iBridge: Improving Unaligned Parallel File Access with Solid-State Drives

Xuechen Zhang[†‡*] Ke Liu[†] Kei Davis[♮] Song Jiang[†]

[†]ECE Department
Wayne State University
Detroit, MI, 48202, USA

[‡] School of Computer Science
Georgia Institute of Technology
Atlanta, GA, 30332, USA

[♮] CCS Division
Los Alamos National Laboratory
Los Alamos, NM 87545, USA

***Abstract*—When files are striped in a parallel I/O system, requests to the files are decomposed into a number of sub-requests that are distributed over multiple servers. If a request is not 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 large ones, the system exhibits heterogeneity in serving sub-requests of different sizes, and the net throughput of the entire system can be severely degraded by the inefficiency of serving the smaller requests, or *fragments*. Because a request is not considered complete until its slowest sub-request is, the penalty is yet greater for synchronous requests. To make the situation even 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.**

In this paper we propose *iBridge*, a scheme that uses solid-state drives to serve request fragments and thereby bridge the performance gap between serving fragments and serving large sub-requests. We have implemented *iBridge* in the PVFS file system. Our experimental results with representative MPI-IO benchmarks show that iBridge can significantly improve the I/O throughput of storage systems, especially for large requests with fragments.

***Keywords*-Solid state drive, parallel file systems, parallel I/O.**

I. Introduction

To meet the demand for high-throughput data access on storage systems by highly parallel scientific and engineering applications, parallel file systems such as GPFS [11], Lustre [16], and PVFS2 [22] have been widely adopted to manage large data files such as checkpoint/restart files and the inputs and outputs of data-intensive applications. In these file systems the files are striped over multiple data servers to take advantage of aggregate I/O capacity, such as network bandwidth and hard disk bandwidth, while programmers are presented with a convenient linear logical file address space.

With file striping a request for a segment of logically contiguous file space is divided into sub-requests that are distributed over multiple data servers. While the striping unit size is usually reasonably large, such as 64KB by default in PVFS2, so that sub-requests are sufficiently large to obtain high disk efficiency at each server, the first and/or last sub-requests can be much smaller than the striping unit if the request pattern does not match the striping pattern, i.e., data access is unaligned. These smaller sub-requests, which we call *fragments*, are effectively random accesses on their respective servers. Because a hard disk can be an order of magnitude or more less efficient in serving random requests than for sequential ones, fragments and other sub-requests of the same request can be served with very different efficiency. In the case of synchronous requests the entire storage system's productivity will be degraded.

Apps	Unaligned (%)	Random (%)	Total (%)
ALEGRA-2744	35.2	7.3	42.5
ALEGRA-5832	35.7	6.9	42.6
CTH	24.3	30.1	54.4
S3D	62.8	5.8	68.6

Table I
PERCENTAGES OF UNALIGNED AND RANDOM DATA ACCESSES IN DIFFERENT I/O TRACES WITH A 64KB STRIPING UNIT. UNALIGNED REFERS TO REQUESTS THAT ARE LARGER THAN A STRIPING UNIT (64KB) BUT ARE NOT ALIGNED TO THE STRIPING UNIT BOUNDARIES. REQUESTS SMALLER THAN 20KB ARE CATEGORIZED AS RANDOM.

By analyzing I/O traces from various computing environments we have determined that unaligned access is common. One example is the set of traces of HPC applications from the Scalable I/O project at Sandia National Laboratories [13] including applications ALEGRA [1], CTH [7], and S3D [25]. As shown in Table I, for these applications up to 62.8% of I/O requests, and 35.9% on average, are unaligned with the striping pattern on the data servers if we assume a 64KB striping unit. Programmers may not be aware of the unaligned data access because it also depends on system configuration details that determine how files are striped. Another reason unaligned access is pervasive is because of displacements due to, for example, file formats with small headers such as superblock and data object headers in HDF5 files [3]. Table I also shows percentages of random requests in the traces. Here we consider requests that are smaller than 20KB to be random because they are less likely to concatenate into large sequential accesses.

[*]Xuechen Zhang conducted this work while at Wayne State University.

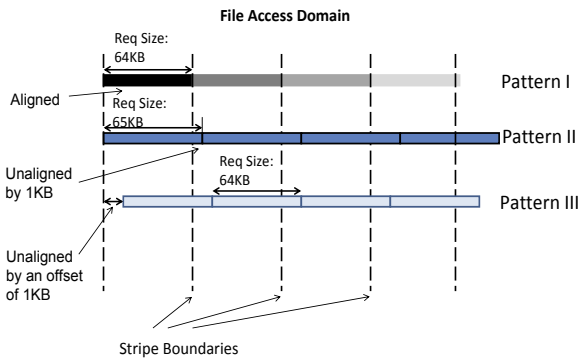


Figure 1. File access with three different alignment patterns.

A. *Effects of Unaligned Access on Performance*

To experimentally investigate the effects of unaligned access on storage system performance we ran the *mpi-io-test* benchmark [22] in which N processes iteratively read data from a 10GB file striped over eight data servers. (Details of the system's configuration are given in Section III.) All read requests are of the same size s, which is configurable. At the kth iteration Process i, $0 \leq i < N$, reads one segment of data at file offset $k*N*s+i*s$ using the PVFS2 parallel file system with a 64KB striping unit. We experimented with three access methods with differing alignment patterns on the data servers as illustrated in Figure 1. In the first, Pattern I, the size of the requests is the same as the striping unit size and they are exactly aligned. In Pattern II the request size differs from the striping unit size, here by 1KB. In the logical file address space the processes still issue sequential I/O requests but fragments are generated by the parallel file system at each data server. In Pattern III the request size is equal to the striping unit size but the requests are shifted by an offset of 1KB relative to the striping boundaries, requiring each 64KB request to be served by two data servers. In these experiments a barrier operation is not applied between access iterations.

Figure 2(a) shows how the unaligned access of Pattern II degrades system throughput relative to Pattern I. For Pattern II we varied request size among 64KB, 65KB, 74KB, 84KB, and 94KB with process count ranging from 16 to 512. Note that Pattern II becomes Pattern I for request size 64KB. Unaligned data access results in throughputs that are consistently lower than their respective aligned counterparts. For example, with 16 processes and unaligned requests that are larger than the striping unit (64KB) by 1KB and 10KB, the throughputs with Pattern II are reduced by 52% (from 159.6MB/s to 77.4MB/s) and 45% (from 159.6MB/s to 88.1MB/s), respectively, relative to Pattern I. Throughput is also reduced as process count increases, for example with 64KB requests (aligned access) the throughputs are 159.6MB/s and 116.2MB/s when 16 and 512 processes are used, respectively. More processes means greater I/O concurrency at each data server and less spatial locality, especially when a barrier is not applied between I/O operations.

Because a disk's efficiency is directly correlated to the sizes of block-level requests dispatched to it, we collected block-layer request activities using the *blktrace* tool to obtain the distributions of request sizes shown in Figures 2(c) and 2(d) for 64KB and 65KB requests, respectively. In Figure 2(c) most requests are of only two sizes: 72% of for 128 sectors (64KB) and 18% for 256 sectors (128KB). In contrast, Figure 2(d) shows a much greater fraction of small requests, demonstrating the generation of fragments.

Figure 2(b) compares throughputs with file access using 64KB requests with varying offsets (Pattern III). Note that Pattern III becomes Pattern I for 0KB offset. The I/O throughputs with offsets are substantially lower than those that are aligned (Pattern I). For example, with 512 processes and offsets of 1KB and 10KB, the throughput with Pattern III is reduced by 36% (from 159.6MB/s to 102.1MB/s) and 49% (from 159.6MB/s to 81.8MB/s), respectively. In comparison a 1KB offset has smaller throughput degradation than that with larger offsets across different process counts (Figure 2(b)) because many of the fragments are relatively large (63KB). Figure 2(e) shows the block-level request size distribution for a 10KB-offset, where the two most frequent request sizes are 40KB and 88KB. Compared to Figure 2(c) the request sizes are significantly reduced. Replacing the *read* operation in *mpi-io-test* with *write* gives similar results.

Interestingly, and importantly, for both of the unaligned access patterns the I/O scheduler and normal prefetching do not consistently produce large requests even though the requests across the MPI program's processes are for sequential data in the same file. The reason is the nondeterminism of parallel execution, where requests received by a data server are issued by uncoordinated concurrent processes and so are less likely to produce opportunities for in-kernel prefetching and request merging in the I/O dispatch queue [29], [33].

In general the larger a request, the more servers the data is striped over and the more servers will be affected by inefficiency due to serving the request's fragments. To demonstrate this we ran a program to synchronously send requests of constant size. If the request size is a multiple k of the 64KB striping unit size it is served by Servers 0, 1, ..., $k-1$. If the request size is 1KB larger than a multiple of 64KB it is served by Servers 0, 1, ..., $k-1$, and k, where the last 1KB fragment is served by Server k. The sub-requests to the first k servers are contiguous. Because each server is normally serving more than one process concurrently, we ran another program to simultaneously read 64KB random data segments from Server k. We ran 16 processes to collectively issue these requests. Figure 3 compares the system's throughput with and without fragment requests for different valuse of k, and each with and without a barrier between I/O operations. The general trend is that throughput in the presence of

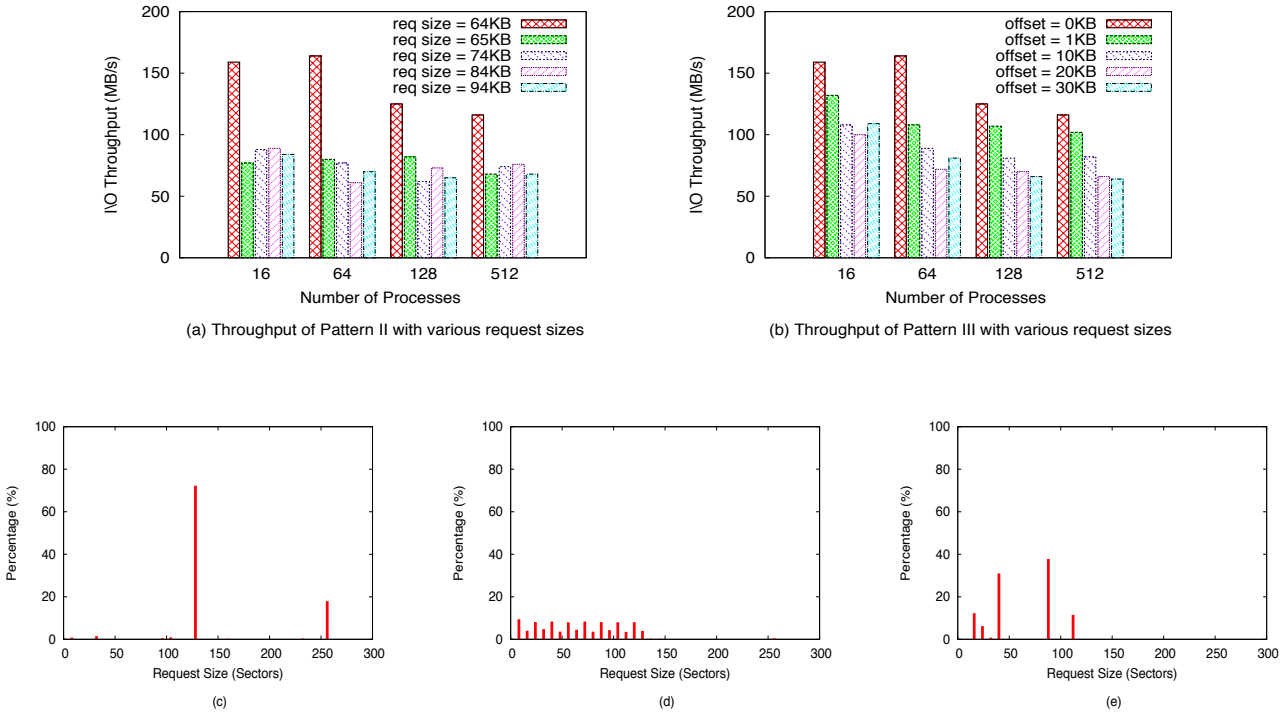


Figure 2. I/O throughput and block-level request size distribution with various alignment patterns. (a) I/O throughput with differing request sizes and process counts; (b) I/O throughput with 64KB requests and differing offsets and process counts; (c) Block-level requests' size distribution with aligned 64KB requests; (d) Block-level requests' size distribution with 65KB requests; (e) Block-level requests' size distribution with 64KB requests with 10KB offset. In Figures (c), (d), and (e) the block-level requests are measured in the disk sector size unit of 0.5KB.

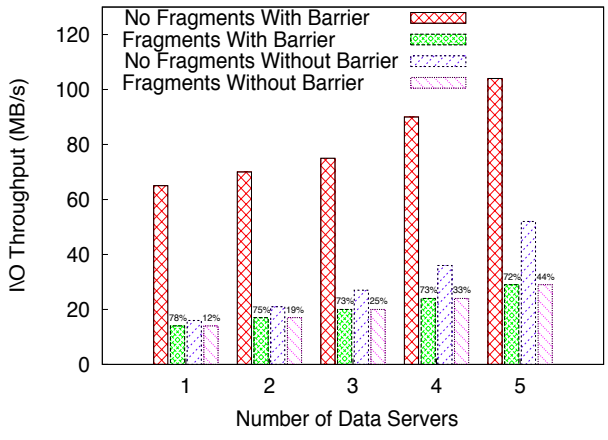


Figure 3. Throughput with differing numbers of servers involved in the serving of requests. The X axis describes the number of servers serving non-fragment requests. The numbers above the bars indicate respective reductions in throughput due to fragments.

fragments is significantly lower. Moreover, as the number of servers increases, relative throughput increases more slowly in the presence of fragments. If a barrier is used between requests the serving of sub-requests is synchronized across the servers, which helps improve access spatial locality and increase throughput. However, serving fragments with sub-optimal efficiency can not only delay the completion of its owner's request, but also delay all requests synchronized by the barrier, resulting in even greater loss of performance. We call the phenomenon of the increasing loss of performance with increasing number of servers involved in a request serving, due to the reduced efficiency of one or more servers serving fragments, the *striping magnification effect*.

B. Using SSD to Eliminate Unaligned Data Access

We propose a scheme that employs a SSD of relatively small capacity at each data serverW to efficiently serve fragments identified at the client side. Because SSD's performance is less sensitive to spatial locality, a fragment can be quickly served by SSD and so avoid being the last finished sub-request of its owner's request.

In summary, we make the following contributions.

- We extensively investigate the performance implications of unaligned data access using benchmark programs with various request patterns in parallel file

systems and show that a mismatch of the request pattern and file striping pattern can significantly compromise I/O performance.
- We propose *iBridge*, a scheme leveraging SSD's high random access performance to remove the inefficiency caused by the mismatch. Based on a dynamic analysis of resource-effectiveness, *iBridge* prioritizes the use of SSD for serving small sub-requests (fragments) over other regular random requests for increased performance benefits. Throughout the paper regular random requests specifically refer to those that by themselves are smaller than a pre-set threshold, such as 20KB, while a fragment refers to a sub-request that is a component of a larger request spanning multiple data servers and whose size is smaller than a pre-set threshold.
- We describe the design and implementation *iBridge* in the MPI-IO library and the PVFS2 file system for identifying fragments and transparently pre-loading/pre-writing them to make unaligned access as efficient as aligned access. By replaying traces of scientific programs such as *ALEGRA*, *CTH*, *S3D*, and running representative MPI-IO benchmarks such as *ior-mpi-io*, *mpi-io-test*, and *BTIO*, we show that *iBridge* can effectively bridge the performance gap between unaligned and aligned access and thereby improve I/O throughput by more than a factor of two on average.

II. The Design of *iBridge*

The objective of *iBridge* is to use SSDs to serve fragments produced by unaligned data accesses. Such a use of SSDs represents a highly desirable combination: SSD's strong performance advantage for random access, fragments' relatively small sizes, and a strong need to efficiently serve fragments.

To make the approach truly effective there are several questions to be answered in the design. First, how to distinguish regular random requests and fragments? Second, what are the criteria for determining whether a regular random request or a fragment should be admitted into the SSD? Third, when SSD space is limited for the serving of small requests, how should it be allocated between regular random requests and fragments? In this section we describe the architecture of *iBridge* and the management of data on the server side to answer these questions.

A. The Architecture

The *iBridge* scheme has client-side and server-side components. In a parallel file system like PVFS2 or Lustre a large request is split at the client into a number of sub-requests that are issued to respective servers. We call this request the sub-requests' *parent*, and the sub-requests are each other's *siblings*. The data servers are not aware of the distinction between requests and sub-requests so the client-side component is responsible for identifying fragments and passing the information to the data servers. We instrument the client-side PVFS2 function *io_datafile_setup_msgpairs()* to flag fragments. As long as the striping unit size of the parallel file system is known at the client side, sub-requests belonging to a request can be determined in this way. To be designated a fragment a sub-request must be smaller than a pre-defined threshold. In addition to setting the fragment flag *iBridge* also passes the identifiers of the servers holding this fragment's sibling sub-requests. This information allows a data server to assess the potential performance effect of serving the fragment on its parent request, accounting for the striping magnification effect.

On the server side the *pvfs2-server* daemon responsible for creating I/O jobs (I/O requests) is instrumented to retrieve the fragment flag from a request. At a server both the disk and the SSD are managed by their respective local file systems (Linux Ext2 on our experimental platform). In *iBridge* the SSD is treated as a cache for the local disk to store selected writeback dirty data and pre-loaded data for reading. *iBridge* maintains a mapping table to record data and their statuses (dirty or clean). This table is backed up on the SSD. Because sequential writes are much more efficient than random writes on SSD, *iBridge* writes new data into the SSD sequentially into a pre-created large file that is maintained much like a log-based file system.

B. SSD Management for Fast and Balanced Disk Access

Because the SSD is of relatively small size only small requests (or sub-requests) should be cached. However, we must dynamically evaluate the potential performance benefit of redirecting regular random requests and fragments to prioritize their eligibility for caching. Conceptually the metric is by how much faster, in terms of average request service time, a disk could be if a request to the disk were served by its companion SSD. However, this concept must be expanded for caching fragments to a disk to account for the fact that where a fragment (disk or SSD) is served also affects the efficiency of the disks serving the fragment's siblings.

To quantitatively evaluate the benefit we calculate the average request service time T_i for the ith request arriving at and served by a disk by

$$T_i = \frac{T_{i-1}}{8} + \frac{(D\_to\_T(\lambda_i - \lambda_{i-1}) + R + Size_i/B) * 7}{8} \quad (1)$$

where λ_i is the location of the ith request, which is represented as the logical block number (LBN) of its first requested block, $Size_i$ is the size of the ith request, B is the disk's peak throughput, R is the disk average rotation time, and D_to_T is a function for converting the disk seek distance to seek time. We use the approach described by Huang et al. to obtain this function from an offline profiling of the disk [12]. In the calculation of average time we incorporate a decay effect so that more recent requests are

better represented: specifically, we adopt an approach that is similar to the one developed in Linux for anticipatory scheduling by using the weights $1/8$ and $7/8$ for the last average value and the new one, respectively [15].

If the ith request is served at SSD the disk's average request service time does not change, i.e.,

$$T_i = T_{i-1}. \tag{2}$$

Therefore, the average service time T_i should be updated differently depending on where the ith request will be served. If the request is served at the disk, T_i $(= T_i^{disk})$ should be updated according to Equation (1). If the request is served at the SSD, T_i $(= T_i^{ssd})$ should be updated according to Equation (2). Their difference, $T_i^{ret} = T_i^{disk} - T_i^{ssd}$, represents the *return*, or the benefit of serving the i^{th} request at the SSD. A positive T_i^{ret} indicates that serving the request at the disk will increase the disk's average service time, i.e. slow the disk down. In such a case the ith request will be served at the SSD. Otherwise, serving the request at the disk helps improve the disk efficiency and there is no need to serve it at the SSD.

The return T_i^{ret} can be underestimated for a fragment if it is the slowest among its siblings because of the striping magnification effect. To know which server is currently the slowest in terms of disk efficiency, each server runs a daemon that periodically (every second by default) reports its current T_i value to the metadata server, which also has a daemon to receive the values and then broadcast this up-to-date information to each data server. If the ith request reaching a data server is a fragment we need to check if its T_i^{ret} is underestimated. If this current T_i value is not the largest among the current T values of the disks holding the fragment's siblings, being a fragment does not provide additional benefit, as its parent request's service efficiency is bottlenecked at some other server that has the largest average efficiency. Accordingly, the return of this fragment request is $T_i^{ret\_frag} = T_i^{ret}$. Otherwise, this fragment holds the largest T_i value, which is denoted by T^{max}. We denote the second largest T_i value by T^{sec_max}. The return of serving the ith request, which is a fragment with n sibling subrequests, at the SSD should be updated by

$$T_i^{ret\_frag} = T_i^{ret} + (T^{max} - T_i^{sec\_max}) * n. \tag{3}$$

When the ith request, either a regular random request or a fragment, arrives at a data server, we calculate its T_i^{ret} or $T_i^{ret\_frag}$, respectively, and if it is positive it will be served at the SSD. In addition to answering the question at each server of which requests should be served by the SSD, in consideration of the limited capacity of the SSD we also need to quantitatively prioritize, according to their return values, the caching of the requests in the SSD.

To enforce the caching priority we partition the SSD space between the two types of requests—regular random requests and fragments. When requested data is cached in the SSD its corresponding return value is recorded with it. For all of the data of the same type cached in the SSD we calculate the average return values and the SSD space is partitioned proportionally to the types' respective averages. In this way fragments on a slow disk causing their completed sibling sub-requests to wait will produce a larger average return value and have greater SSD space allocated. The cached data items are replaced using the LRU replacement algorithm if the allocation for its type is full. If the request to be redirected to the SSD is a write, its data is written into the SSD. If a read request arrives, *iBridge* will first check the mapping table to see if it is cached. If so the data will be read from the SSD, otherwise the request is served by the disk. However, *iBridge* still evaluates its return value and determines if its data should be cached. If so *iBridge* will write the data to the SSD when the SSD is idle. During quiet I/O-device periods a system thread is woken to write dirty cached data in the SSD to the disk. These writes are scheduled to form as many long sequential accesses as possible for higher disk efficiency. To ensure reliability, the dirty entries of the mapping table are immediately updated on the SSD with the write requests to the SSD.

iBridge cannot help with I/O efficiency of read requests if the requested data have not yet been cached in the SSD. However, a production MPI program is often executed on a parallel computer many times, possibly with different sets of parameters. As the data access patterns of its processes are generally consistent from one run to another run, the performance-compromising fragments identified and cached in the SSD in one run are very likely to appear in the following runs and *iBridge* can improve performance.

In the discussion we assume each data server has only one disk and one SSD. An extension to a system with more disks or greater SSD capacity would be straightforward and would not require modification of the described design.

III. Performance Evaluation

A. Experimental Setup

iBridge was prototyped on the Darwin cluster at Los Alamos National Laboratory. The cluster includes 116 48-core (12 core by 4 socket) 2GHz AMD Opteron 6168 nodes running Fedora Linux of kernel-2.6.35.10. All nodes were interconnected with a dual-rail 4X QDR Infiniband network. We configured eight nodes as data servers and one node as the meta-data server for the PVFS2 2.8.2 parallel file system. Each data server had one 7200-RPM disk drive (HP model MM0500FAMYT) and a 120GB SSD drive (HP model MK0120EAVDT).

Table II summarizes the performance characteristics of the storage devices. NCQ is enabled on all disks. The Noop [4] I/O scheduler is used for SSD, and CFQ [2] for hard disk. MPICH2 [19] compiled with ROMIO MPI-IO was used to generate executables. By default data are striped over the data servers with 64KB striping unit size. When *iBridge* is

	SSD	Hard Disk
Capacity	120GB	1TB
Interface	SATA	SAS
Sequential Read	160MB/s	85MB/s
Random Read	60MB/s	15MB/s
Sequential Write	140MB/s	80MB/s
Random Write	30MB/s	5MB/s

Table II
COMPARISON OF BASIC PERFORMANCE OF THE SSD AND HDD DEVICES USED IN THE EXPERIMENTS. 4KB REQUESTS ARE USED IN THE BENCHMARKING.

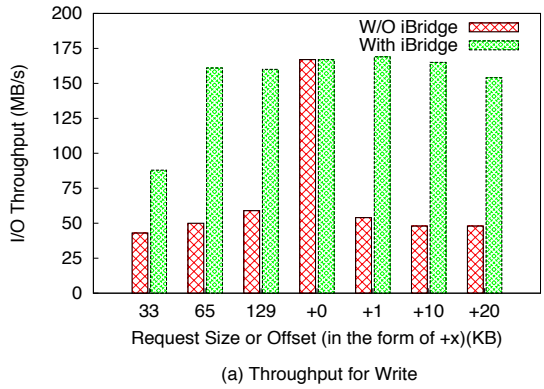
enabled an SSD partition of 10GB is used and the thresholds for determining regular random requests and fragments are both 20KB except where otherwise specified. To make our comparison fair and conservative we include in program execution time the time for writing dirty data back to the hard disk after program termination. Before each run we flush the system buffer caches to ensure that all requested data are served at the storage devices. To ensure that write throughput on the storage devices is correctly measured we flush the dirty blocks in the memory to the storage devices every second.

In this section we experimentally answer the following questions.

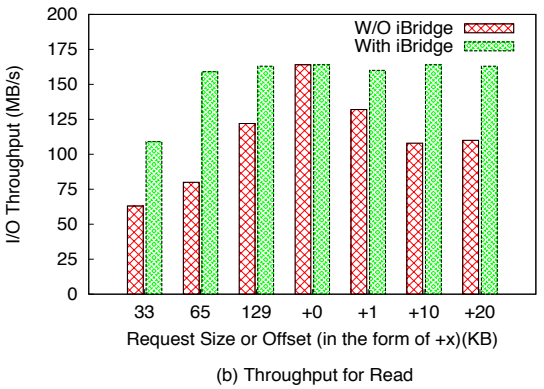
- Can unaligned requests be effectively served using *iBridge* in hybrid storage systems?
- Is *iBridge* effective for real scientific workloads with diverse data access patterns?
- How efficiently is *iBridge* implemented in PVFS2?
- What are the performance implications for *iBridge* when parameters such as the SSD size and threshold of request size are changed?

B. *The* mpi-io-test *Benchmark*

We first revisit the *mpi-io-test* benchmark that is designed to test the I/O throughput of PVFS2 for sequential data access. The benchmark's access can be configured as either writes or reads. We remove the barrier functions from the original program, making the program non-blocking and so allowing more concurrent I/O requests to be generated. In the experiments a 10GB file is accessed. First, with a 64-process instance of the program, we measure the throughput when the request size increases from 33KB, 65KB, to 129KB, as shown in Figures 4(a) and 4(b) for writes and reads, respectively. When *iBridge* is used the throughput for writes is increased by 105%, 183%, and 171%, respectively, compared to the stock system. At a request size of 33KB the sub-requests served at the disks are of sizes between 20KB and 33KB, substantially smaller than the 64KB striping unit size, while the sub-requests served by the disks for the 65KB and 129KB request sizes can be much larger. Therefore, *iBridge*'s throughputs for the 65KB and 129KB request sizes are much larger, fairly close to that with the fully aligned 64KB requests, which is 167MB/s and shown as the "+0" bars in Figure 4(a). For three request sizes, 33KB, 65KB, and 129KB, the data served at the SSDs account for 19%, 10%, and 4%, respectively, of the total amount of data accessed. We can see that *iBridge* is most effective for access of very small fragments. The performance trend is similar for read requests, as shown in Figure 4(b).



(a) Throughput for Write



(b) Throughput for Read

Figure 4. Throughputs of the *mpi-io-test* benchmark with various request sizes and request offsets. The requests in the program can be configured either as (a) writes or (b) reads. For the cases where requests have a '+xKB' offset, the request size is always 64KB.

Next we introduce an option for the program to set a request offset. We use a 64KB request size with 64 processes to execute *mpi-io-test*. With no request offset a 64KB request is dedicatedly served by one data server, and no cross-boundary data access occurs. However, when an offset is introduced every request is served by two data servers. The results with offsets are denoted by $+x$ in Figures 4(a) and 4(b) for writes and reads, respectively. Because the identified fragment requests are served by SSDs, *iBridge* can efficiently serve large requests from the disks. As evidence of the improved efficiency we show the distribution of sizes of block-level read requests in Figure 5 for a file offset of 10KB. As shown, request sizes of 128 sectors and 256 sectors (sector size 0.5KB) predominate, in contrast to what is shown in Figure 2(e) wherein much smaller requests are

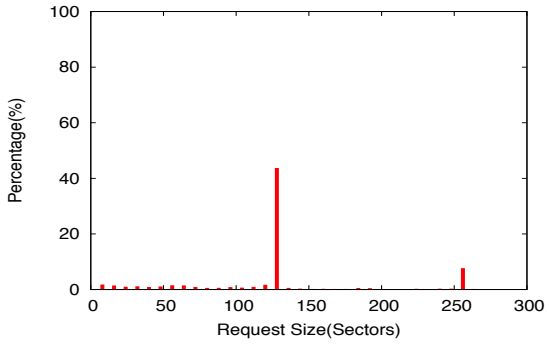


Figure 5. Block-level request size distribution with 64KB requests and 10KB offset when *iBridge* is used for *mpi-io-test* with read requests.

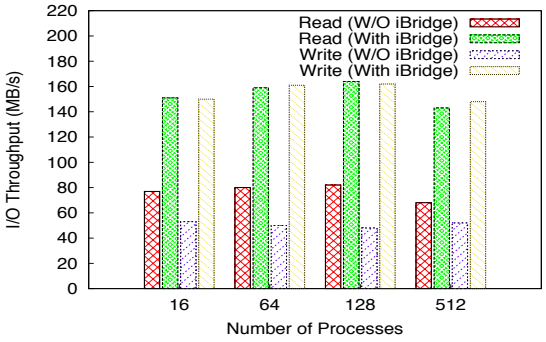
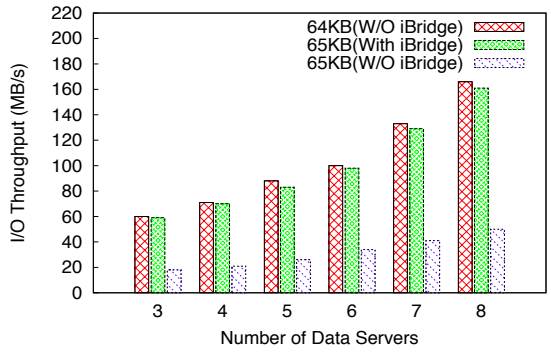


Figure 6. Throughputs without and with *iBridge* as process count increases.

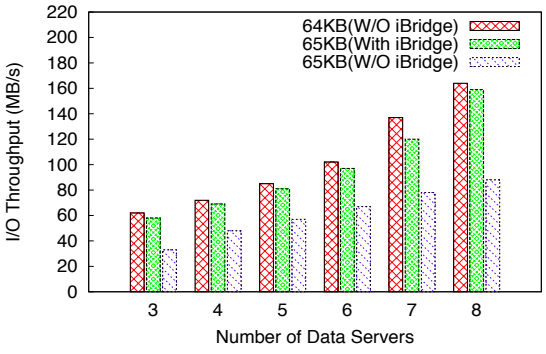
frequent. When the offset is 0KB all requests are aligned and *iBridge* does not redirect requests to the SSDs, so *iBridge* has the same throughput as the stock system. However, with non-zero offsets the throughout of *iBridge* changes little while that for the stock system is reduced significantly by 40% and 226% on average for reads and writes, respectively. In the stock system write requests are served more efficiently than read requests. By comparing Figures 4(a) and 4(b), it is apparent that in the stock system write throughput is considerably less than corresponding read throughput in all test scenarios except for the aligned 64KB (+0KB) case. This is because the hard drives installed in our testbed exhibit up to three-times different bandwidth (5MB/s vs. 15MB/s) for random writes and random reads as shown in Table II. By applying *iBridge* the throughput gap between writes and reads is largely closed and both approach that of aligned access.

Next we investigate the scalability of *iBridge* with increasing number of processes. In the experiment the request size is 65KB. As we increase the process count of *mpi-io-test* from 16, 64, 128, to 512, I/O concurrency at data servers correspondingly increases. I/O throughput with and without *iBridge* is shown in Figure 6. *iBridge* consistently improves the throughput, by 154% on average, for both reads and writes. When 512 processes are used the throughput is moderately lower than with a smaller number processes for both the stock system and *iBridge*. We believe that this is caused by access interference on the disks due to highly concurrent requests from a large number of processes. In these experiments only 10% of requested data is served by the SSDs.

Lastly we investigate the scalability of *iBridge* with increasing number of data servers. In the experiment we run *mpi-io-test* with 64 processes and differing numbers of data servers over which the file data is striped. For each server configuration we first use the stock system to serve requests of 64KB with the aligned access pattern to serve as a performance reference. We then use requests of 65KB



(a) Write



(b) Read

Figure 7. Throughputs of the *mpi-io-test* benchmark for request sizes 64KB and 65KB as data server count increases. The requests in the program can be configured as either writes (a), or reads (b).

to assess the performance of unaligned access in the stock system and with *iBridge*. The general performance trend is that with a larger number of data servers the throughput improves in all of the test cases as more servers are available to concurrently serve requests. For 65KB requests *iBridge* consistently performs better than the stock system, nearly closing the large performance gap between access with 65KB unaligned requests and 64KB aligned requests in the stock system. As the gap increases with increasing number of servers, the improvement provided by *iBridge* also increases, especially for write requests, which have lower throughputs than read requests with unaligned access.

C. *The* ior-mpi-io *Benchmark*

In this section we use the *ior-mpi-io* benchmark to study the performance of *iBridge* with random data access patterns. This benchmark is from the ASCI Purple benchmark suite developed at Lawrence Livermore National Laboratory [14]. We run the benchmark with 64 processes accessing a 10GB file. In the program a file is split into 64 chunks of equal size and each process is responsible for sequentially reading or writing one data chunk using requests whose sizes can be configured. However, because requests for data at the same relative offset are issued concurrently by different processes, the effective access pattern is random from the perspective of a parallel file system. In the experiments request sizes range over 33KB, 64KB, 65KB, and 129KB. The program's throughputs are shown in Figures 8(a) and 8(b).

For both reads and writes *iBridge* significantly improves the throughput except for the case of fully aligned 64KB requests when it has the same throughout as the stock system. On average *iBridge* achieves a larger improvement for writes (169%) than that for reads (48%), as writes from the SSD to the disks can be highly optimized for strong spatial locality. For request sizes of 33KB and 65KB, 19% and 10% of total data are served from the SSDs, respectively, reducing the load on the hard disks, which allows *iBridge* to produce greater improvements, especially for writes. In contrast, for 129KB accesses only 4% of data are served from the SSDs. With such little help from the SSDs the improvements are still substantial—35% and 60% for reads and writes, respectively.

D. *The* BTIO *Benchmark*

Next we study the performance of *iBridge* in serving regular random requests with macro-benchmark *BTIO*, a Fortran MPI program for solving the 3D compressible Navier-Stokes equations [21]. We compile the code with the MPI-IO option and use computing scale C which generates 6.8GB data. In the experiment we increase the number of parallel processes from 9, 16, 64, to 100. The program generates random and very small I/O requests during execution. Unlike the previous two benchmarks, with the increase of process count from 9 to 100, I/O request size is correspondingly

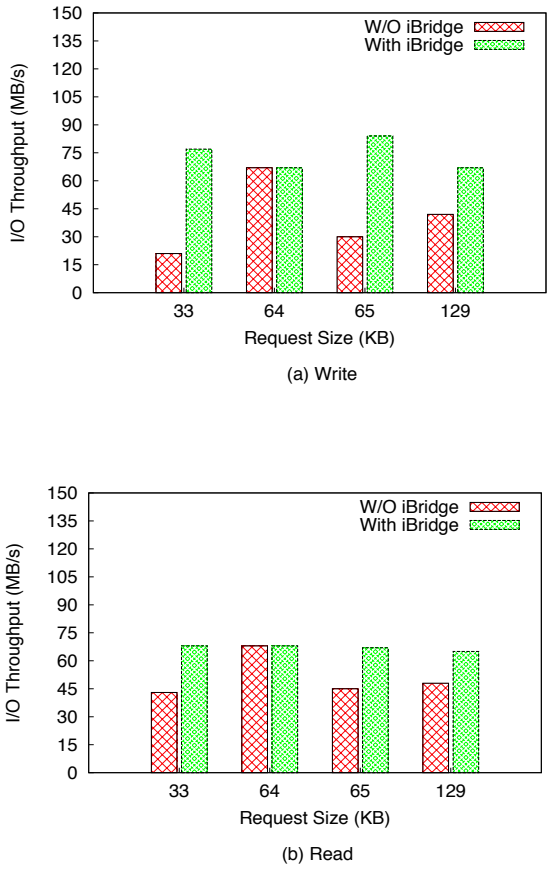


Figure 8. Throughputs of *ior-mpi-io* when request size is increased from 33KB to 129KB. Both writes (a) and reads (b) are evaluated.

reduced from 2160B to 640B, which are all regular random requests. As Figure 9 shows, the program's execution times are reduced by 45%, 55%, 61%, and 59%, for 9, 16, 64, and 100 processes, respectively. Because the threshold for determining random requests is 20KB, all write requests are served by the SSDs. By using SSDs, the proportion of I/O time in the program's total execution time is reduced from 58% to 4% on average with *iBridge*.

We also run the benchmark without *iBridge* with the entire data file on SSDs to determine how efficiently *iBridge* is implemented. Figure 10 shows the results with various numbers of processes. We observe that *iBridge* helps further improve I/O performance over the parallel file system using only SSD for storage. *BTIO* is a write-intensive program and the data in the write requests are sequentially written to the SSDs in a log-structured format by *iBridge*. However, without *iBridge* these requests were often written to random locations on the SSDs, resulting in the reduced throughputs due to the SSD's bandwidth gap between sequential and random writes (140MB/s and 30MB/s, respectively) as shown in Table II.

We also use *BTIO* to demonstrate the effect of available

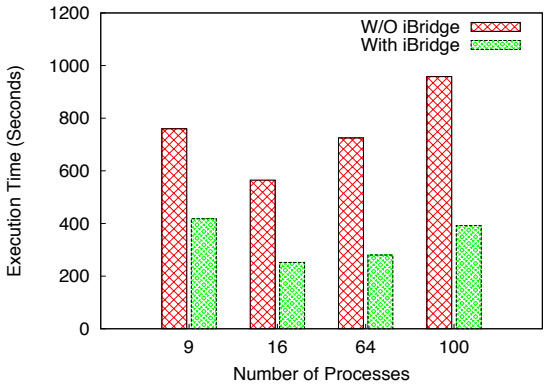


Figure 9. *BTIO*'s execution times using different process count with and without *iBridge*.

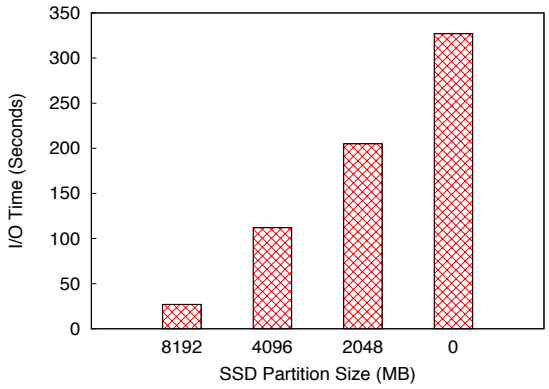


Figure 11. *BTIO*'s I/O time as a function of SSD capacity.

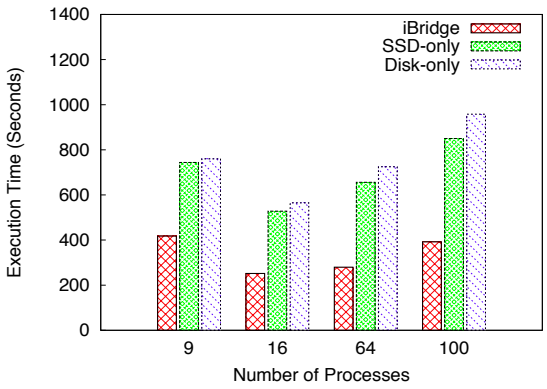


Figure 10. *BTIO*'s execution times with and without *iBridge*. Disk-only refers to the stock system, and SSD-only refers to the system in which only SSD is used as storage.

SSD capacity on I/O performance when using *iBridge*. We decrease the size of SSD available for serving regular random or fragment requests from 8GB, where all data could potentially be served at the SSD, to 0GB where the SSDs are effectively unavailable. The I/O times reported by the benchmark are given in Figure 11. We observe an almost-linear relationship between amount of data cached and I/O performance for this benchmark. The I/O time is increased by 12 times when the SSD size is 0GB, but the program's total execution time is only increased by 2.2 times because its computation time is also significant.

E. Trace Replay of Scientific Programs

In this section we evaluate *iBridge* by replaying the four scientific workload traces examined in Section I. The first two traces, *ALEGRA.2744* and *ALEGRA.5832*, are derived from programs developed for shock and multiphysics simulations. The third trace, *CTH*, contains I/O accesses of strong shock wave, solid mechanics programs. Trace *S3D* is from a combustion simulation. We replay the traces, which provide the offset and size of each request, but not the process ID that issued it, with a single process using the MPI-IO library to access data in PVFS2. We restrict the data size to 10GB during trace replay. The average request service times without and with *iBridge* are shown in Table III. With *iBridge* the request service times are reduced by 13.9%, 18.7%, 25.9%, and 29.8%, respectively. Because *CTH* and *S3D* have 12% and 24% more random and unaligned requests, respectively, than *ALEGRA*, they exhibit larger performance improvements. The average request size of *S3D* is significantly larger than the other three workloads, making its average service request time about twice as long.

	ALEGRA.2744	ALEGRA.5832	CTH	S3D
Stock	16.6ms	17.2ms	19.4ms	36.0ms
iBridge	14.2ms	14.0ms	14.4ms	25.3ms

Table III
COMPARISON OF REQUEST SERVICE TIMES FOR I/O TRACES REPLAYED WITH AND WITHOUT USING *iBridge*.

F. Performance of Heterogeneous Workloads

Fragments and regular random requests are handled differently by *iBridge* considering their relative benefits of being served by SSD. To assess its effectiveness we run *mpi-io-test*, which generates fragments, concurrently with *BTIO*, which generates regular random requests, to observe the behavior of *iBridge* under heterogeneous workloads. Specifically, *mpi-io-test* is executed with 64 processes and 65KB requests to write to a 10GB file. *BTIO* is executed with the C computing scale and 64 processes to access another file of 6.8GB. We first run the two benchmarks on the stock system without SSDs. Then the SSD cache, whose total

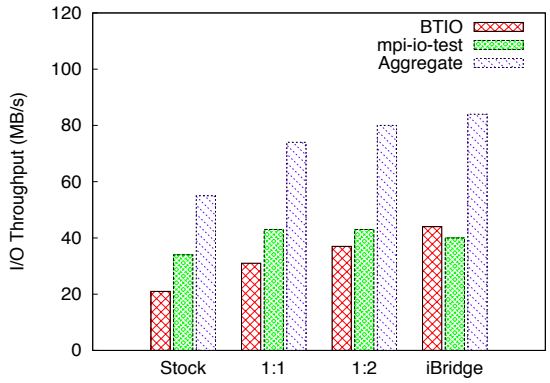


Figure 12. I/O throughput when *BTIO* and *mpi-io-test* are run concurrently. For *iBridge*, the SSD space is either statically partitioned 1:1 or 1:2 for regular random requests and fragments, or allowed to dynamically adapt.

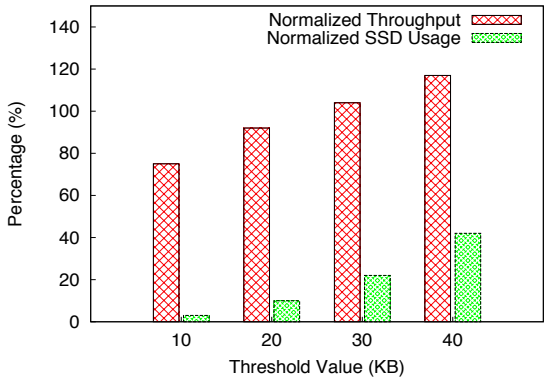


Figure 13. Throughput of *mpi-io-test* with 65KB requests normalized to that of aligned 64KB requests (164MB/s), and SSD usage with 65KB requests normalized to the total amount of accessed data (10GB), as the request size threshold value increases from 10KB to 40KB.

size is 8GB, is statically divided into two partitions with relative sizes either 1:1 or 1:2, for regular random requests and fragments, respectively, for use by *iBridge*. We then compare with *iBridge* dynamically adjusting the partition as it is designed.

Figure 12 shows the I/O throughput of the two benchmarks, and the system's aggregate throughput. *iBridge* achieves a 84MB/s system-wide throughput, which is 53% higher than that of the stock system where SSD is not used. Compared to the 1:1 and 1:2 cases where the SSD is statically partitioned, *iBridge*'s dynamic partitioning improves the aggregate I/O throughput by 13% and 5%, respectively. As shown in Figure 12, the *BTIO* benchmark has consistently higher throughput improvement over *mpi-io-test* because it issues requests of very small size and serving such requests on the SSDs can significantly improve the I/O efficiency of the disks.

G. Effect of Request Size Threshold

iBridge uses a request size threshold to determine whether a request should be considered a regular random request or a fragment. We have used a default 20KB as the threshold. Next we investigate the effect of the threshold size on I/O performance by running *mpi-io-test* with 64 processes and 65KB request size. The request size threshold is increased from 10KB, 20KB, 30KB, to 40KB. Figure 13 shows the throughputs normalized to that with aligned 64KB requests and 64 processes. It also shows the amount of SSD space used in all data servers by *iBridge*. This value is normalized to the total amount of data accessed in one execution.

In general, as the threshold value increases I/O throughput improves because more requests are served by the SSDs. When the threshold value is 40KB, we see an I/O throughput increase of 56% over that with a 10KB threshold. However, at the same time the normalized SSD usage also increases from 3% with the 10KB threshold to 42% with the 40KB threshold. We chose 20KB as the default threshold to balance performance and SSD longevity. For the *mpi-io-test* benchmark the default threshold achieves a throughput only 21% lower than with a 40KB threshold, but with SSD usage 76% less.

IV. Related Work

iBridge is proposed to address the performance loss due to unaligned data access by using an SSD-augmented hybrid storage system at data servers. Here we survey related work on approaches to handling unaligned data access, and other SSD-related approaches, for performance optimization of parallel I/O.

A. Approaches to Handle Unaligned Data Access

Unaligned data access has been identified as one of the I/O bottlenecks in high performance computing [17]. Ward presented four rules for end-users to develop I/O-friendly applications [27]. However, developers usually consider only logical file address space in their programming. Accesses that are well aligned in the logical file address space can be mapped to unaligned disk accesses because of data striping and using I/O optimization techniques such as collective I/O and data sieving [26] in MPI-IO middleware. Data prefetching techniques [6], [8], which hide I/O time behind computation time, become less effective as a program can spends more time on I/O than on computation, and with high access concurrency due to uncoordinated requests from different processes of an MPI program [31]. Zhang et al. proposed a data-driven execution mode to improve I/O efficiency via program pre-execution when program performance hinges on I/O resources [33]. However, the advantage of the method is reduced when I/O dependency becomes significant. Data caching can largely hide performance loss for unaligned data access [20], [18], [24]. However, DRAM-based cache can be of very limited size and DRAM is

much more expensive than SSD. Bent et al. proposed PLFS to handle check-pointing workloads that access a shared file in parallel file systems [5]. Through rearranging data access space using a log-structured file system, unaligned data access could be reduced. Nevertheless, this approach may not be effective for regular workloads, as spatial locality is largely lost in the log file system.

Instead of using SSD, Wang et al. proposed to replicate frequently accessed data chunks at the compute nodes' local disks to reduce data access latency [28]. The frequently accessed data chunks are identified through analysis of I/O traces collected in a profiling run. The legitimacy of their approach of identifying data of certain access patterns through profiling runs is built on their observations that "In scientific applications, file access patterns are generally independent of the data values stored." Using on-disk data replication for improving I/O performance is also proposed on data servers [32]. Nowadays, SSD is a readily available device and is well-suited for speeding up the disk access. For read requests, our identification and in-SSD caching of fragments in prior runs for accelerating future unaligned read requests are based on a similar rationale.

B. Using SSDs for Parallel I/O Performance Optimization

Because the performance of the hard disk can be substantially degraded by random access because of their use of mechanical moving parts, the increasing trend towards high-end storage has elevated SSDs, essentially a uniform memory access device, to first class storage entities in Petascale machines. Gordon is the first SSD-based cluster for high performance computing [10]. Considering limited budgets researchers have proposed other methods to more economically use SSDs. Chen et al. designed *Hystor* to use disks to handle regular data and SSDs to handle performance critical data that are identified at run time based on data access patterns [9]. Compared to *Hystor*, *iBridge* uses SSDs as a temporal area for handling randomness caused by unaligned I/O requests. *iBridge* gives priority to more performance-critical unaligned data access than regular random requests. *iTransformer* [31] was recently proposed to help disk schedulers handle high I/O concurrency. Requested data are either committed to disks or to SSDs according to the I/O workload's spatial locality. SSDs have also been adopted in other layers of the memory hierarchy for effective data management. Prabhakar et al. proposed a multi-layer data staging scheme using DRAM, SSD, and disk to satisfy users' QoS requirements, mainly for checkpointing time [23]. Wang et al. implemented NVMalloc to share SSD space at compute servers for out-of-core HPC applications [30].

V. Conclusion

We have presented the design and implementation of *iBridge*, a hybrid storage scheme to handle the fragmentation resulting from unaligned parallel data access using the MPI-IO library and parallel file systems. *iBridge* identifies and serves fragments and regular random requests only when the cost-effectiveness of serving them on SSD can maximize the benefits of the use of SSDs. By doing so it distinguishes itself from works in which SSD is simply used for caching small and/or random data. *iBridge* is implemented in the client and server sides of the PVFS2 parallel file system. Our experimental evaluation, with both running of representative benchmarks and the replaying of real scientific workloads, shows that *iBridge* can improve I/O throughput by more than a factor of two on average for representative workloads.

Acknowledgments

The authors thank Marcus Daniels and Ryan Braithwaite (LANL) for their technical assistance with the Darwin cluster. We also thank the anonymous reviewers for their constructive comments. This work was supported by U.S. National Science Foundation under CAREER CCF 0845711, CNS 1117772, and CNS 1217948. This work was also funded in part by the Accelerated Strategic Computing program of the Department of Energy. Los Alamos National Laboratory is operated by Los Alamos National Security LLC for the US Department of Energy under contract DE-AC52-06NA25396.

References

[1] ALEGRA, The Shock and Multiphysics Family of Codes, Sandia National Laboratories, *http://www.sandia.gov/CTH/*.

[2] J. Axboe. Completely fair queueing (CFQ) scheduler, 2010. http://en.wikipedia.org/wiki/CFQ.

[3] HDF5 File Format Specification Version 2.0 http://www.hdfgroup.org/HDF5/doc/H5.format.html

[4] J. Axboe. Noop scheduler, 2010. http://en.wikipedia.org/wiki/Noop.

[5] J. Bent, G. Gibson, G. Grider, B. Mcclelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "PLFS: A Checkpoint Filesystem for Parallel Applications." In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2009.

[6] S. Byna, Y.Chen, X. Sun, R. Thakur, and W. Gropp, "Parallel I/O Prefetching Using MPI File Caching and I/O Signatures." In *Proc. of the ACM/IEEE SuperComputing Conference*, 2008.

[7] CTH Shock Physics, Sandia National Laboratories, *http://www.sandia.gov/CTH/*.

[8] Y. Chen, S. Byna, X. Sun, R. Thakur, and W. Gropp, "Hiding I/O Latency with Pre-execution Prefetching for Parallel Applications", In *Proc. of the ACM/IEEE SuperComputing Conference*, 2008.

[9] 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*, 2011.

[10] A. Caulfield, L. Grupp, and S. Swanson, "Gordon: Using Flash Memory to Build Fast, Power-efficient Clusters for Data-intensive Applications", In *Internaitonal Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[11] GPFS, IBM General Parallel File System, *http://www-03.ibm.com/systems/software/gpfs/.*

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

[13] I/O traces, Sandia National Laboratory, *http://www.cs.sandia.gov/Scalable_IO/SNL_Trace_Data/index.html*.

[14] Interleaved or Random (IOR) Benchmarks, 2010. http://www.cs.dartmouth.edu/ pario/examples.html.

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

[16] Lustre, the Lustre File System, *http://wiki.lustre.org/index.php/Main_Page*.

[17] 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*, 2009.

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

[19] MPICH2, Message Passing Interface, *hettp://www.mcs.anl.gov/research/projects/mpich2/.*

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

[21] "NAS parallel benchmarks, NASA Ames Research Center, 2009", http://www.nas.nasa.gov/Software/NPB.

[22] PVFS2, Parallel Virtual File System, *http://www.pvfs.org*.

[23] P. Prabhakar, S. Vazhkudai, Y. Kim, A. Butt, M. Li, and M. Kandemir, "Provisioning a Multi-Tiered Data Staging Area for Extreme-Scale Machines", In *31st International Conference on Distributed Computing Systems*, 2011.

[24] B. Settlemyer, "A Study of Client-based Caching for Parallel I/O", In *Doctoral Dissertation*, 2009.

[25] R. Sankaran, E. Hawkes, J. Chen, T. Lu, and C. Law, "Direct Numerical Simulations of Turbulent Lean Premixed Combustion", In *Journal of Physics: Conference Series*, vol. 46, pp. 38-42, 2006.

[26] R. Thakur, W. Gropp, and E. Lusk, "Data Sieving and Collective I/O in ROMIO", In *Proc. of the 7th Symposium on the Frontiers of Massively Parallel Computation*, 1995.

[27] L. Ward, "A Brief Parallel I/O Tutorial", In *SANDIA REPORT*,SAND2010-1915, 2010.

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

[29] Y. Wang, Y. Xue, K. Davis, and S. Jiang, "iHarmonizer: Improving the Disk Efficiency of I/O-intensive Multithreaded Codes", In *26th IEEE International Parallel & Distributed Processing Symposium*, 2012.

[30] C. Wang, S. Vazhkudai, X. Ma, F. Meng, Y. Kim, and C. Eagelmann, "NVMalloc: Exposing an Aggregate SSD Store as a Memory Partition in Extreme-Scale machines", In *26th IEEE International Parallel & Distributed Processing Symposium*, 2012.

[31] X. Zhang, K. Davis, and S. Jiang, "iTransformer: Using SSD to Improve Disk Scheduling for High-performance I/O", In *26th IEEE International Parallel & Distributed Processing Symposium*, 2012.

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

[33] X. Zhang, K. Davis, and S. Jiang, "Opportunistic Data-driven Execution of Parallel Programs for Efficient I/O Services", In *26th IEEE International Parallel & Distributed Processing Symposium*, 2012.