (the head) up in the hierarchy to the SSD and leave the remaining part (the body) untouched on the disk. As a result, the latency of accessing this whole segment can be removed because access latency of the body can be hidden by the access time of the head on the SSD. Combined with the effect of prefetching a large segment, *LiU* (Lift it Up) can effectively remove disk access latency so disk's high peak throughput can now be fully exploited for data-intensive HPC applications.

We have implemented a prototype of *LiU* in the PVFS parallel file system and evaluated it with representative MPI-IO micro-benchmarks, including *mpi-io-test*, *mpi-tile-io*, and *ior-mpi-io*, and one macro-benchmark *BTIO*. Our experimental results show that *LiU* can effectively improve the I/O performance for HPC applications, with the throughput improvement ratio up to 5.8. Furthermore, *LiU* can bring much more benefits to sequential-I/O MPI applications when the applications are interfered by other workloads. For example, *LiU* improves the I/O throughput of *mpi-io-test*, which is under interference, by 1.1-3.4 times, while improving the same workload without interference by 15%.

I. Introduction

Some mission-critical systems (e.g. [1]) and a few experimental prototype systems (e.g. Gordon [12]) pursue I/O performance without much consideration of the cost. However, for most HPC systems it is impractical to replace the hard disks completely with SSDs, even when I/O performance is important for their workloads. There are two main reasons. First, data-intensive applications, which become increasingly popular on today's HPC systems, usually have a huge amount of data as input. These data are stored in tens of thousands of hard disks. Replacing all these disks with SSDs for high-performance data access is too expensive. Second, the hard disk has many advantages that match the needs of HPC well, including large capacity, rapidly decreasing per-GB price, and decent peak throughput when accessed sequentially.

On the other hand, the hard disk has its well-known Achilles' heel in performance, which is access latency. As long as the requested data is not at the current position of the disk head, the hard disk has to initiate time-consuming mechanical operations, i.e., spend potentially long latency time, to relocate the disk head before it can start data access. If substantial amount of random requests are involved in the disk access, the latency would turn a hard disk into a device of unacceptably poor performance. If this latency issue cannot be effectively addressed, then hard disks cannot meet the requirements of HPC systems. As a result, we cannot benefit from disks' appealing advantages.

Equipped with a limited amount of SSD space, we could remove the latency associated with disk access for HPC applications so that the overall performance can be competitive again even compared purely with the SSD. While using SSD for caching or using main memory for prefetching can hide disk access time, which includes access latency, some unique characteristics of HPC applications challenge these conventional practices.

Recent studies [14], [20], [24] have proposed hybrid storage systems in which SSD is used as a cache for the hard disk to store small and frequently accessed data items, such as small files and metadata. These approaches can be effective for I/O requests that have strong temporal locality. However, in the HPC environment, workloads often exhibit much weaker temporal locality with one-time or limited times of access in large data sets, making the caching strategy less attractive. Prefetching can be effective in hiding disk latency if a long sequence of data access can be accurately predicted. However, in a HPC system multiple processes running on different compute nodes simultaneously issue requests to different files or different parts of a large file, it is difficult to prefetch in the right order, and even if accurate prefetching can be initialized,

requests for data on different locations on the disk can still incur significant latencies, which are hard to be hidden for an I/O-intensive application. As a result, the disk latency problem can still be a major obstacle that limits I/O performance in the HPC systems.

This research is to provide a storage system that mainly consists of hard disks and is made almost latency-free with the use of a limited amount of SSDs. Instead of caching all recently or frequently accessed data into the SSD, which could consume exceedingly large amount of SSD space and may not be necessary, we organize SSD and hard disk into a storage hierarchy and change the data layout in the storage system. In the hierarchy, the SSD logically stays above the disk. For a segment of data that can be accessed together, we split it into two sections, the head and the body. For the data segment that is originally stored on the disk, we *lift* it *up* so that the head section is placed in the SSD and the body section remains in the disk. Because of this new data layout, the potential long disk latency for accessing the segment can be eliminated as described below: requests for the head and the body are issued simultaneously to the SSD and the disk, respectively, in an asynchronous fashion. The access of the head is served by the SSD with minimal latency. The disk head relocation time, or the latency, for accessing the body can be overlapped with the access time of the head from the SSD. That is, right after the head is served by the SSD, the body is ready to be accessed at the disk and the only latency exposed to the user of the segment is the very small SSD latency.

Figure 1 illustrates the effect of the latency hiding. Without *LiU*, accessing a segment of data exposes a gap of latency at the beginning of request service on the disk (Figure 1(a)), while the ideal scenario is that the disk serves the request at the disk's peak throughput immediately after it receives the request (Figure 1(c)). As shown in Figure 1(b), *LiU* leverages SSD's lower latency and higher throughput to fill the initial time gap when the disk is relocating its head. Otherwise requester cannot receive any data from the disk in this period. In addition to the advantage on latency, *LiU* also has an advantage on I/O throughput which could be potentially much larger than what is illustrated in the figures. This is because a segment can contain data to be accessed in the following multiple requests. By accessing them together as one segment, rather than multiple requests individually served on the disk, *LiU* can further improve the disk's efficiency.

The design of the new data layout scheme for the hybrid storage system, as well as its implementation in a parallel file system (PVFS2) are collectively called *Lift-it-Up (LiU)*. However, to be effective *LiU* has to address a number of challenges in its design.

How to determine an appropriate segment size? There are two requirements on the size. It must be large enough to fully exploit the advantage of the hard disk's sequential throughput. In the meantime, it cannot be exceedingly large, because we have to make sure that the accessed data would be needed by the running programs in the future. An unnecessarily large segment would lower this likelihood.

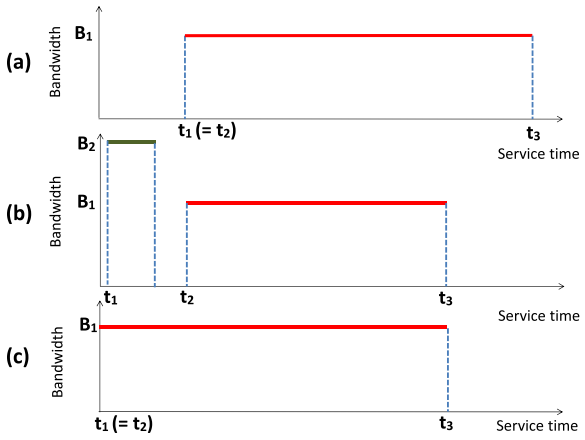


Fig. 1. A segment's access time and real-time throughput during the access when (a) *LiU* is not adopted, (b) *LiU* is adopted with the segment head served at the SSD, and (c) an ideal latency-free disk is employed. In the figures, we assume that the request for the segment is received at time 0, data start to be available at time t_1, the disk starts to make data available at time t_2, and the access is completed at time t_3. As denoted, B_1 and B_2 are the disk's and SSD's peak throughputs, respectively.

How to determine an appropriate head size? While a larger head helps hide the disk latency for accessing the body access, it consumes more SSD space and thus increases the system cost. A head should has a well-calculated size to balance the performance and the cost.

How to minimize the usage of the SSD space? We should selectively *lift* segments *up*, or only segments that are actively used or exhibit strong spatial locality are placed on both the SSD level and the disk level in the hierarchy. In other words, *LiU* needs to monitor data access patterns and dynamically determine segments for *lifting* to maximize performance benefit with minimal use of SSD space.

We summarize our contributions as follows:

- We propose a new storage hierarchy *LiU* for HPC systems which is (almost) latency-free and has high throughput by integrating the advantages of SSD (of very low latency) and hard disk (of high throughput, large capacity, and low cost).
- We design an intelligent data storage policy in the hierarchy, which could use minimal SSD space for the maximal removal of disk latency. This is achieved by adaptively changing data layout in the hierarchy to match workload access pattern.
- We have built a prototype system of *LiU*, which is a non-intrusive implementation as it only changes PVFS2 server-side code. It is highly portable and we do not see any particular challenges to apply it to other parallel file systems.
- We have evaluated *LiU* with representative MPI-IO micro-benchmarks, including *mpi-io-test*, *mpi-tile-io*, and *ior-mpi-io*, and one macro-benchmark, *BTIO*. Our experimental results show that *LiU* improves the I/O performance by up to 5.8 times.

The rest of this paper is organized as follows. Section II

describes the design of *LiU*, followed by the experimental results presented in Section III. Section IV discusses related works, and Section V concludes the paper.

II. The Design of *LiU*

The objective of *LiU* is to eliminate disk latency by re-laying out selected data segments onto disks and SSDs. This latency is actually hidden by the SSD's access time. In this section, we will describe a design that not only has high-performance but also is cost-effective and easy to be implemented. This is achieved by adequately exploiting disk's throughput advantage and carefully using costly SSD space. To this end, there are several issues to be addressed, including determining segment size and its head size, selection of segments for lifting up, and management of segments loaded into the DRAM buffer.

A. Positioning LiU in the Parallel File System

LiU is designed to be part of a parallel file system (e.g., PVFS2) for an optimized data layout and I/O request service. A typical parallel file system has its client-side software on the I/O nodes (or compute nodes where parallel programs run) and its server-side software on the data nodes. Files are striped across multiple data nodes with a fixed striping unit. A metadata sever provides information about on which data node(s) requested data are located. A client usually has to contact the metadata server before issuing an I/O request to the data node. If its requested data are distributed on more than one data node, the client needs to issue multiple sub-requests to the corresponding data nodes. On each data node, the parallel file system relies on the local file system to access data on the disk(s) attached to the node.

One design goal of *LiU* is simplicity. To this end, *LiU* is implemented in the individual data nodes without coordination among them. This choice can be justified by the *LiU*'s ability of providing data access of much reduced latency. The ability allows the client to see predictable and consistent service times across sub-requests, which the client breaks a request into and are issued to different data nodes, without communication between clients and data nodes and coordination between data nodes [23]. Specifically in PVFS2, on each data node there is a server daemon, *pvfs2-server*, responsible for creating I/O jobs (i.e., I/O requests). We instrument its code to add *LiU*'s functional modules for optimized request scheduling and data management.

In the design there are three modules at each data node. They are responsible for re-laying out data and redirecting requests, for buffering segments and measuring their access locality, and for selecting segments for lifting, respectively. Below we describe the design and considerations involved for each of them.

B. Lifting a segment up

As we have mentioned, a selected data segment can have its access latency removed by placing its head and body in the SSD and the disk, respectively, or so-called being lifted up. For an efficient segment management, *LiU* fixes the segment size and uses segment as the basic unit to access any lifted data. To implement this data re-laying out, we need to determine the size of the head and the body and they have different trade-offs.

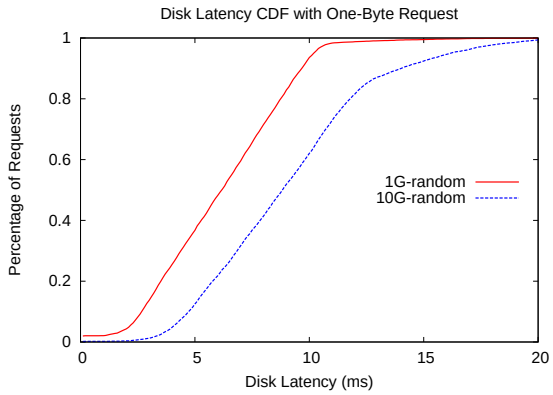


Fig. 2. Cumulative distribution function of disk latency

First, the head's size should be large enough to produce the effect that the disk starts to read or write the requested data at the disk's peak throughput right upon receiving a request. If we denote a request's on-disk latency as t, and the disk's peak throughput as B, then the head size should be at least $B \times t$ for hiding the disk latency. Assuming SSD has a much smaller latency and higher throughput, the SSD would spend a time less than t to finish accessing $B \times t$ amount of data. Unfortunately, the latency t can vary from one request to another widely. Furthermore, a request's latency is usually not predictable because it depends not only on the location of this request but also on where the disk head currently is, which depends on the last request just served by the disk. Because of the I/O request scheduling internal and external to the disk, *LiU* has to assume that the disk head is at a random position in a large range and accordingly the latency is a random number within a certain range. Accordingly, we choose a t that is the 80 percent of all latency values collected during a profiling run of a micro-benchmark, which repeatedly issues one-byte request for data randomly located in a disk region. As an example, Figure 2 shows two latency CDF curves, with requests for data in the 1 GB and 10 GB files, respectively. Assuming a long-term workload characteristics, we can choose an appropriate curve to determine the latency. In our prototyped system we choose the curve with 10 GB file access and determine the t as 12 ms. Additionally, we measured that the disk peak throughput B on our platform is 70 MB/s. Therefore, the head size should be 70 MB/s $\times$ 12 ms = 0.84 MB and we choose 1 MB as the head size for management convenience.

Second, the body size should be large enough to take advantage of the hard disk's peak throughput. As data accessed by one request must be sequential on the disk, a larger request means a more sequential access and a throughput closer to the peak one. However, an exceedingly large body and segment, even if with a fixed head size, has the risk that

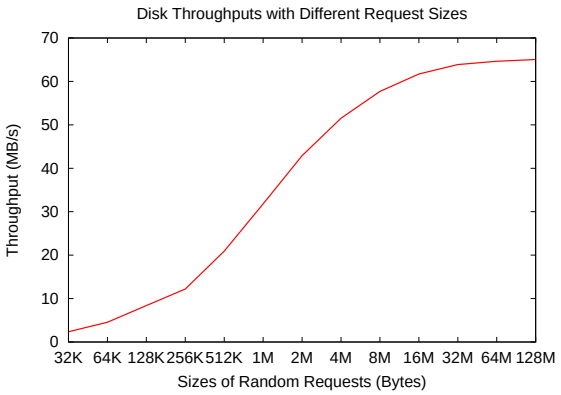


Fig. 3. Disk throughput with different sizes of random requests

only a fraction of the data in a segment are actually requested by user programs before the segment is being evicted (see Section II-D) and the disk bandwidth can be wasted. To make a tradeoff, we need to identify a sufficiently small segment that can enable a throughput close to the peak one. To this end, we ran a micro-benchmark that issues fixed-size read requests with randomized offsets in a 20 GB file in one of our experimental disks. Figure 3 shows the disk throughput with difference request sizes. From the figure we can see that before the request size reaches around 4 MB, the throughput goes up quickly with increasingly large requests. However beyond this point, the trend of throughput improvement diminishes. For this reason we choose the body size of a segment as 4 MB, or the total segment size as 5 MB.

To lift up a segment, we copy its head to the SSD. By copying instead of moving, *LiU* can conveniently change the layout of the segment to its original, or *de-lift* the segment, when the segment is no longer actively used.

C. Re-Shaping I/O Traffic

For any lifted segment of strong access locality, or of a locality value larger than a pre-defined threshold (more details provided later), if a read request asking for its data but the data has not been loaded in the buffer yet, *LiU* holds on the request. And then *LiU* issues an asynchronous I/O request to the disk for the segment's body. Expecting the request to the disk would take a relatively long service time, *LiU* immediately sends a request to the SSD for the head. In this way, SSD can quickly stream the requested data into the buffer while the disk is still moving its disk head to the segment body. After the segment is retrieved into the buffer, *LiU* uses the data in the buffer to serve the read request.

With this data access schedule, a segment can be efficiently loaded in the buffer in an almost latency-free manner. If the initial read request that triggers the segment access asks for all or a major part of data in the segment, the request's latency is largely eliminated. Otherwise, if there are many requests, each for a portion of data contained in the segment, arriving later than the initial request or possibly after the segment is buffered, their latencies are also reduced or eliminated. Because *LiU* requires existence of a strong locality to conduct the segment-based access, the above scenarios are highly likely to occur and the average latency of the involved requests can be significantly reduced. However, there still can be a few of high-latency requests, including the aforementioned initial request, as the access of the segment can take a relatively long time. To address the issue, *LiU* adopts two optimizations to reduce latency for requests arriving before the entire segment is in the buffer. First, if a request is only for data in the head and the data is not being accessed by another request, *LiU* immediately issues this request to the SSD. Second, instead of sending one large request to the disk for the body, *LiU* evenly divides the body into a number of sections, which is four by default in its prototype, and sends multiple smaller asynchronous requests, each for one section, to the disk. Without having to wait for the entire body to be ready in the buffer, a process can obtain its data as long as the section(s) containing the data are loaded.

For a write request, *LiU* relies on the system's write-back policy to buffer dirty data and initiate batched writes to the disk for a high I/O throughput. For any written data that is located in a lifted segment, *LiU* re-directs them to the SSD. For a segment that is being de-lifted, *LiU* flushes its head in SSD back to disk, if it's not changed, then de-lifting operation is to simply delete the head in SSD. To avoid any consistency issue, *LiU* flushes the system buffer before it starts to lift up a new segment.

D. Buffering Segments and Measuring Access Locality

Although SSD stores relatively small share of data in the hybrid storage managed by *LiU* (20% of lifted segments in our prototype), the demand on its capacity can still be substantial if it indiscriminately lifts every segment on the disk. To address the issue, *LiU* lifts only selected segments. There is one important factor involved in the selection. The access of the segment must have strong locality. That is to say, when a segment is loaded into the buffer, all or most of its data must be actually requested by processes before the data leave the buffer.

LiU maintains a buffer for storing recently loaded segments. The buffer has two purposes. One is to provide the caching space for segment data. This is especially important in a parallel execution environment, as requests for the data in a segment can be issued by different processes of an MPI program and accordingly arrive at different times. For an efficient use of the buffer, *LiU* applies the LRU algorithm to decide the victim segment for replacement if no space is available for a newly loaded one. The other is to measure the spatial locality, which is quantified in this paper as the percentage of a segment's data that has been requested by users' processes after the segment is loaded into the buffer and before it is evicted out of the buffer. As segment is the unit for *LiU* to access the lifted data in the storage, a weak locality means wasting I/O bandwidth as well as buffer space. To measure the locality, *LiU* flags the data in a segment when they

are requested and calculate the segment's locality, a percentage value, when the segment is replaced.

Apparently a segment of weak locality should not be loaded into the buffer at all, even if it has been lifted. However, such segment also needs to have its locality evaluated because its locality could rise and accordingly may need to be loaded in its next access. To this end, *LiU* monitors access of un-loaded segments, including un-lifted segments and lifted segments with sufficiently weak locality. When data in such a segment is requested, *LiU* treats it as a ghost segment (with only metadata created) and manages it in the same way as regular (i.e., buffered) segments in the LRU replacement. In the LRU algorithm for managing the segment buffer, an LRU stack, named as the buffer stack, is maintained. A regular segment at the bottom of the buffer stack is replaced. With the ghost segments in the stack (possibly interleaved with the regular segments in the stack), the stack size can be larger than the buffer size in terms of segment count. After a regular segment is evicted, *LiU* continuously removes ghost segments if they are at the bottom of the buffer stack. This is because ghost segments that are even less active (or less recently used) than the least recently used (LRU) regular segment are of less interest. At the time of the removal, the locality value of a ghost segment is calculated and will be used in the *LiU*'s module for determining which segments should be lifted as well as be loaded into the buffer in their next accesses.

III. Performance Evaluation and Analysis

We have implemented a prototype system of *LiU* on PVFS2 [7] (version 2.8.2), a production parallel file system. To evaluate *LiU*, we have conducted the experiments on a cluster, which consists of seven compute nodes and six data nodes. Each compute node has a 1.6 GHz dual-core Pentium processor, 1 GB DRAM, and an 80 GB SATA hard drive. Each data node is equipped with a 2.13 GHz Intel Core-2 Duo processor, 1 GB DRAM, and two 160 GB Western Digital hard drives (Model#: WD1602ABKS-18N8A0) and one 32 GB OCZ SSD (Model#: OCZ-ONYX). The two disks on data nodes are used for OS and PVFS2 storage space, respectively. While the DRAM memory is of relatively small size, it is not a concern in the evaluation as 256MB memory is sufficient for being used as LiU's buffer (see Section III-E) and the benchmarks are not memory-intensive. Compute nodes and data nodes are connected via one Gigabit Ethernet Switch. Both compute nodes and data nodes are installed with CentOS of the Linux kernel version 2.6.18. Additionally, we installed MPICH2-1.4.1p1 on compute nodes and PVFS2-2.8.2 on compute nodes and data nodes. We used the default unit size of PVFS2, i.e., 64 KB, to stripe file data over six data nodes.

A. The Benchmarks

We evaluated *LiU* with three micro-benchmarks (i.e., *mpi-io-test*, *ior-mpi-io*, and *mpi-tile-io*), and one macro-benchmark *BTIO*. These MPI-IO benchmarks cover a wide spectrum of parallel I/O access patterns (at each data node), varying

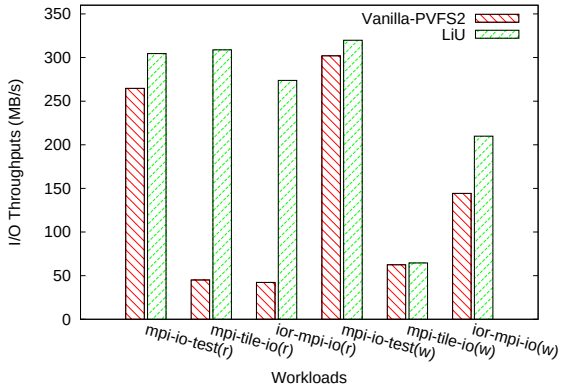


Fig. 4. I/O throughput for different MPI-IO benchmark programs. Each benchmark is configured as both read and write.

from sequential accesses (e.g., *mpi-io-test*) to non-sequential accesses (e.g., *ior-mpi-io* and *BTIO*), from contiguous accesses (e.g., *mpi-io-test*) to non-contiguous accesses (e.g., *mpi-tile-io*), from read accesses to write accesses, and from requests that are aligned with 64 KB striping units (e.g., *mpi-io-test* and *ior-mpi-io*) to requests with different sizes (e.g., *mpi-tile-io* and *BTIO*). Next we will briefly introduce each benchmark.

Mpi-io-test, included in PVFS2 software package, is an MPI-IO benchmark jointly developed by Clemson University and Argonne National Laboratory [4]. In this benchmark, each MPI process accesses a contiguous chunk of the file in one iteration. A barrier is used to synchronize each iteration across all processes. The following iteration of all the processes accesses the next contiguous chunk of the file. In our experiments, the I/O request size from each process is 64 KB.

Mpi-tile-io is from Parallel I/O Benchmarking Consortium [6]. This benchmark logically views a data file as a two dimensional sets of tiles and two adjacent tiles can be configured as partially overlapped in both dimensions. Each MPI process is responsible for accessing one tile via non-contiguous file accesses. In our experiments, there are 14 tiles, 64 byte per element, and 16 elements shared between adjacent tiles in the X and Y dimensions.

Ior-mpi-io is a program in the ASCI Purple Benchmark Suite developed at Lawrence Livermore National Laboratory [3]. In this benchmark, each of the n processes accesses a contiguous $1/n$ portion of the entire file. Each process continuously issues sequential I/O requests with a fixed size. While the requests within a process is sequential, in each data node, the requests from different MPI processes are non-sequential since they belong to different parts of the data file. In our experiments, the I/O request size from each process is 64 KB.

BTIO is an I/O-intensive Fortran program from NAS parallel benchmarks [5]. It is implemented as a solver of 3D compressible Navier-Stokes equation. The program is compiled with ROMIO/MPIIO library for its on-disk data access. After computation in each time step, *BTIO* writes a large amount of data to storage using small non-contiguous I/O requests.

When computation is completed, all the data are read back for result sanity tests.

For all of the above benchmarks, the file accesses can be set as either read or write. In our experiments, we ran 14 processes on compute nodes, two on each node, and used a 10 GB data file to accommodate all benchmarks' read needs. To avoid the effect of the page cache in OS kernel, write experiments had a relatively large data set, compared to read experiments. More details about the read/write data size for each benchmark are presented in the Section III-B.

B. Micro-benchmark Results

We first evaluate *LiU* with each MPI micro-benchmark by running one instance alone, i.e., without interference. In particular, we measured the I/O throughput when running one instance of the MPI benchmark program with the original PVFS2 (the baseline) and with *LiU* (the enhanced PVFS2), respectively. Figure 4 shows the results for all the benchmarks. In this set of experiments, each benchmark was configured both read and write.

For read experiments, as shown in Figure 4, *LiU* improves the read throughput of the file system for all the benchmarks by 15% to 5.8 times, with an average of 3.8 times. This indicates that *LiU* is effective in improving the I/O throughput of the file system by hiding disk latency. Additionally, this experimental result reveals that *LiU* has different performance gains for different file access patterns. To better understand the reasons, we further collect the LBNs (logical block numbers) accessed on the disk and latency of serving each request for two benchmarks with different I/O access patterns (one random and one sequential access patterns).

For the *ior-mpi-io* benchmark, *LiU* improves the I/O throughput by 5.5 times for read (the total input size is 2.2GB). The data access pattern for this benchmark is that each process accesses a continuous chunk of file. Therefore, the disk head moves among these chunks when multiple requests arrive from different MPI processes. Figure 5 shows the order of LBNs accessed with the original PVFS2 and with *LiU*, respectively. Under the original PVFS2, the disk serves the requests for 14 different contiguous chunks of the data file and the disk head heavily alternates among these chunks (shown in Figure 5 (a)). On the contrary, *LiU* clusters the disk accesses within one contiguous chunk and has fewer requests issued to the disk. This is mainly because *LiU* accesses disks at the granularity of segment (5 MB in our prototype).

Furthermore, for the *ior-mpi-io* benchmark, we investigate the latency of servicing each request (i.e., the difference between the times when a request arrives and when the request is responded within PVFS2 or WM on a data node). Figure 6 shows the latency of each request with the original PVFS2 and with *LiU*, respectively. As shown in Figure 6 (a), the latencies of all requests in the original PVFS2 vary from near zero to 250 milliseconds, and most latencies are at around 100 milliseconds. After applying *LiU*, the latencies of all the requests drastically reduced, i.e., most requests have near zero and some at around 15 milliseconds. This indicates that *LiU* can effectively hide disk latency.

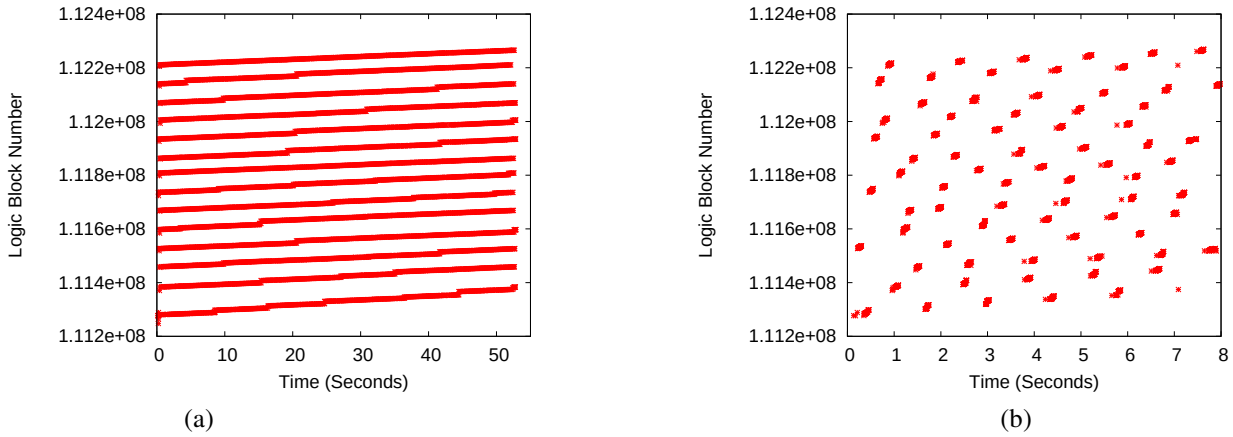
For the *mpi-io-test* benchmark, *LiU* improves the I/O throughput by 15% in read (the total input size is 5.1G). As mentioned in Section III-A, different MPI processes in this benchmark simultaneously access a contiguous region of the data file at each iteration and two consecutive iterations access two consecutive regions. This essentially is a sequential I/O access pattern. So the accessed LBNs are sequentially increasing with both the original PVFS2 and *LiU* (due to space limitation, we didn't show the LBN figure here). Therefore, the average disk seek distance is expected equally small for both PVFS2 and *LiU* with such workload. This indicates that *LiU* brings less significant performance gains on sequential I/O accesses than that on random accesses.

For write performance, *LiU* has improvements of 6% and 3% for *mpi-io-test* and *mpi-tile-io*, respectively (the total write size is 20G for *mpi-io-test* and 7.6G *mpi-tile-io*). Note that *LiU* only writes 80% data into the disk and writes 20% data into the SSD simultaneously, so the theoretical improvement is 25% if we only consider the amount of data written to disks. However, some segments that are flushed to disk may be brought back to the buffer for further writing since the large segment (5 MB in our prototype) is not fully modified. As a result, the write performance improvement in *LiU* cannot achieve the theoretical value. In *ior-mpi-io* benchmark, the write performance improvement is 45% which is surprisingly larger (the total write size is 20G). This is mainly because of its unique access pattern. Figure 7 shows the disk access addresses during the execution time from 62.5 to 62.7 seconds, when running *ior-mpi-io* with the original PVFS2 and *LiU*, respectively. Without *LiU*, the 14 processes simultaneously access their individual data. We can see from Figure 7 (a) that the disk head moves back and forth. The root cause is the limited queue size of the OS disk scheduler [24], resulting in that only part of these requests can be sorted and merged for spatial locality. In contrast, from Figure 7 (b), we can observe that LBN addresses almost grow in one direction while running in *LiU*. This indicates that *LiU* helps minimize the disk head movement.

C. Macro-benchmark Results

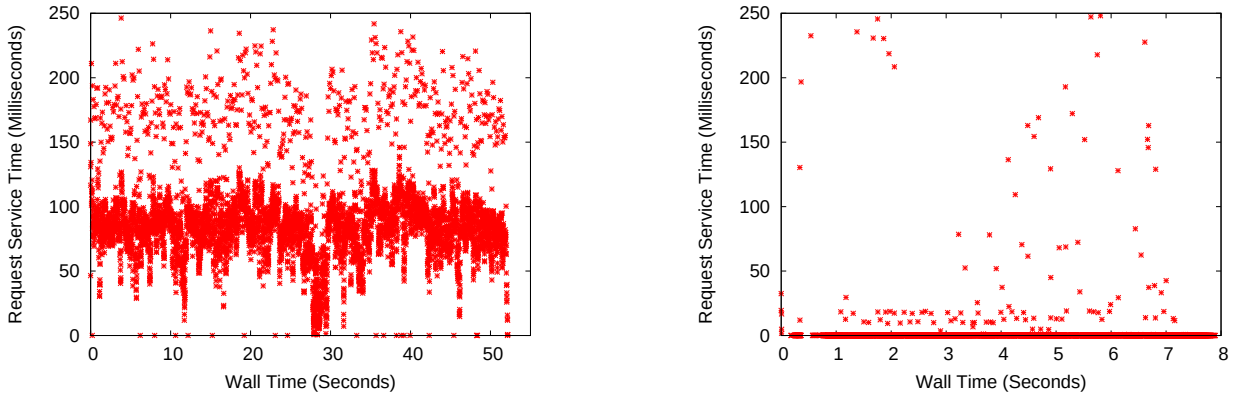
In this section we evaluated *LiU* with the *BTIO* macro-benchmark, which has interleaved computation and I/Os during its execution. We ran the program using 8, 16, and 25 processes, with problem sizes coded as both B and C in the benchmark, which generated a data set of 1.7 GB and 4.9 GB, respectively, using non-collective I/O operations. Most I/O requests are small non-contiguous writes. As shown in Figure 8 with *LiU* I/O throughput for B and C increased by 31% and 12%, respectively, on average. The decreased performance advantage is because the *LiU* buffer size did not change as problem size was increased from B to C, which could result in more misses on the buffer accordingly.

From Figure 8 we can also observe that under problem size B the improvement ratio of I/O throughput became larger



(a) (b)

Fig. 5. Disk addresses (LBNs) of data accesses on disks of data node 1. (a) Running *ior-mpi-io* with the original PVFS2, and (b) running *ior-mpi-io* with *LiU*.



(a) (b)

Fig. 6. Latency of each request on data node 1. (a) Running *ior-mpi-io* with the original PVFS2, and (b) running *ior-mpi-io* with *LiU*.

as process count increased from 9 to 25 with *LiU*. More specifically, when 25 processes were used, the I/O throughput increased by 42%, compared with 31% when 16 processes were used and 20% when 9 processes were used. The reason is that in the same problem size I/O request size is accordingly reduced when process count becomes larger. Therefore, without *LiU* the disk has to perform more frequent seeks when serving random requests of smaller size. In comparison, with the help of *LiU* as more such requests are completed in the *LiU* buffer, rather than on the disks, disk seeks are significantly reduced. We also present the total execution time of *BTIO* benchmark in Table I, as the number of processes increased. Compared with the vanilla system, *LiU* helps PVFS2 file system become more scalable in seamlessly combining disk, SSD, and memory buffer.

# of Processes	9	16	25
Time w/o *LiU*(s)	588	613	523
Time w/ *LiU*(s)	585	579	501

TABLE I
EXECUTION TIMES OF *BTIO* BENCHMARK AS PROCESS CONCURRENCY INCREASED FROM 9, 16 TO 25 WITH C PROBLEM SIZE. BOTH RESULTS WITHOUT AND WITH *LiU* ARE SHOWN, RESPECTIVELY.

D. Sequential Workload with Interferences

While *LiU* provides moderate performance improvement for running one instance of sequential-access benchmarks alone, will it be helpful if the sequential workload is interfered by other workloads? This section answers the question with two sets of experiments. In the first set of experiments, we ran a sequential-access workload *mpi-io-test*, together with another sequential-access workload, i.e., the second instance of the same benchmark *mpi-io-test*. In the second set of experiments, we ran a sequential-access workload *mpi-io-test*, together with a random-access workload *ior-mpi-io*. In both sets, we ran two instances of the same or different benchmarks with 14 processes and both instances access different data files. Additionally, in both sets, we measured the throughput of the first sequential-access workload *mpi-io-test* with the original PVFS2 and with *LiU*, respectively. Note that the interfering workload would not terminate until the target workload *mpi-io-test* finishes the job.

Table II summarizes the experimental results. Compared

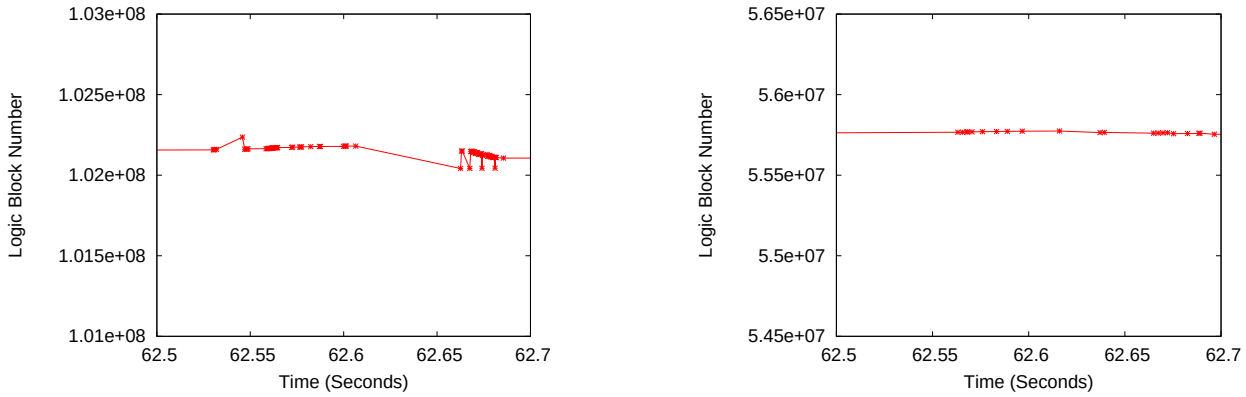


Fig. 7. Disk addresses (LBNs) of write accesses in the time period from 62.5 second to 62.7 second on the disk of data node 1. (a) Running *ior-mpi-io* write with the original PVFS2, and (b) running *ior-mpi-io* write with *LiU*

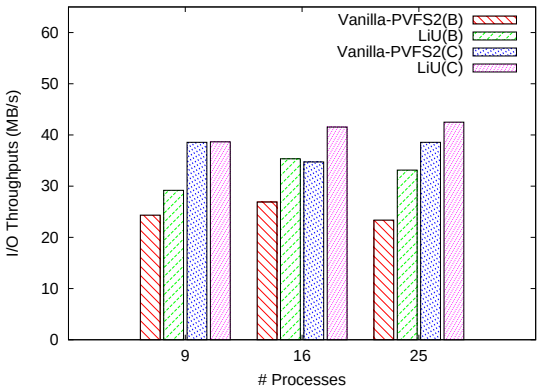


Fig. 8. I/O throughput of the *BTIO* benchmark when they are executed without *LiU* (Vanilla-PVFS2) or with *LiU*, and with different number of processes and problem sizes.

mpi-io-test Interfered?	Throughput (MB/s)		
	w/o *LiU*	w/ *LiU*	Ratio
None	264.7	304.5	1.15 X
by *mpi-io-test*	124.8	261.6	2.10 X
by *ior-mpi-test*	47.6	164.1	3.45 X

TABLE II
PERFORMANCE OF *mpi-io-test* THAT IS WITHOUT INTERFERENCE, INTERFERED BY A SEQUENTIAL-ACCESS WORKLOAD, AND INTERFERED BY A RANDOM-ACCESS WORKLOAD, RESPECTIVELY.

with the original PVFS2, *LiU* improves the I/O performance for *mpi-io-test* by 15%, 1.1 times, and 2.45 times, in the configurations of no interferences, being interfered by a sequential-access workload, and being interfered by a random-access workload, respectively. This indicates that a sequential workload can benefit a lot more from *LiU* when it is interfered by a concurrently-running workload. The benefit becomes more significant when the interference is more severe.

We further collected the average disk seek distance in terms of LBN difference and average latency of I/O requests

mpi-io-test Interfered?	Seek Distance (millions)		Latency (millisec)	
	w/o *LiU*	w/ *LiU*	w/o *LiU*	w/ *LiU*
None	0.4	0.2	4.4	1.0
by *mpi-io-test*	11.6	0.5	9.7	1.4
by *ior-mpi-io*	15.1	2.5	42.1	4.9

TABLE III
TOTAL DISK SEEK DISTANCE AND AVERAGE LATENCY OF *mpi-io-test* THAT IS WITHOUT INTERFERENCE, INTERFERED BY SEQUENTIAL-ACCESS WORKLOAD, AND INTERFERED BY A RANDOM-ACCESS WORKLOAD, RESPECTIVELY.

issued by *mpi-io-test* in the experiments under the three configurations, i.e., no interference, interfered by two different types of workloads. Table III shows the results. With the moderate interference introduced by a sequential workload *mpi-io-test*, the average seek distance and latency of the sequential workload under the original PVFS2 become worse, increased by 28 times and 1 time, respectively. However, with severe interferences introduced by a random workload, the average seek distance and latency of the sequential workload become much worse, increasing by 37 times and 8.5 times, respectively. In contrast, with the help of *LiU*, the degradation of the average seek distance and the average latency of the sequential workload is modest, i.e., 1.5-11 times, and 40%-3.9 times, respectively. This is because *LiU* exploits SSD to serve the head and load the entire segment at a time, which effectively hides the disk latency and enjoys the disk peak performance.

E. Impact of Buffer Sizes

We further conducted another set of experiments on the impact of buffer sizes. In the experiments, we measured the I/O throughput of the benchmarks with the original PVFS2 and with *LiU*, respectively, under the varying sizes of the buffer. Figure 9 shows the results using two different types of benchmarks, i.e., *mpi-io-test* with sequential-access patterns and *ior-mpi-test* with random access patterns. For the sequential workload *mpi-io-test*, the I/O throughput improvement provided by

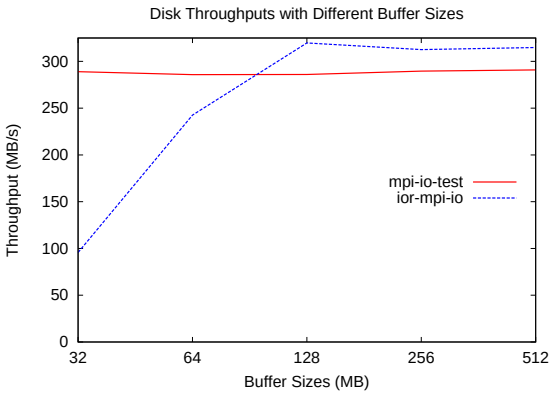


Fig. 9. Performance impact of the buffer size on *LiU*.

LiU is not sensitive to the buffer size. For example, *LiU* with a 32 MB buffer can achieve similar I/O throughput as that with a 512 MB buffer for *mpi-io-test*. This is because *mpi-io-test* accesses the data sequentially, thus has strong space locality and weak temporal locality. On the contrary, the random workload *ior-mpi-io* running on top of *LiU* is more sensitive to the buffer size. For example, *LiU* with a 32 MB buffer can only achieve one third of the I/O throughput with a 128 MB buffer. When the buffer size contains the working set of the workload, the further improvement via *LiU* is minimal. Considering various types of workloads and the burden on the system, we choose a reasonable buffer size, i.e., 256 MB, in our prototype system.

IV. Related Works

LiU uses SSD's very low latency to help remove hard-disk's high latency and accelerate I/O-intensive HPC applications that have their data on the hybrid storage hierarchy. *LiU* is most effective with requests issued by different processes and exhibiting strong spatial access locality. In this section we discuss prior works in the literature primarily on hiding disk latency and on incorporation of SSD for improving disk-based storage systems.

A. Reducing Disk Latency

Disk latency has a major impact on the I/O throughput of workloads accessing non-contiguous data on the disk. Many techniques have been proposed in each layer of the I/O stack to hide the latency.

Asynchronous I/O [18] allows computing processes to proceed beyond an unfinished I/O call. If there is no dependence between data requested in the I/O call and the execution following it, I/O time could be hidden behind the computation time. For data-intensive scientific applications, researchers developed in-situ [10] execution or DataStager [9], where the input/output data can be asynchronously loaded/unloaded into/out of compute nodes or a staging area. Applying data asynchronous access usually requires programmers' or users' involvements.

Furthermore, another widely used technique for hiding disk latency is I/O prefetching [16], [11], [22], which predicts data to be accessed according to prior I/O activities and fetches them into memory buffer cache before they are actually requested. While prediction accuracy is critical to the effectiveness of the prefetching, Chang et al. [13], [17] proposed to generate I/O prefetching hints for higher accuracy using speculative execution of programs. This technique relies on the support of operating system on execution roll-back because speculation may prove to be wrong. For parallel program executions, Chen et al. [15] developed a pre-execution-based prefetching scheme to hide I/O latency. More recently, Zhang et al. [25] proposed a data-driven execution mode to make prefetch requests more efficiently served on the disk.

Compared with *LiU*, in addition to their requirements on manual efforts, accurate prediction, or complicated system engineering, a critical disadvantage of these approaches is their reliance on process's computation for hiding I/O latency. In the context of I/O intensive computing, the I/O time can well dominate the entire program execution time and there could be relatively little computation time for hiding the latency, which makes them less effective. In contrast, *LiU* uses SSD's I/O time to hide the disk latency. Furthermore, it does not require any supports from the programmers, libraries, and compiler and does not change process execution method. This makes *LiU* a latency-reduction technique more effective in the I/O-intensive computing and easier to implement and deploy.

B. Using SSD in Storage Systems

Because of SSD's clear performance advantage over hard disks, SSD-based storage systems are widely applied in both enterprise and high-performance computing. In these applications, SSD provides either a caching space or a storage space.

Being conscious of its relatively high cost and small capacity, SSD is usually used as a cache of data stored on the hard disks. Srinivasan et al. designed Flashcache [21], which is a block-level cache between DRAM and hard disks and is managed by using a cache replacement algorithm such as FIFO or LRU. Liu et al. [19] used SSD as a cache to support virtual memory by allowing actively used memory pages to be swapped to the SSD. iTransformer [24] extends the disk scheduling queue on the SSD, so that requested data can be buffered in the queue on the SSD for more aggressive scheduling to recover spatial locality, which could have been lost with a high I/O concurrency. Recently, a scheme, iBridge [26], was designed to use SSD for caching and serving unaligned data in a cluster of data nodes to improve disk efficiency. In contrast, *LiU* is not a conventional scheme for caching any actively used data in the SSD, which could quickly use up the limited SSD space and incur frequent data replacements in the SSD. It stores only the head of a segment, even when the entire segment is actively requested. This makes the SSD space more efficiently used and the on-disk data more efficiently accessed.

If the SSD is used as a storage device, efforts are usually made on selecting only performance-critical data to store on it. Chen et al. [14] proposed a hybrid storage system, Hystor,

which is comprised of both SSD and hard disk. It selects actively used metadata and small files to be stored on the SSD. I-CASH [20] is another hybrid storage framework based on data-delta pairs to store only changes of file blocks on the SSD by exploiting the spatial locality of data access. Fusion Drive developed at Apple. Inc is another example for using SSD in a hybrid storage [2]. It stores the most frequently accessed files on the faster flash storage and leaves less used files on the hard disk. However, there are complaints that some very frequently accessed large files are not stored on the flash drive [8]. Understandably if a decision is about whether an entire file is moved to the flash or not, a compromise has to be made to leave large files on the disk even if they are actively used due to constraint on the SSD space. *LiU* overcomes the difficulty by placing only necessary data (the heads) on the SSD. In this way, accessing of large files can be accelerated without fully caching them. To remove I/O bottleneck in data-intensive scientific computing, a few systems use SSD as the only device in their storage subsystems, e.g. Gordon [12]. Though *LiU* also identifies actively used data for accelerated access, it does not simply move them to the SSD. Instead, it lifts them up to make a balanced use of the disk and the SSD, so advantages of both can be well exploited.

V. Conclusions

In this paper, we propose *LiU*, a hybrid and almost latency-free storage system for accelerating I/O-intensive HPC applications. It achieves this goal by leveraging limited amount of SSD storage for its low-latency access and hard disk of large capacity for its high peak throughput. Based on the above ideas, We implement a prototype system of *LiU* in PVFS2. Our evaluation of *LiU* with representative MPI-IO micro-benchmarks, including *mpi-io-test*, *mpi-tile-io* and *ior-mpi-io*, and one macro-benchmark *BTIO* show that *LiU* can effectively improve the I/O performance for HPC applications, improving the throughput up to 5.8 times. Furthermore, *LiU* can bring much more benefits to sequential-I/O MPI applications when the applications are interfered by other workloads. For example, *LiU* improves the I/O throughput of *mpi-io-test* that is under interference by 1.1-2.5 times, while improving the same workload without interference by 15%.

VI. Acknowledgements

The authors would like to thank the anonymous reviewers for their helpful feedback. This research is partially supported by NSF grants CCF#0845711 (CAREER), CNS#1117772, CNS#1217948, CCF#0953759 (CAREER) and CCF#1218358.

References

[1] Flash disks: Enabling mission-critical systems. http://www.cotsjournalonline.com/articles/view/100052.

[2] Fusion drive. http://en.wikipedia.org/wiki/Fusion_Drive.

[3] IOR: Parallel filesystem I/O benchmark. https://github.com/chaos/ior.

[4] mpi-io-test source code. http://mirror.anl.gov/pvfs2/tests/mpi-io-test.c.

[5] NAS Parallel Benchmarks, NASA Ames Research Center. http://www.nas.nasa.gov/Software/NPB.

[6] Parallel I/O Benchmarking Consortium. http://www.mcs.anl.gov/research/projects/pio-benchmark.

[7] Parallel Virtual File System, Version 2. http://www.pvfs.org/.

[8] Update on apple 'fusion': Writes are fast, no smart migration. http://macperformanceguide.com/blog/2012/20121108_3-Fusion-MacMini.html.

[9] Hasan Abbasi, Matthew Wolf, Greg Eisenhauer, Scott Klasky, Karsten Schwan, and Fang Zheng. Datastager: Scalable data staging services for petascale applications. In *International ACM Symposium on High Performance Distributed Computing*, HPDC'09, 2009.

[10] Janine C. Bennett, Hasan Abbasi, Peer-Timo Bremer, Ray Grout, Attila Gyulassy, Tong Jin, Scott Klasky, Hemanth Kolla, Manish Parashar, Valerio Pascucci, Philippe Pebay, David Thompson, Hongfeng Yu, Fan Zhang, and Jacqueline Chen. Combining in-situ and in-transit processing to enable extreme-scale scientific analysis. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC'12, 2012.

[11] Surendra Byna, Yong Chen, Xian-He Sun, Rajeev Thakur, and William Gropp. Parallel I/O prefetching using MPI file caching and I/O signatures. In *High Performance Computing, Networking, Storage and Analysis*, SC'08, 2008.

[12] Adrian M. Caulfield, Laura M. Grupp, and Steven Swanson. Gordon: Using flash memory to build fast, power-efficient clusters for data-intensive applications. In *the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'09, 2009.

[13] F. Chang. Using speculative execution to automatically hide I/O latency. In *Carnegie Mellow Ph.D Dissertation*, CMU-CS-01-172, 2001.

[14] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Hystor: Making the best use of solid state drives in high performance storage systems. In *Proceedings of the international conference on Supercomputing*, ICS '11, 2011.

[15] Yong Chen, Surendra Byna, Xian-He Sun, Rajeev Thakur, and William Gropp. Hiding I/O latency with pre-execution prefetching for parallel applications. 2008.

[16] Xiaoning Ding, Song Jiang, Feng Chen, Kei Davis, and Xiaodong Zhang. DiskSeen: Exploiting disk layout and access history to enhance I/O prefetch. In *Proceedings of the 2007 USENIX Annual Technical Conference*, USENIX'07, 2007.

[17] K. Fraser and F. Chang. Operating system I/O speculation: How two invocations are faster than one. In *Proceedings of the 2003 USENIX Annual Technical Conference*, USENIX'03, 2003.

[18] M. Tim Jones. Boost application performance using asynchronous I/O. http://www.ibm.com/developerworks/linux/library/l-async/?ca=dgr-lnxw02aUsingPOISIXAIOAPI.

[19] Ke Liu, Xuechen Zhang, Kei Davis, and Song Jiang. Synergistic coupling of SSD and hard disk for QoS-aware virtual memory. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS'13, 2013.

[20] 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.

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

[22] Fengguang Wu, Hongsheng Xi, Jun Li, and Nanhai Zou. Linux readahead: less tricks for more. In *Proceedings of the Linux Symposium*, volume 2, 2007.

[23] Xuechen Zhang, Kei Davis, and Song Jiang. IOrchestrator: Improving the performance of multi-node I/O systems via inter-server coordination. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'10, 2010.

[24] Xuechen Zhang, Kei Davis, and Song Jiang. iTransformer: Using SSD to improve disk scheduling for high-performance I/O. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, IPDPS'12, 2012.

[25] Xuechen Zhang, Kei Davis, and Song Jiang. Opportunistic data-driven execution of parallel programs for efficient I/O service. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, IPDPS'12, 2012.

[26] Xuechen Zhang, Ke Liu, Kei Davis, and Song Jiang. iBridge: Improving unaligned parallel file access with solid-state drives. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, IPDPS'13, 2013.