SSDUP: A Traffic-Aware SSD Burst Buffer for HPC Systems

Xuanhua Shi, Ming Li, Wei Liu, Hai Jin, Chen Yu
Services Computing Technology and System Lab
Big Data Technology and System Lab
Huazhong University of Science and Technology
Wuhan, China 430074
{xhshi,limingcs,weiliu0727,hjin,yuchen}@hust.edu.cn

Yong Chen
Department of Computer Science
Texas Tech University
Lubbock, Texas, USA
yong.chen@ttu.edu

ABSTRACT

Many *high performance computing* (HPC) applications are highly data intensive. Current HPC storage systems still use *hard disk drives* (HDDs) as their dominant storage devices, which suffer from disk head thrashing when accessing random data. New storage devices such as *solid state drives* (SSDs), which can handle random data access much more efficiently, have been widely deployed as the buffer to HDDs in many production HPC systems. *Burst buffer* has also been proposed to manage the SSD buffering of bursty write requests. Although burst buffer can improve I/O performance in many cases, we find that it has some limitations such as requiring large SSD capacity and harmonious overlapping between computation phase and data flushing stage.

In this paper, we propose a scheme, called SSDUP (a traffic-aware SSD burst buffer), to improve the burst buffer by addressing the above limitations. In order to reduce the SSD capacity demand, we develop a novel traffic-detection method to detect the randomness in the write traffic. Based on this method, only the random writes are buffered to SSD and other writes are deemed sequential and propagated to HDDs directly. In order to overcome the difficulty of perfectly overlapping the computation phase and the flushing stage, we propose a pipeline mechanism for the SSD buffer, in which the data buffering and data flushing are performed in pipeline. Finally, in order to further improve the performance of buffering random writes in SSD, we covert the random writes to sequential writes in SSD by storing the data with a log structure. Further, we propose to use the AVL tree structure to store the sequence information of the data. We have implemented a prototype of SSDUP based on the OrangeFS and performed extensive experimental evaluation. The experimental results show that the proposed SSDUP scheme can improve the write performance by more than 50% on average.

KEYWORDS

High Performance Computing; Hybrid Storage System; Solid State Drive; Burst Buffer.

ACM Reference format:
Xuanhua Shi, Ming Li, Wei Liu, Hai Jin, Chen Yu and Yong Chen. 2017. SSDUP: A Traffic-Aware SSD Burst Buffer for HPC Systems. In *Proceedings of ICS '17, Chicago, IL, USA, June 14-16, 2017,* 10 pages.
DOI: http://dx.doi.org/10.1145/3079079.3079087

ICS '17, Chicago, IL, USA

DOI: http://dx.doi.org/10.1145/3079079.3079087

1 INTRODUCTION

While *high performance computing* (HPC) systems are moving towards the exascale era, the I/O performance remains one of the main bottlenecks, especially for many data-intensive scientific applications. With the continuous and rapid growth of data volume, the storage demand of many HPC systems has reached the petabyte level [21]. Under this level of storage demand, *hard disk drives* (HDDs) are still used as the main permanent storage devices in HPC systems, partly because of their low cost and partly because HDDs can offer high bandwidth when accessing large continuous data chunks. However, HDDs have a major drawback and they suffer from the poor performance when the data are accessed randomly because of the slow mechanical movement of disk heads.

New storage devices such as *solid state drives* (SSDs) have been widely deployed in the HPC environment because of their near-zero seek latency and superior performance (especially for random accesses). However, SSDs are much more expensive than HDDs. Therefore, it is not a cost effective solution to use SSDs as the sole storage devices in large-scale production HPC systems, not to mention the technical limitations of SSDs, such as the issues of wear-out and limited lifetime. A popular solution to address the problem of random data access to HDDs is to use the SSD to buffer the data streams between HDDs and computing nodes, which has been adopted by many production supercomputers (i.e., Sunway TaihuLight and Tianhe-2).

Another feature of HPC systems is that most applications running on HPC systems are write-heavy. The exemplar applications include those from the domains of climate science, physics, earth science etc., which mainly perform the numerical simulations. Moreover, the write requests issued by these applications are bursty because the applications often generates and store a large amount of intermediate results [20] [19]. Further, in order to provide failure protections the concurrent processes of an application often perform checkpointing simultaneously and dump the in-memory data to the permanent storage, which generate the bursty write operations as well. The bursty random writes to HDDs could significantly degrade the performance of data-intensive applications running on HPC systems.

In order to address the above issue, *Burst Buffer* (BB) [19] has been introduced, which uses an SSD buffer as an intermediate layer between the compute nodes and the HDD-based storage servers to absorb the bursty write requests. In BB, the data generated by an application are first written to the SDD buffer, which effectively absorbs the bursty writes of the application since SDD offers very low latency. While the data in the SSD buffer are flushed to HDDs, the HPC system go on to process next computation tasks. Although BB significantly improves the write performance of HPC applications in many cases, we find that the workings of BB rely on the following

two conditions to achieve the expected performance improvement, both of which cannot be easily met.

Large SSD capacity. BB requires the SSD buffer to have the adequate capacity to store all data generated by the computation phase of an application. Otherwise, when the SSD buffer is fully occupied by the write data, the application has two options depending on the implementations of BB: writes the following data to HDD directly or blocks the incoming I/O requests till the BB has enough available capacity. The overall I/O performance would still be dominated by HDD in both cases, which goes against the original intention of burst buffer. This condition means that the capacity of the SSD buffer needs to be no less than the maximum size of the data generated by a computation phase of an application. However, an application can access hundreds of terabytes or even tens of petabyte of data. For instance, the total data transmission of the Earth1 application in the Mira supercomputer has reached about 10PB [21]. On the other hand, although the data size of individual applications has reached the petabyte level, such applications account for only 10% of all applications running in a typical HPC platform. The data size of the remaining 90% of the applications are still around several GBs [21]. There is now a dilemma: shall we use large (therefore expensive) SSDs to meet the need of a small fraction of applications or use moderate-sized (therefore cost effective) SDDs to satisfy most applications but sacrifice the high-demanding applications?

Overlapping computation with data flushing. BB needs to overlap the computing phase with the flushing stage of the data writing. It is not easy to meet this condition either. On one hand, it is difficult to predict the duration of a computation phase, which could be very variable subject to a number of factors. On the other hand, the HPC platforms use job scheduling tools, such as PBS [8], SGE [10], and slurm [9], to manage job submissions and executions. To the best of our knowledge, these job schedulers do not take the usage of storage resources into consideration. Therefore, it is very likely to happen that before the previous flushing stage is completed, the computing phase from either the same application or a different application start to write new data. Once the SSD buffer becomes full, the applications once again face the aforementioned two options, both of which unfortunately deliver low performance.

This work strives to address the above difficult issues in BB and develop the SSDUP, a traffic-aware SSD burst buffer, for large-scale HPC systems. The following strategies are developed in SSDUP to reduce the SSD capacity required to satisfy the performance of bursty, large-scale I/O accesses (or from another perspective, improve the I/O performance with the same SDD capacity).

First, in order to reduce the demand for the SSD buffer capacity, we select only a proportion of data to write on SSD, while the remaining data are written to HDDs directly without sacrifice the I/O performance. A novel I/O-traffic detection method is developed to detect the data access patterns of the running processes. Only the random access writes are directed to SSD while the remaining writes are deemed as sequential accesses and propagated to HDDs directly. Our I/O-traffic detection method is novel because the current method of detecting the data access pattern from multiple processes is mainly through calling the collective MPI-IO operations in the applications. This *client-side* method has the following limitations. i) The multiple processes are in the scope of the same application. Therefore, it can only detect the data access pattern within an application, not from different applications. ii) The method can only detect the data access pattern related to the currently invoking I/O operation, not across different I/O invocations. Our I/O-traffic detection component is located at the side of the storage server. The *server-side* detection method proposes a novel metric, termed *random factor*, to detect the randomness of the write streams, no matter they are across different I/O invocations by the same application or across different applications.

Second, in order to overcome the difficulty of accurately predicting the duration of the computing phase and further improve I/O performance, we develop a pipeline mechanism for the SSD buffer. In the pipeline mechanism, the SSD buffer is divided into two halves. While one half is receiving the writing data, the other fully occupied half flushes the data from SSD to HDD. With the pipeline mechanism, a portion of SSD buffer will have been flushed and therefore ready to accommodate new data even if the data produced by the current computation phase fully occupy the SSD buffer. When the new computation phase produces new data (either due to the inaccurate prediction of computation phases or I/O unawareness of job schedulers), which will block the application or be written to HDDs directly in the existing implementations of BB, the new data can still be written to the SSD buffer in our SSDUP and consequently the I/O performance can be significantly improved.

Finally, we only buffer the random writes to SSD. However, SSD is desired for sequential writes due to the features of SSD writing. In order to improve the performance of buffering random writes in SSD. We covert the random writes to sequential writes by storing the writes in a log-structured manner, i.e., appending the data to the end of the buffered files. A negative aspect of such a log structure is that the original sequence of file requests is lost. In order to address this issue, we propose to use the AVL tree structure to store the information of the file sequences. Our analysis shows that we only need a tiny fraction of extra storage space to store the AVL tree structure. Therefore, we significantly improve the performance of write buffering at a very small storage expense.

In summary, we develop a scheme, called SSDUP, in this research to improve the existing burst buffer. The detailed contributions are as follows:

- We carefully analyze the common access patterns in HPC systems and provide a random access detection method using a proposed *random factor*. Based on the identified access pattern, the I/O traffic is reshaped and random accesses are directed to the SSD for a highly efficient write buffering.
- We design a pipeline mechanism for the SSD buffer. It handles the buffering stage and the flushing stage in pipeline and can effectively mitigate the impact of not being able to well overlap the computation phase and the flushing stage.
- A log structure is used to covert the random writes to sequential writes in the SDD buffer and an AVL tree structure is further used to maintain the sequence information of the file requests.
- We have implemented a prototype of SSDUP based on the OrangeFS and performed extensive experimental evaluation. The experimental results have verified the effectiveness of the SSDUP scheme.

The rest of the paper is organized as follows. Section 2 describes the detailed design of the SSDUP solution. Evaluations and analyses are presented in Section 3. We then discuss related work in recent years and compare them with this research in Section 4. Finally, we summarize our work in Section 5.

2 DESIGN

In this section, we describe the design of the SSDUP (a traffic-aware SSD burst buffer) solution. The SSDUP contains four main modules, as shown in Figure 1, including a *random access detector* component that identifies random/irregular I/O accesses, a *data redirector* component that redirects I/Os to different devices, a *pipeline* component that orders and schedules I/Os in an efficient and pipelined way, and an *AVL tree management* that manages buffered data and maintains the data sequence to order them. We will introduce an overview of the SSDUP first, followed by detailed introduction of each component including the algorithm and data structures.

2.1 Overview of SSDUP

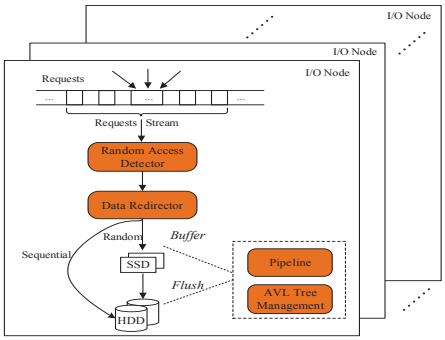


Figure 1: Architecture of SSDUP

SSDUP is designed as a part of OrangeFS [7], the latest version of PVFS2. Please note that the SSDUP methodology is general, and the design, algorithms, and data structures are applicable to other file systems. OrangeFS adopts a client/server model. It has been widely used as both an experimental and a production platform in HPC areas. Files are striped across multiple I/O nodes for concurrent accesses. When issuing an I/O request, the client first communicates with the metadata server to retrieve the data location, and then issues multiple sub-requests to I/O nodes where data is located.

Figure 1 shows the architecture of SSDUP. SSDUP lays on each I/O node and integrates with the *pvfs2-server* daemon, so the system needs no communication with other I/O nodes. After requests arrive at an I/O node, SSDUP groups the requests as sequences of accesses, called *requests streams*. Next, SSDUP reshapes the requests stream with the help of four modules. The random access detector is responsible for deciding whether the following requests stream is unsuitable for HDDs or not. The data redirector is responsible for sending data to the dedicated devices based on random access detector. The pipeline module is responsible for handling the data buffering stage and the flushing stage and maintain enough SSD space for the incoming requests. The AVL tree management module is responsible for managing the metadata of buffered data and sorting random data for better flushing performance. We explain details below.

2.2 Detecting Random Access

Random accesses cause a significant seek delay due to the need to move the disk head to the right location before transmitting any data. The I/O behavior of HPC applications can be influenced by many factors such as the way processes access data, the request size of each I/O, the number of processes participating in I/O, the number of data servers, the I/O subsystem of Unix kernel. All these factors may lead to random accesses, which is hard to detect from a single process's view.

To understand the relationship between access pattern and disk latency, we used IOR [4] to conduct a series of tests. Specifically, three different access patterns were tested: *segmented-contiguous*, *segmented-random*, and *strided* [11]. The total data size tested was 16 GB. Each I/O request size was 256 KB. The number of total processes varied from 4 to 128. We used 10 nodes, with 8 of them as compute nodes, and 2 of them as I/O nodes. Both I/O nodes were equipped with OrangeFS 2.9.3. Other details of the experimental platform are given in Section 3.

Segmented-Contiguous: In this pattern, each process accesses $1/n$ portion of the shared 16GB file (n represents the number of processes). Each process issues sequential requests to access part of data.

Segmented-Random: This pattern is similar to segmented-contiguous, and the only difference is that each process issues random I/O requests.

Strided: In this pattern, assuming the number of involved processes is n, the process ID is identified as $rank$, then in the ith iteration, process $rank$ issues I/O requests at offset $i * n + rank$.

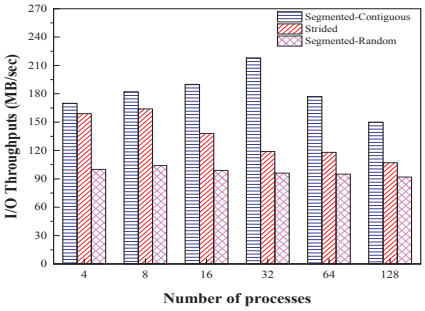


Figure 2: I/O throughput with different access patterns and different number of processes for the IOR benchmark

The experimental results are presented in Figure 2. It can be observed that the throughput increased first and then dropped with the increase of the number of processes in both segmented-contiguous and strided pattern cases. The reason for the throughput increase is that OrangeFS uses *asynchronous I/O* (AIO) as its default trove method [7]. AIO [1] can improve its throughput when the number of I/O participants grows, because the CFQ (*Completely Fair Queuing*) [2] scheduler queues more requests for better spatial locality by sorting and merging requests. When the number of processes continues to grow (from 32/16 processes to 128 processes), the throughput of segmented-contiguous accesses dropped

from 218 MB/s to 150 MB/s, and the throughput of strided accesses dropped from 164 MB/s to 107 MB/s, resulting in 31% and 34% degradation, respectively. The reason for this is that the CFQ scheduler cannot achieve better locality with a limited queue size when more processes issue I/O requests concurrently. The throughput of segmented-random accesses remained around 95 MB/s, because the CFQ scheduler can hardly merge any requests when the offsets are nonconsecutive.

To verify this observation, we also traced the offsets of these three access patterns. For instance, Figure 3 shows the case of 16 processes, where the x axis represents the sequence of requests. There were 65536 (16GB/256KB = 65536) requests in total issued to the I/O nodes, and Figure 3(a) illustrates the distribution of the first 128 requests. As shown in the figure, offsets of segmented-contiguous accesses are regular, whereas segmented-random accesses have completely random offsets (Figure 3(a) and Figure 3(b)). Offsets of strided accesses seem compact with a slight fluctuation, as shown in Figure 3(c). Then we also sorted the offsets in a unit of 128 requests, similar to what CFQ does. After sorting the offsets, the distribution of offsets is much more well-ordered. Apparently, such a sorted access order is an ideal situation. In fact, the order of requests issued from the CFQ scheduler to the device changes dynamically. If the requests' offsets in the queue are adjacent, they will be merged by the CFQ scheduler, and the seek distance is zero. Otherwise, the disk head needs to move around for relocation, and will have to move back and forth when the offsets are reverse-ordered, which leads to long access delay.

We introduce a notion of *random factor* (*RF*) in our design to indicate the frequency of disk head movement. After the offsets are sorted, we consider two requests as sequential requests if the distance between them equals to the request size. In this case, the random factor is 0. Otherwise, requests are considered random, and the random factor is 1. We calculate the sum of random factors in every requests stream, as shown in Equation (1), where S is the sum of random factors in a requests stream and N is the total number of distances, which equals to the length of the requests stream minus one.

$$S = \sum_{i=1}^{N} RF \tag{1}$$

Figure 4 illustrates the sorting procedure in a requests stream. Requests in a stream arrive out of order because of processes' random/irregular accesses and competitions. As we have defined above, in this example, the random factor between requested data item #2 and data item #3 is 0 because after being sorted, the offset distance between these two requests equals to the request size. On the other hand, the random factor between data item #4 and data item #7 is 1. Note that the random factor is defined based on the logical address instead of physical address. Although using the logical address to indicate disk head movement is not 100% accurate, the disk seek time is linearly related to the logical address distance in most cases [18]. Figure 5 shows the distribution of offsets after being sorted. In this figure, the solid black line between spots represents the condition when the random factor is 1. To make these figures more readable, we only present 32 offsets out of 128 in both segmented-random and strided patterns.

As shown in Figure 5, the random factor of segmented-contiguous accesses is 15, which represents 11% of total 127 distances (there are 127 distances for 128 requests). The random factor of segmented-random accesses is 127, which represents 100% of all distances, and the random factor of strided accesses is 57, which represents 45% of all distances.

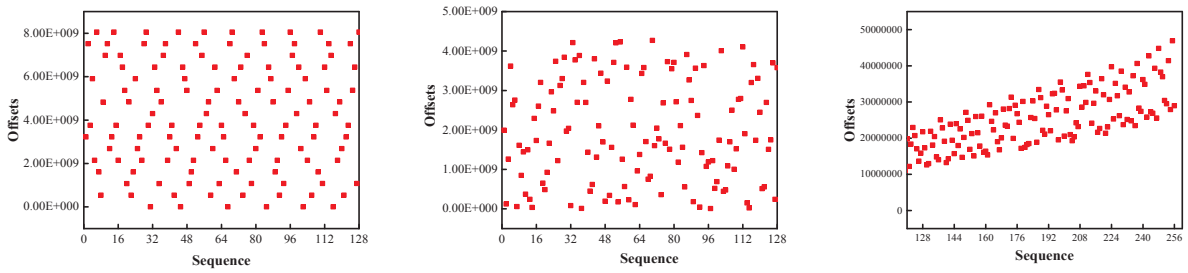
Algorithm 1 Redirection Algorithm

Require: Requests: *reqs*.
1: Send *reqs* to HDDs
2: **repeat**
3: Group *reqs* into requests stream
4: Sort the offsets of *reqs* in requests stream
5: Calculate S (sum of *RF*)
6: Calculate *percentage* of *RF*
7: **if** *percentage* > 30% **and** *reqs* being sent to HDDs **then**
8: Send *reqs* of next requests stream to SSDs
9: **else if** *percentage* < 45% **and** *reqs* being sent to SSDs **then**
10: Send *reqs* of next requests stream to HDDs
11: **else**
12: Send *reqs* of next requests stream to the current device
13: **end if**
14: **until** No incoming *reqs*

2.3 Data Redirection Based on Random Factor

The data redirector module in SSDUP is designed to transmit data, in random or sequential accesses, to different devices. Algorithm 1 shows the workflow of the data redirector component. At the beginning of an application's execution, data is written to the HDD. In the meantime, the offsets of requests are traced, in the unit of requests streams. The default length of a requests stream is 128, which is the same as the queue size of the CFQ scheduler. The setting of the length of a requests stream can be configured when the CFQ queue size changes. The offsets of a requests stream can be sorted then, and the sum of the random factors of the requests stream can be calculated. The *percentage* of the random factor is calculated as $percentage = S/N$, S is the sum of random factors and N is the total number of distances in a requests stream.

When the *percentage* is greater than a "high-water mark" threshold (more specifically, 45% in our prototype), the stream is considered as random, and the upcoming requests are redirected into SSDs. If the *percentage* is less than a "low-water mark" threshold when writing to SSDs (specifically 30% in our prototype), the stream is treated as sequential and the following requests are redirected to HDDs. The reason we choose two thresholds is to achieve better locality in data flushing and better performance in data buffering. It is unlikely that an access pattern constantly changes because many HPC applications present stable access patterns [25, 27], thus such a method is effective. Admittedly, these thresholds can be configured and fine tuned. These two specific threshold values were chosen based on the average RF percentage together with considering the throughput. Our extensive tests and empirical results indicate that the specific choice of these two thresholds are reasonable and desired as the default value. Additionally, note that the data redirector module and the redirection algorithm perform with the offsets and sizes of requests, instead of the data itself.



(a) *Offset distribution of segmented-contiguous accesses* (b) *Offset distribution of segmented-random accesses* (c) *Offset distribution of strided accesses*

Figure 3: Offset distribution of various access patterns with 16 processes

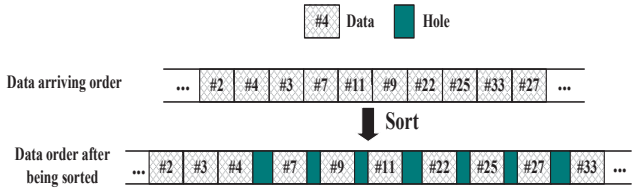


Figure 4: The sorting procedure in a requests stream

2.4 Pipeline with Double Buffer

In SSDUP, we divide the SSD into two regions, and both have the same size and are empty at the beginning. When an I/O request is forwarded to the SSD, SSDUP will first pick an empty region, assumed to be *Region1* (*R1* in Figure 6), to buffer the incoming requests. When *R1* is filled and there are other requests being forwarded to SSD, SSDUP then picks the next free region, assumed to be *Region2* (*R2*) to buffer the incoming requests. Meantime, *R1* starts to flush data to the HDD, as Figure 6 shows. The SSDUP handles *R1*'s flushing stage and *R2*'s writing stage in parallel. Thus, after *R2* is filled, the space of *R1* is also released. Both regions can be filled up in some cases (when the data amount of random writes is larger than the SSD capacity). As shown in Figure 6, the system keeps waiting until a region becomes empty. In this way, we can always have enough SSD space to handle incoming requests and we can achieve improved throughput with a small SSD capacity.

We demonstrate the effectiveness of our pipeline design with an analysis. Assuming the I/O phase is divided into n stages, and the size of data transmission in each stage is the same. The data transfer time to HDD is defined as T_{HDD}, and the data transfer time to SSD is defined as T_{SSD}. The capacity of the SSD is sufficient for m I/O stages ($m < n$) before the SSD is divided into two regions. Then the total I/O time is T_1 , calculated as Equation (2).

$$T_1 = m * T_{SSD} + (n - m) * T_{HDD} \tag{2}$$

In the SSDUP pipeline design, the SSD is divided into two regions. Each region can handle $m/2$ I/O stages. Assuming a write request is completed after being written to SSD, the time of the last I/O stage is calculated without the flushing time (the last flushing stage is performed in the computation phase and we make this assumption for simplicity in this model and analysis). The first $m/2$ stages are handled by *R1*. After that, $m/2$ stages (except the last stage) are handled as the pipelining progresses. Thus, the total number of I/O stages handled by the pipeline is $(n - 2 * m/2)$. In the pipeline, the time of each stage is determined by the maximum of the flushing time and buffering time. For example, in Figure 6, *R1* flushes data into the HDD, and the time spent in this stage is defined as T_f. In the meantime, *R2* buffers incoming data, and the time spent in this stage is defined as T_b. The total I/O time is defined as T_2, as calculated in Equation (3).

$$\begin{aligned} T_2 &= \frac{m}{2} * T_{SSD} + (n - 2 * \frac{m}{2}) * max\left\{T_f, T_b\right\} + \frac{m}{2} * T_{SSD} \\ &= m * T_{SSD} + (n - m) * max\left\{T_f, T_b\right\} \end{aligned} \tag{3}$$

In Equation 3, we can assume $T_b = T_{SSD}$ because buffering is in fact writing data to the SSD. In the flushing stage, data is written back to the HDD in a well-ordered fashion, and the flush time T_f is the time of sequentially accessing HDD. As writing data to the SSD is faster than writing to HDD usually, the total I/O time can be expressed as:

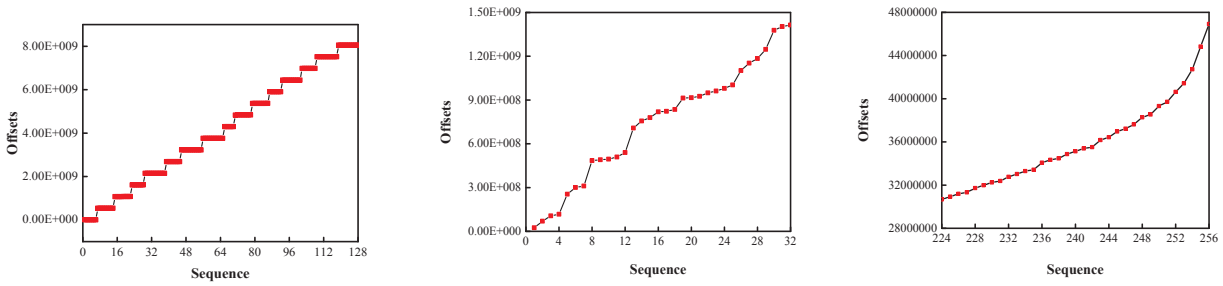
$$T_2 \approx m * T_{SSD} + (n - m) * T_f \tag{4}$$

Considering Equation (2) and Equation (4), we can see that the comparison between T_1 and T_2 is determined by the comparison between T_{HDD} and T_f. As long as T_f is less than T_{HDD}, T_2 will be less than T_1. Since the SSDUP writes sorted data into the HDD in the flushing stage, and random requests are forwarded to the SSD and rearranged into sequential requests, T_f is essentially the time to access HDD sequentially in our prototype. T_{HDD} is the time of accessing HDD without reordering requests. Since the SSDUP is designed to only buffer random requests, T_f is always smaller than T_{HDD}, and we can achieve an improved performance.

2.5 Buffer Management Using AVL Tree

In order to make better use of the SSD space, SSDUP is also designed to write data to the SSD in a log-structured way, which means appending data to the end of cache files. However, in the log-structured mode, the original sequence of file requests are disrupted when writing data to SSD. As shown in Figure 7, the original data {#1,#2,#3,...,#10} is requested in sequence {#1,#7,#8,...,#9}, thus data is written to the SSD in sequence {#1,#7,#8,...,#9}.

To recover the original data order and maintain the sequence, we need a mechanism to manage the metadata of cached data. Normally, a hash table is a desired choice because of $O(1)$ time for queries. In SSDUP, as we aim to quickly write the disrupted data back into the HDD, we have to re-sort the cached data. A typical



(a) *Offset distribution of segmented-contiguous accesses after being sorted* (b) *Offset distribution of segmented-random accesses after being sorted* (c) *Offset distribution of strided accesses after being sorted*

Figure 5: Offset distribution of various access patterns with 16 processes after being sorted

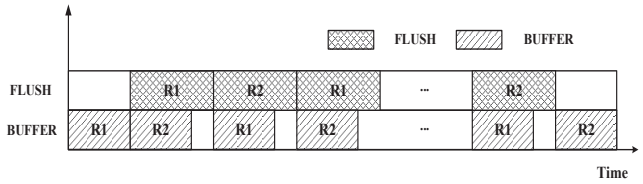


Figure 6: SSDUP pipeline design with double buffer

sorting algorithm like quicksort still takes $O(nlogn)$ time. Thus we choose to use the AVL tree to manage the cached data instead of a hash table. The AVL tree is a self-balancing binary search tree, and takes $O(logn)$ time for basic operations. SSDUP records the original metadata (including the original offset and size) and new metadata (including new offset and size), while writing data to the SSD. Both original and new metadata of the same data are stored in one leaf node. Each value requires 8 bytes, which adds up to 24 bytes for one node. The AVL tree requires about 3MB storage in our experiments (the file size is 40GB and request size is 256KB). Nodes are sorted based on the original data offset. As shown in Figure 7, in node (#2,*4), #2 represents the original offset and *4 represents the new offset in SSD. Each AVL tree is responsible for one file. In this way, data sequence can be maintained while buffering, which saves the time of an unnecessary sorting phase.

One significant advantage of using AVL trees in our design is that when data needs to be flushed to disk in a sequential order, SSDUP just needs to conduct an in-order traversal of the AVL tree. Note that when traversing the AVL tree, we can ensure the data is in its original sequence. But, as the way data is written changes the original data layout, it is actually random read to retrieve data from the SSD. Since an SSD has nearly zero seek delay, random read and sequential write are desired for SSD. Additionally, writing an SSD sequentially can avoid write amplification [23] when the SSD has high space utilization.

3 EXPERIMENTAL RESULTS AND ANALYSES

We have conducted extensive tests to validate the SSDUP design and to verify its performance potential. We present the results and analyses in this section.

3.1 Experimental Setup

The SSDUP is currently implemented in OrangeFS-2.9.3 and is integrated into the trove module. It retrieves parameter settings (e.g. thresholds, SSD directory, SSD capacity, etc.) from the configuration file. When the data flows arrive at the trove module, SSDUP intercepts the requests and performs optimizations as described in Section 2.

The experiments were conducted on a cluster consisting of 10 nodes, in which 8 nodes were used as compute nodes, and 2 nodes were used as I/O nodes. Each compute node is equipped with 16 Intel Xeon E5-2670 CPU processors, 64GB RAM, and a 300GB SATA hard drive. Each I/O node is equipped with 16 Intel Xeon E5-2670 CPU processors, 8GB RAM, a 300GB SATA hard drive (Toshiba model MBF2300RC) and a 300GB SSD (Intel model SSDSA2CW300G3). Each node runs CentOS with the Linux kernel version 2.6.32. The default I/O scheduler for HDD is CFQ with a queue size of 128 and the default I/O scheduler for SSD is NOOP [6]. All nodes are connected via a Gigabit Ethernet. MPICH-3.0.2 release [3], compiled with ROMIO, is installed on compute nodes. Two I/O nodes are installed with OrangeFS-2.9.3 and SSDUP. The default stripe size was set as 64KB.

3.2 Results with the IOR Benchmark and Analysis

We used IOR-2.10.3, a parallel file system benchmark developed at Lawrence Livermore National Laboratory, as one evaluation benchmark. IOR provides several APIs: HDF5, MPI-IO, and POSIX. The MPI-IO API was used in our experiments. IOR can also run with different access patterns. We performed three series of tests with IOR one by one, with three different access patterns to simulate different execution stages. The first series tests were conducted with a *segmented-contiguous* pattern. The second and the third series of tests were performed with a *strided* pattern and a *segmented-random* pattern (described in Section 2), respectively. The request size of each I/O was 256 KB and the first two series of tests performed write operations to a shared 16GB file. The last series of tests with the segmented-random access pattern wrote to a shared 8GB file because a completely random pattern is relatively rare. The default number of processes is 32 unless otherwise specified.

3.2.1 Performance with different numbers of processes. In the first set of evaluations, we ran the IOR instances with 8, 16, 32, 64, and 128 processes with segmented-contiguous, strided, and

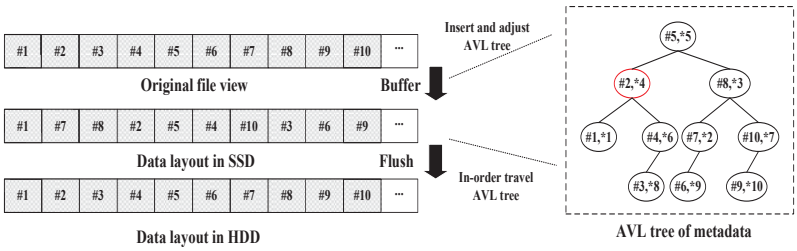


Figure 7: Metadata management with AVL tree

Table 1: Data Distribution with Different Number of Processes

No. of Processes	Ratio of Data in SSD(%)
8	19.2%
16	23.1%
32	58.8%
64	78.9%
128	81.3%

segmented-random patterns to understand the performance implications of the SSDUP. The results are shown in Figure 8. We present the average performance of three instances in one figure due to the page limit. Overall the SSDUP improved the system I/O throughput by 18%, 26%, 48%, 92%, and 99%, for the cases of 8, 16, 32, 64, and 128 processes, respectively. With 8 and 16 processes, both segmented-contiguous and strided patterns were considered sequential because of a small random factor in these cases. The performance was improved primarily because segmented-random requests were redirected to the SSD. The performance of the vanilla OrangeFS dropped in the case of 128 processes because inferences happened and each server had to serve more requests from different processes, which led to the accesses becoming more random. This is a case where SSDUP is beneficial and desired due to the increasing inference and random accesses. The SSDUP achieved much better performance (nearly doubled) with more requests redirected to SSD.

To better understand the performance improvement, we analyze the data amount redirected to SSD. As we can see from Table 1, only around 20% of data were redirected to the SSD due to little interference in the cases of 8 and 16 processes. However, the ratio of data identified as randomly accessed and redirected to the SSD became 58.8% and 78.9%, in the cases of 32 and 64 processes, respectively, due to increased interferences and random factor. In the case of 128 processes, almost 81.3% of data accessed was identified as random (and redirected to the SSD) because the disk head significantly suffers from moving back and forth. In addition, the experimental results and observations were stable, and higher concurrency always led to more randomness because more interferences happened [31]. In other words, as the ratio of compute servers to storage servers increases, each storage server has to serve more clients. These clients compete for the disk head, which makes random accesses even worse. Based on this trend, we expect SSDUP will achieve stable performance benefit when the scale/concurrency increases.

3.2.2 Performance with different CFQ queue sizes. We also measured the impact of the CFQ queue size on the SSDUP performance. We performed three groups of experiments with the size of the CFQ queue as 32, 128, and 512. The IOR instances ran with 32 processes. SSDUP achieved 90.9%, 48%, and 15% performance improvement, respectively, as shown in Figure 9. Note that the default size of the CFQ queue is 128. When the size was changed to 32, CFQ became more sensitive to concurrent accesses with interferences, which resulted in the system I/O bandwidth dropping to 113 MB/s. We also adjusted the length of requests stream relatively to test this, so the percentage of random factor became larger, which made more data identified as random accesses and redirected to SSD. The ratio of data in the SSD became 87.2%, which is larger the ratio in the case of 128 entries as the queue size. When the CFQ queue size became 512, the aggregate throughput of the original system increased to 164 MB/s. With a larger queue size, the CFQ scheduler has more opportunities to merge adjacent requests and achieves better locality. In this scenario, SSDUP only achieved 15% improvement because random access was relatively insensitive to larger queue size [29], and only a small part of segmented-random data was redirected to SSD.

3.2.3 Performance with different SSD capacity. We have also studied the impact of the SSD buffer size on the I/O throughput. For this purpose, we also compared the SSDUP against two other solutions: one is a common burst buffer implementation that buffers all data in the SSD as long as there is enough space, and the other one is the SSDUP without pipeline, which buffers identified data, but not in a pipeline fashion. The performance of the BB solution is plotted as black squares in Figure 10. The performance of the other solution, SSDUP without pipeline (SSDUP-WOP), is plotted as red circles. The performance of SSDUP is indicated with green triangles.

We ran the IOR benchmark in the same way as discussed above and 32 processes were used. The capacity of the SSD was varied from 0GB to 40GB. As shown in Figure 10, BB achieved nearly no improvement when SSD sizes were 8GB and 16GB, but achieved 19%, 29%, and 65% throughput speedup with SSD size being 24GB, 32GB, and 40GB, respectively. This is because that the first 16GB data is sequentially accessed, BB hardly benefits from buffering these data in the SSD. Only when SSD capacity was sufficient, BB delivered a better I/O throughput; otherwise, the throughput was dominated by the HDD. SSDUP-WOP achieved 13%, 31%, 65%, 66%, and 65% improvements with different SSD capacity sizes. SSDUP-WOP identifies random accesses and buffers them in SSD, but not in a pipeline fashion. It achieved better I/O throughput than BB when SSD capacity was smaller than the total data size. After the SSD size reached 24GB, the I/O throughput no longer increased because 24GB was large enough to hold all random accessed data.

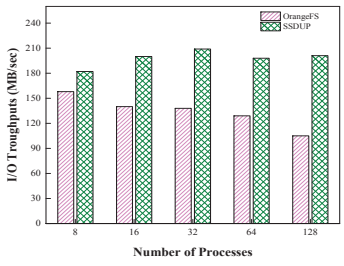


Figure 8: I/O throughput with different numbers of processes for the IOR benchmark, for the original OrangeFS and SSDUP

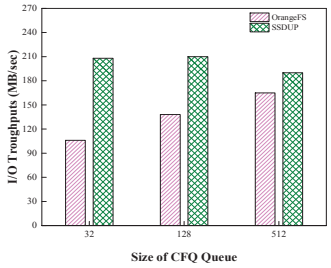


Figure 9: I/O throughput with different sizes of CFQ queue for the IOR benchmark, for the original OrangeFS and SSDUP

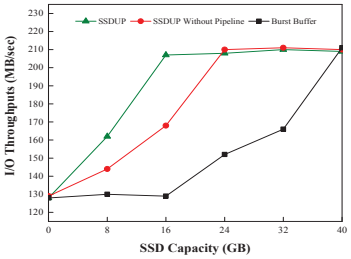


Figure 10: I/O throughput with different SSD capacity sizes for Burst Buffer, SSDUP Without Pipeline, and SSDUP

As a comparison, SSDUP achieved 25% improvement with SSD capacity being 8GB. It improved the throughput by 64% when the SSD capacity was larger than 16GB. Overall SSDUP outperformed the other two strategies when the SSD size was 8GB (20% of total data size). The sustained I/O throughput was 23% better than the case with BB and 12% better than the case with SSDUP-WOP because SSDUP not only selected critical data into SSD, but also flushed data in a pipeline to make the SSD have enough space to buffer incoming data. In addition, SSDUP reached a peak throughput by using a 16GB SSD (40% of total data size), while SSDUP-WOP and BB required 24GB (60% of total data size) and 40GB (100% of total data size), respectively, to reach the peak throughput. When the SSD capacity was larger than 40GB, the sustained I/O bandwidth of BB was determined by the bandwidth of writing to the SSD. The main purpose of SSDUP is to use small SSD capacity to improve the system performance, while reducing the cost at the same time. That is why we do not conduct the evaluation beyond 40GB. These test results have proven that SSDUP is a more promising solution that uses a relatively small SSD capacity to achieve desired performance.

3.2.4 Performance with different computing time. We have studied the impact of computing time on the system performance as well. We ran two same IOR instances sequentially and adjusted the time from 0s to 10s between these two instances. Each IOR was executed with segmented-random access pattern and produced a 2GB shared file. We performed five groups of experiments with SSD size varying from 0% to 100% of the total file size. The computing time is not counted when calculating the throughput.

As shown in Figure 11, the I/O throughput was improved when SSD size increased (as we have seen from previous sets of evaluations). The throughput increased too when the computing time increased; however, the improvements were little, with 4.6%, 1.8%, 24%, and 10%, for 100%, 75%, 50%, and 25% of SSD size, respectively. From these results we can draw a conclusion that SSDUP does not rely on the computing time much to overlap with the flushing time to achieve its benefits. However, the overall system performance (including the original system) can be tuned with a appropriate SSD capacity and computing time. We will study the best configuration of SSD capacity with real applications that have different percentages of computing time.

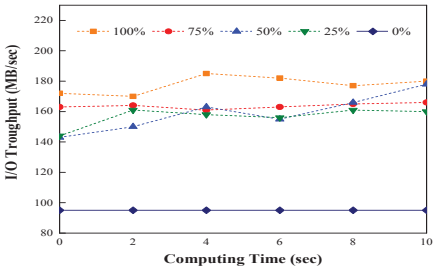


Figure 11: I/O throughput with different SSD capacity sizes and different computing time for SSDUP

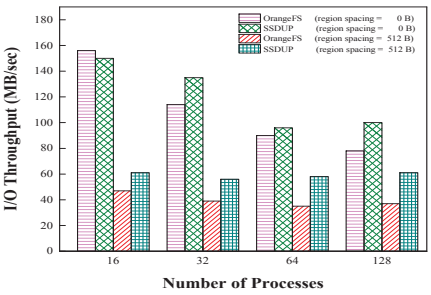


Figure 12: I/O throughput with different numbers of processes and different region spacing for the HPIO benchmark, for the original OrangeFS and SSDUP

3.3 Results with HPIO Benchmark and Analysis

We also used the HPIO benchmark [14] to evaluate and compare the performance of OrangeFS and SSDUP. HPIO is a benchmark developed at Northwestern University and has been widely used to evaluate noncontiguous I/O performance. Similar to IOR, HPIO can be run in different access patterns too. Three parameters were used in this test: region size, region count, and region spacing. A region is a piece of contiguous data in a file that will be accessed by a process. Region count is the number of regions accessed by

one process. Region spacing is the distance between two adjacent regions. We set the region size as 64KB and varied the region count together with the number of processes to keep the file size around 2GB. We also set region spacing to vary from 0B to 512B to evaluate both contiguous and noncontiguous access patterns. The total SSD capacity was half of the file size.

We ran the HPIO benchmark with 16, 32, 64, 128 processes, respectively. As shown in Figure 12, when region spacing was 0B, SSDUP had similar performance with that of OrangeFS when the process number was 16. This is because there is no data identified as random. When the process number increased from 32 to 128, SSDUP improved the throughput by 18%, 6.7%, and 28% respectively. When the process number was 32, the throughput of OrangeFS dropped because of interferences, which actually left SSDUP to have more opportunities to optimize. When the process number was 64, more data was identified as randomly accessed and redirected to the SSD. The specific ratio was 35%, which was 70% of the total SSD capacity. However, in SSDUP, when the amount of random data is more than half of the SSD capacity, the flushing procedure starts and the throughput is determined by the performance of sequentially accessing the HDD. Thus, when the amount of random data varied from 50% to 100% of the SSD capacity, the throughput achieved by SSDUP remained roughly the same. On the other hand, the throughput achieved by OrangeFS were quite different. This also explained why the throughput improvement in the case of 64 processes was lower than that of 128 processes.

When region spacing was 512B, the throughput of OrangeFS was less than 50 MB/s because all accesses were noncontiguous. In these cases, SSDUP achieved 30%, 43%, 66%, and 65% improvements with different numbers of processes. However, the overall system performance was not as good as the case with 0B region spacing. The reason is that the data layout in the HDD is still noncontiguous, and the flushing stage writes noncontiguous data sequentially to the HDD. Nevertheless, SSDUP delivered better performance than the OrangeFS.

3.4 Overhead Analysis

The overhead of SSDUP primarily comes from two aspects: the cost of grouping and sorting requests of a requests stream (named *group cost*), and the cost of keeping AVL tree balanced and in-order traveling the AVL tree (named *AVL cost*). We also used the IOR benchmark to study and analyze these overheads. The total size of accessed data was 2GB, and the request size varied from 32KB to 512KB. The SSD capacity was set as 2GB. The IOR benchmark was executed in the segmented-random pattern, and all requests were sent to the SSD.

The measured system overhead costs are 0.4% (in the case of 512KB requests) and 1.9% (in the case of 32KB requests) of the total execution time, which is minor and can be ignored compared to the performance gain. As shown in Table 2, these two types of overhead increased when the request size became smaller. This is because that these operations were based on every request, and the number of I/O requests became smaller when the request size became larger. The overhead in the case of 128KB and 64KB requests were close, because the request was striped across two data servers when request was larger than the default stripe size.

These experimental results have shown that the SSDUP improved the write performance by using 40% of the total SSD space (the ratio of random accesses is 20%) for the IOR benchmark, and by using 50% of total SSD space (the ratio of random accesses is around 20% to 100%) for the HPIO benchmark. The overhead was minor and negligible, less than 2% of the total execution time. The average ratio of random accesses in HPC applications has been reported as around 50% [32], which is consistent with the ratio setting in our experiments. Compared with burst buffer, SSDUP can save about 50% cost. In addition, our approach does not introduce more writes to SSD. Instead, SSDUP only buffers random I/O requests and reduces writes to SSD. As a consequence, SSDUP can extend the lifetime of SSD compared with conventional burst buffer design. Since IOR and HPIO are typical benchmarks that represent the common access patterns (segmented-contiguous, segmented-random, strided, and noncontiguous) of scientific applications, we believe that SSDUP is beneficial for real workloads.

Table 2: System Overhead

Request Size	Total Time	Group Cost	AVL Cost
32 KB	12.7 s	184 ms	60.2 ms
64 KB	11.6 s	126 ms	32.8 ms
128 KB	12.8 s	135 ms	35.3 ms
256 KB	11.8 s	68 ms	17.4 ms
512 KB	11.7 s	41 ms	7.9 ms

4 RELATED WORK

Many research studies have been conducted to improve the performance of I/O system for HPC applications [24, 27, 30]. In this section, we focus on previous work in three different areas: addressing random access problems caused by accessing hard disks concurrently, recognizing critical data and access patterns, and storage systems integrated with SSDs.

4.1 Concurrent Access Problem

Wang et. al. introduced an IBCS method [26], based on two-phase I/O, to reorganize the data transferring order in the shuffle stage to keep a one-to-one object storage target access pattern [15] in each iteration, which avoids multiple processes competing for one disk head. Because only one process accesses a disk at a time, the system throughput is significantly improved due to reduced disk seek cost. Chen et. al. [13] analyzed the gap between data layout at the client side and data layout in parallel file systems. These gaps cause multiple processes to access hard disks in an interfering way. Zhang et. al. [28] reported that the file striping strategy of parallel file systems would break individual program's locality when multiple programs were concurrently served by data servers. Zhang et. al. [31] studied a common pattern of HPC applications, that is, individual process's offset is continuous, but multiple processes may compete for one disk head when processes carrying requests concurrently, which makes the disk head move back and forth. They propose a data replication method that copies the same process's data to the same I/O node, and ensures that each I/O node serves as few processes as possible. All these approaches can be effective in addressing concurrent access problems. In this study, we have introduced a new and different solution that utilizes new SSD devices and redirects random accesses to the SSD to avoid costly HDD accesses.

4.2 Recognizing Critical Data and Access Patterns

S4D-Cache [16] introduces a new technology to identify performance critical data, to select larger cost data and redirect data to the SSDs. S4D-Cache migrates data between SSDs and HDDs according the temporal locality which keeps data with strong temporal locality in the SSDs longer. Byna's prefetching technique [12] prefetches data into memory by detecting a stable local access pattern. Both S4D-Cache and I/O prefetching analyze access patterns from the perspective of a single process. However, multiple processes that access the same server can cause competition and random accesses. The more processes involved, the more serious randomness it can be. It is hard to calculate the accurate cost from a single process's point of view. Our proposed method traces the access pattern in a global view that can measure random accesses caused by both application's native behavior and multi-processes's competitions.

4.3 Storage Systems Integrated with SSDs

Burst buffer [19] inserts I/O forwarding nodes equipped with SSDs between client and storage nodes, so data is quickly flushed from memory to persistent store and returns to the computation phase as soon as possible. ITtransformer [29] uses SSDs as an extension of the memory queue to improve I/O throughput in high concurrency scenarios. IBridge [32] treats SSD as a local cache of HDD and only small random and unaligned requests are sent to the SSD. SieveStore [22] provides an ensemble-level and selective caching mechanism which uses SSD to cache the most popular blocks. Liu [17] designs a low latency storage system by lifting part of data up to SSD to cover the seek time of hard disk.

Our system differs from other SSD-HDD systems in two fundamental ways. First, by tracing I/O from a global point of view, we can detect random accesses caused by various situations, which makes I/O accesses more efficient. Second, by using a pipeline and AVL tree, we can make use of SSD space in a more efficient way. The end result is using less space to buffer more requests and without relying on unpredictable computation time to overlap with the flushing stage.

5 CONCLUSION

In this paper, we have proposed a scheme, called SSDUP (a traffic-aware SSD burst buffer), to improve the burst buffer. We carefully studied three common access patterns in HPC environment and the proposed I/O-traffic detection method was able to identify random/irregular I/O accesses caused by various access patterns. The SSDUP is designed to buffer selected random data into SSD based on the random factor and flush well-ordered accesses to HDD in a pipelined way. In addition, an AVL tree structure is used to manage the metadata of buffered data, which maintains the data sequence. With these efforts, SSDUP can achieve the desired I/O performance by using small SSD space.

We have implemented a prototype of SSDUP in OrangeFS to validate the design and verify its potential performance benefits. Note that the methodology introduced in this paper is general though and can be applicable to other file systems. The performance was measured using two widely-used benchmarks, IOR and HPIO. The experimental results have shown that SSDUP improved the system write performance by up to 1X, and by 50% on average with very minor and negligible overhead. In the near future, we plan to further explore more characteristics of SSD and well leverage it for HPC storage systems. We also plan to port our design into Lustre [5] file system for broader adoption.

ACKNOWLEDGMENT

We thank the reviewers and our shepherd, Saugata Ghose, for the helpful comments that have significantly improved the paper. Moreover, we thank Ligang He, and Xiaosong Ma for their feedback and insights. This work is supported by the National Key Research and Development Plan (No. 2016YFB0200502), NSFC (No. 61370104, 61433019, U1435217), and the Outstanding Youth Foundation of Hubei Province (No. 2016CFA032).

REFERENCES

[1] *Boost application performance using asynchronous I/O.* https://www.ibm.com/developerworks/library/l-async.
[2] *Completely fair queueing (CFQ) scheduler.* http://en.wikipedia.org/wiki/CFQ.
[3] *High-performance and widely portable implementation of the Message Passing Interface (MPI) standard.* http://www.mpich.org.
[4] *Interleaved Or Random (IOR) Benchmarks.* https://github.com/LLNL/ior.
[5] *Lustre File System.* http://wiki.lustre.org/Main_Page.
[6] *Noop scheduler.* http://en.wikipedia.org/wiki/Noop.
[7] *Orange File System.* http://www.orangefs.org.
[8] *PBS.* http://pbspro.org.
[9] *Slurm Workload Manager.* http://slurm.schedmd.com.
[10] *Sun Grid Engine.* https://en.wikipedia.org/wiki/Oracle_Grid_Engine.
[11] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. 2009. PLFS: a checkpoint filesystem for parallel applications. In *SC*. 21.
[12] S. Byna, Y. Chen, X.-H. Sun, R. Thakur, and W. Gropp. 2008. Parallel I/O prefetching using MPI file caching and I/O signatures. In *SC*. 44.
[13] Y. Chen, X.-H. Sun, R. Thakur, H. Song, and H. Jin. 2010. Improving parallel I/O performance with data layout awareness. In *CLUSTER*. 302–311.
[14] A. Ching, A. Choudhary, W.-K. Liao, L. Ward, and N. Pundit. 2006. Evaluating I/O characteristics and methods for storing structured scientific data. In *IPDPS*. 49–49.
[15] P. M. Dickens and J. Logan. 2009. Y-lib: a user level library to increase the performance of MPI-IO in a lustre file system environment. In *HPDC*. 31–38.
[16] S. He, X.-H. Sun, and B. Feng. 2014. S4D-Cache: Smart selective SSD cache for parallel I/O systems. In *ICDCS*. 514–523.
[17] D. Huang, X. Zhang, W. Shi, M. Zheng, S. Jiang, and F. Qin. 2013. LiU: Hiding disk access latency for HPC applications with a new SSD-enabled data layout. In *MASCOTS*. 111–120.
[18] H. Huang, W. Hung, and K. Shin. 2005. FS2: dynamic data replication in free disk space for improving disk performance and energy consumption. In *SOSP*. 263–276.
[19] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn. 2012. On the role of burst buffers in leadership-class storage systems. In *MSST*. 1–11.
[20] Y. Liu, R. Gunasekaran, X. Ma, and S. Vazhkudai. 2014. Automatic identification of application I/O signatures from noisy server-side traces. In *FAST*. 213–228.
[21] H. Luu, M. Winslett, W. Gropp, R. Ross, P. Carns, K. Harms, M. Prabhat, S. Byna, and Y. Yao. 2015. A multiplatform study of I/O behavior on petascale supercomputers. In *HPDC*. 33–44.
[22] T. Pritchett and M. Thottethodi. 2010. SieveStore: a highly-selective, ensemble-level disk cache for cost-performance. In *ISCA*. 163–174.
[23] L. Tang, Q. Huang, W. Lloyd, S. Kumar, and K. Li. 2015. RIPQ: Advanced photo caching on flash for Facebook. In *FAST*. 373–386.
[24] R. Thakur, W. Gropp, and E. Lusk. 1999. Data sieving and collective I/O in ROMIO. In *Frontiers*. 182–189.
[25] Y. Wang and D. Kaeli. 2003. Profile-guided I/O partitioning. In *ICS*. 252–260.
[26] Z. Wang, X. Shi, H. Jin, S. Wu, and Y. Chen. 2014. Iteration based collective I/O strategy for parallel I/O systems. In *CCGrid*. 287–294.
[27] Y. Yin, J. Li, J. He, X.-H. Sun, and R. Thakur. 2013. Pattern-direct and layout-aware replication scheme for parallel I/O systems. In *IPDPS*. 345–356.
[28] X. Zhang, K. Davis, and S. Jiang. 2010. IOrchestrator: Improving the performance of multi-node I/O Systems via inter-server coordination. In *SC*. 1–11.
[29] X. Zhang, K. Davis, and S. Jiang. 2012. iTransformer: Using SSD to improve disk scheduling for high-performance I/O. In *IPDPS*. 715–726.
[30] X. Zhang, K. Davis, and S. Jiang. 2012. Opportunistic data-driven execution of parallel programs for efficient I/O services. In *IPDPS*. 330–341.
[31] X. Zhang and S. Jiang. 2010. InterferenceRemoval: removing interference of disk access for MPI programs through data replication. In *ICS*. 223–232.
[32] X. Zhang, K. Liu, K. Davis, and S. Jiang. 2013. iBridge: Improving unaligned parallel file access with solid-state drives. In *IPDPS*. 381–392.