Optimizing the SSD Burst Buffer by Traffic Detection

XUANHUA SHI and WEI LIU, Huazhong University of Science and Technology

LIGANG HE, University of Warwick

HAI JIN and MING LI, Huazhong University of Science and Technology

YONG CHEN, Texas Tech University

Currently, HPC storage systems still use *hard disk drive* (HDD) as their dominant storage device. *Solid state drive* (SSD) is widely deployed as the buffer to HDDs. *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 phase.

In this article, we propose a scheme, called SSDUP+.¹ SSDUP+ aims to improve the burst buffer by addressing the above limitations. First, to reduce the demand for the SSD capacity, we develop a novel method to detect and quantify the data randomness in the write traffic. Further, an adaptive algorithm is proposed to classify the random writes dynamically. By doing so, much less SSD capacity is required to achieve the similar performance as other burst buffer schemes. Next, to overcome the difficulty of perfectly overlapping the computation phase and the flushing phase, we propose a pipeline mechanism for the SSD buffer, in which data buffering and flushing are performed in pipeline. In addition, to improve the I/O throughput, we adopt a traffic-aware flushing strategy to reduce the I/O interference in HDD. Finally, to further improve the performance of buffering random writes in SSD, SSDUP+ transforms the random writes to sequential writes in SSD by storing the data with a log structure. Further, SSDUP+ uses the AVL tree structure to store the sequence information of the data.

We have implemented a prototype of SSDUP+ based on OrangeFS and conducted extensive experiments. The experimental results show that our proposed SSDUP+ can save an average of 50% SSD space while delivering almost the same performance as other common burst buffer schemes. In addition, SSDUP+ can save about 20% SSD space compared with the previous version of this work, SSDUP, while achieving 20–30% higher I/O throughput than SSDUP.

CCS Concepts: • **Information systems** → **Information storage systems**;

Additional Key Words and Phrases: High-performance computing, burst buffer, hybrid storage system, solid state drive

¹The previous version of this scheme, called SSDUP, has been published in ICS2017 [34].

This work is supported by the National Key Research and Development Plan (No. 2017YFC0803700), NSFC (No. 61772218, 61832006), and the Outstanding Youth Foundation of Hubei Province (No. 2016CFA032).

Authors' addresses: X. Shi, W. Liu, H. Jin, and M. Li, National Engineering Research Center for Big Data Technology and System, Service Computing Technology and System Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China; emails: {xhshi, cccloude, hjin, limingcs}@hust.edu.cn; L. He, Department of Computer Science, University of Warwick, UK; email: ligang.he@warwick.ac.uk; Y. Chen, Department of Computer Science, Texas Tech University, Texas, USA; email: yong.chen@ttu.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

1544-3566/2020/03-ART8

<https://doi.org/10.1145/3377705>

ACM Reference format:

Xuanhua Shi, Wei Liu, Ligang He, Hai Jin, Ming Li, and Yong Chen. 2020. Optimizing the SSD Burst Buffer by Traffic Detection. *ACM Trans. Archit. Code Optim.* 17, 1, Article 8 (March 2020), 26 pages. <https://doi.org/10.1145/3377705>

1 INTRODUCTION

While *high-performance computing* (HPC) systems are moving toward 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 [25]. 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: They suffer from 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) [7, 13, 14, 43, 50]. 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 using the SSDs to buffer the data streams between HDDs and computing nodes, which has been adopted by many production supercomputers (e.g., Sunway TaihuLight [11] and Tianhe-2 [21]).

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, and so on, which mainly perform the numerical simulations. Moreover, the write requests issued by these applications are bursty, because the applications often generate and store a large amount of intermediate results [22, 23]. Further, 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 generates 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.

To address the above issue, *Burst Buffer* (BB) [22] 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 SSD buffer, which effectively absorbs the bursty writes of the application, since SSD offers very low latency. While the data in the SSD buffer are flushed to HDDs, the HPC system continues to process next computation phase. 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: writing the future data to HDD directly or blocking the incoming I/O requests until BB becomes available again. In both options, the overall I/O performance will be dominated by HDD, which violates the purpose of using burst buffer. This condition means that the capacity of the SSD buffer has to be no less than the maximum size of the data generated by a computation phase of an application. However, an HPC application can access hundreds of terabytes or even tens of

petabytes of data. For instance, the total data transmission of the Earth1 application in the Mira supercomputer [28] has reached about 10 PB [25]. However, 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 [25]. 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) SSDs 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 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 varies greatly subject to many factors. On the other hand, the HPC platforms use job scheduling tools, such as PBS [18], SGE [27], and slurm [19] 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 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 a traffic-aware SSD burst buffer scheme, called the *SSDUP+*², 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, to improve the I/O performance given the same SSD capacity).

First, to reduce the demand for the SSD buffer capacity, we select only a proportion of data to write to SSD, while the remaining data are written to HDDs directly without sacrificing 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) These multiple processes are in the scope of the same application. Therefore, it can only detect the data access pattern within a single application, not from different applications. (ii) The method can only detect the data access pattern related to the I/O operation that is being invoked currently, not across different I/O invocations. Our I/O-traffic detection component is located at the side of the storage server. In our *server-side* detection method, a novel metric, termed *random factor*, is proposed 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. Furthermore, an adaptive algorithm is proposed to dynamically adjust the threshold of random factor according to the feature of the workload. The threshold is used to determine whether the current requests are buffered in SSD or written to HDD directly.

Second, 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 the SSD buffer will have been flushed and therefore ready to accommodate new data even if the amount of data produced by the computation phase is larger than the capacity of SSD buffer. Therefore, when the new computation phase produces new data (either due

²The source code is available at <https://github.com/CGCL-codes/SSDUPplus>.

Fig. 1. The architecture of SSDUP+.

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 and, consequently, the I/O performance can be significantly improved. Moreover, when multiple applications access a storage node concurrently, we find that when the data from one application are being flushed from SSD to HDD, other applications may be writing their data to HDD at the same time, which will cause intense I/O interference in HDD and, consequently, lead to performance degradation. To address this issue, we propose a traffic-aware flushing strategy in SSDUP+. The pipeline module of SSDUP+ dynamically analyzes the current workload and decides a good timing for performing the flushing operation according to the current workload in HDD.

Finally, we only buffer the random writes to SSD. However, SSD is desired for sequential writes due to the features of SSD writing. To improve the performance of buffering random writes in SSD, we convert 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 drawback of such a log structure is that the original sequence of file requests is lost. 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.

The rest of the article is organized as follows. Section 2 describes the detailed design of SSDUP+. Evaluations and analyses are presented in Section 3. The related work in recent years and their relation are discussed in Section 4. Finally, this article is concluded in Section 5.

2 SSDUP+

In this section, we describe the design of SSDUP+ (a traffic-aware SSD burst buffer). 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/O requests to different devices, a *pipeline* component that orders and schedules I/O requests in pipeline efficiently, and an *AVL tree management* that manages the buffered data and maintains the data sequence to order the data. In this section, we first overview SSDUP+ and then present the details of each component in SSDUP+, including the designed algorithms and the data structures.

2.1 Overview of SSDUP+

Although SSDUP+ is implemented as a part of OrangeFS [31] (the latest version of PVFS2) in this work, the methodology used to design SSDUP+ is generic. The algorithms and the data structures designed in SSDUP+ can also be applied to other file systems.

OrangeFS adopts a client/server model. It has been widely used not only as an experimental platform but also a production platform in the HPC area. In OrangeFS, 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 the data are located.

Figure 1 shows the architecture of SSDUP+. SSDUP+ resides in each I/O node and integrates into the *pvfs2-server* daemon. SSDUP+ in different I/O nodes does not need to communicate with each other. When the requests arrive at an I/O node, SSDUP+ records the requests as sequences of accesses called *request stream*. The length of a sequence (e.g., 128) is a system parameter and can be configured by the administrator. Note that a request stream is simply a logical grouping of I/O requests used by SSDUP+ to identify the randomness of the I/O requests. We do not manipulate the I/O requests physically. Next, SSDUP+ reshapes the request stream with the help of four components. The *random access detector* is responsible for determining whether the subsequent request stream should be written to HDD or SSD. The *data redirector* is responsible for sending the data to the dedicated devices based on the result of the random access detector. The *pipeline* component handles the data buffering stage and the flushing stage and maintains the sufficient SSD space for the incoming requests. The *AVL tree management* component manages the metadata of the buffered data and sorts the random data for better flushing performance. In SSDUP+, when the data are written to SSD, they are written in the order they arrive, i.e., SSDUP+ appends newly arriving data to the end of the cached data. The data are not organized into sequential ones. The relation between the write order of a data block and its logical order in the file is recorded in the AVL tree. When the data are flushed from SSD to HDD, the data blocks are read from SSD (and written to HDD) in their logical order by looking up the AVL tree. Although reading the data blocks from SSD in their logical order is random, it is acceptable as random reading from SSD has low latency.

There exist different architectures for implementing the burst buffer. Two typical architectures are I/O server-oriented architecture [20, 38] and computing node-oriented architecture [40, 51]. In the I/O server-oriented architecture, the burst buffer system is deployed as an I/O server, which accepts the I/O requests from clients (e.g., computing nodes), while in the computing node-oriented architecture, the burst buffer mechanism is implemented inside the local computing node, handling the I/O requests issued by the local node. Our SSDUP+ is designed mainly for the I/O server-oriented burst buffer architecture. However, the core techniques in SSDUP+, including detection of random data, the traffic-aware pipeline mechanism, and log-structure metadata management, can also be applied to the local burst buffer mechanism in computing nodes.

The details of these four components are explained next.

2.2 Detecting Random Access

For scientific applications in HPC, the I/O stage and the computing stage are often performed alternately. There may be significant differences in the distribution of I/O stages and computing stages. But the I/O pattern in each I/O stage is largely the same. Parallel I/O in each I/O stage mainly includes parameters such as I/O request size, I/O concurrency, file format, and file layout strategies. The file layout strategies include continuous, strided, and random strategies. It is well understood that random accesses can cause significant seek delay due to the need to move the disk head to the correct position before transmitting any data. To understand the relationship between

Fig. 2. The I/O throughput of IOR with different access patterns and different number of processes.

the access pattern in each I/O stage and the disk latency, we carried out a series of experiments using IOR [24], a benchmark to test the performance of parallel file systems, which can be used to effectively simulate the behavior of I/O stages in HPC applications under various conditions. Specifically, three different access patterns were tested: *segmented-contiguous*, *segmented-random*, and *strided* [3]. The total data size tested was 16 GB. The size of each I/O request was 256 KB. The number of processes varied from 4 to 128. Ten nodes were used in the experiments, 8 of which were compute nodes and 2 of which are I/O nodes. OrangeFS 2.9.3 were installed on both I/O nodes. Other details of the experimental platform are given in Section 3.

Note that 256 KB is not the file size but the data size of an I/O request. In parallel file systems in HPC, a file is typically striped and stored across multiple storage nodes, and when accessing the file, the data blocks in different storage nodes are accessed in parallel through parallel I/O. An I/O request accesses a data block. In this scenario, it is unnecessary to consider the file size. Moreover, 256 KB is only the data size of the I/O request used in our experiments, which can actually be of any value determined by the parallel file system.

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

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

Strided: In this pattern, the number of processes is n . In the i th iteration, process with ID j issues an I/O request to access the data at the offset $i * n + j$.

The experimental results are presented in Figure 2. It can be observed that the throughput increases first and then drops with the increase in the number of processes in both segmented-contiguous and strided patterns. The reason for this result is because OrangeFS uses *asynchronous I/O* (AIO) as its default I/O method [31]. AIO [17] can improve its throughput when the number of I/O participants grows. This is because the *Completely Fair Queuing* (CFQ) [1] scheduler tends to queue more requests so as to achieve better spatial locality by sorting and merging these requests. When the number of processes increases further (from 16/32 processes to 128 processes), the throughput of the segmented-contiguous accesses drops from 218 MB/s to 150 MB/s, while the throughput of the strided accesses drops from 164 MB/s to 107 MB/s, amounting to 31% and 34% degradation, respectively. The reason for this is because the size of the queue in the CFQ scheduler is limited. When the number of I/O processes increases further, CFQ will not be able to achieve even higher locality.

Fig. 3. Offset distribution of various access patterns.

The throughput of the segmented-random accesses remains at around 95 MB/s, because the CFQ scheduler can hardly merge any requests when the offsets are non-consecutive.

To verify these observations, we also traced the offsets of these different access patterns. Figure 3 shows the offsets for a single application in three access pattern and the offsets for two applications that are segmented-contiguous $\times$ segmented-random mixed loads. In this experiment, 65,536 ($16 \text{ GB}/256 \text{ KB} = 65,536$) requests were issued by 16 processes in total for every single application, and 65,536 ($8 \text{ GB}/256 \text{ KB} + 8 \text{ GB}/256 \text{ KB} = 65,536$) requests were issued by 32 processes in total for mixed loads. Figure 3(a) depicts the offset distribution of the first 128 requests. As shown in the figure, the offsets of the segmented-contiguous accesses are regular, whereas the segmented-random accesses show the completely random offsets (Figure 3(a) and Figure 3(b)). The offsets of the strided accesses (Figure 3(c)) are compact with slight fluctuations. Besides, the offsets shown in Figure 3(d) seem rather random.

Similarly to what we did for CFQ, we also used 128 requests as a unit block and sorted the offsets in every block of 128 requests. After sorting the offsets, the offsets of the request blocks manifested a much better order. If the offsets of the requests in a block are adjacent, then they will be merged and the seek distance is regarded zero, although the offsets of the consecutive requests are not contiguous. The sorted access order is beneficial, because without the sorting, the disk head has to move back and forth from the offset of one request to that of next consecutive request, which leads to the longer access delay.

We introduce a notion of *random factor* (RF) in SSDUP+ to indicate the number of disk head movements. After the offsets are sorted, if the distance between their offsets equals the request size, we regard two requests as the sequential requests and assign the random factor to be 0, which represents the fact that the disk head does not need to seek the disk location for next request. Otherwise, the requests are considered random and the random factor is 1, meaning that the disk head needs to move once. We calculate the sum of random factors, denoted by S , in every request stream, using Equation (1), where N is the number of requests in a request stream,

Fig. 4. The sorting procedure in a requests stream.

Fig. 5. Offset distribution of various access patterns with 16 processes after sorting.

$$S = \sum_{i=1}^{N-1} RF_i. \quad (1)$$

Figure 4 illustrates the sorting procedure for a request stream. Requests in a stream may arrive out of order because of the random or irregular accesses issued by the processes and also the competitions among processes. As we have discussed above, in this example, the random factor between requested data item #2 and #3 is 0, because after sorting, the offset distance between these two requests equals to the request size. However, the random factor between data item #4 and data item #7 is 1. Note that the random factor is defined based on logical address instead of physical address. Although using the logical address to indicate the movement of disk head is not 100% accurate, the disk seek time is linearly related to the logical address distance in most cases [15]. In addition, there is no need to distinguish among different applications when sorting the logical addresses of the requests. This is because in a single request stream, there is little correlation between the logical addresses of requests for different applications.

Figure 5 shows the distribution of offsets after the requests are sorted. In these figures, a solid black line connecting two discrete red lines (or dots) represents a disk seeking movement. To make

these figures more readable, we only draw 32 offsets of 128 for both segmented-random and strided patterns. We draw 64 offsets of 128 for the mixed load.

In a request stream, there are 128 requests, and therefore the maximum number of disk seeking that the disk head has to perform when serving the request stream is 127 (i.e., $128 - 1$). As can be seen from Figure 5 (the random factor equals the total number of discrete lines (or dots) in the figure minus 1), the random factor of segmented-contiguous accesses becomes 15 after sorting, which means that when serving the requests the disk head needs to move 15 times, accounting for 11% of the maximum 127 movements.

We introduce another notion, called *random percentage*, to represent the level of randomness in a request stream. The random percentage in Figure 5 is 11%. The total random factor of the segmented-random accesses is 127, which means the random percentage is 100% (i.e., there is no reduction of disk movement at all). In the strided accesses, the total random factor is 57, and the random percentage is 45%.

The situation becomes more complicated under the mixed load. As shown in Figure 5(d), some consecutive segments interleaved with random segments. The offsets of consecutive segments correspond to segmented-contiguous, while most offsets of random segments correspond to segmented-random. It can be observed that after sorting, the offsets show their respective characteristics when they are executed independently, which means that in the mixed load, the random factor exhibits a superimposed characteristic. The random percentage for the mixed loads is 71.88% (the total random factor is 91).

2.3 Data Redirection Based on Random Factor

The data redirector component in SSDUP+ is designed to transmit the data to HDD or SSD. This component mainly consists of two parts, which calculate the random percentage in the incoming request stream and dynamically determine a threshold for the random percentage through the proposed adaptive algorithm. When the random percentage of the incoming request stream is higher than the threshold, the requests will be redirected to SSD.

2.3.1 Calculating the Random Percentage. When the execution of an application starts, the data are written to HDD. In the meantime, the offsets of the requests in every request stream are traced and sorted. The random percentage, *percentage*, is calculated by $percentage = \frac{S}{N-1}$, where S is the total random factor calculated by Equation (1) and N is the number of requests in the request stream. The default length of a request stream is 128, which is the same as the queue size in the CFQ scheduler. The length of a request stream can be re-configured when the CFQ queue size changes.

Typically, the greater random percentage of the request stream, the lower the I/O bandwidth. To demonstrate this relationship, we use IOR to conduct a set of tests, in which the experimental environment is the same as that presented in Section 2.2. In this experiment, we calculate the random percentage of each workload whose access pattern is strided. The reason why we choose the strided pattern and not others is because the changes in random percentage in segmented-contiguous and segmented-random are not prominent because of the features of their data accessing. The experimental results are presented in Figure 6. It can be observed that in the cases of 8, 16, 32, 64, and 128 processes, the random percentages are 7%, 15%, 28%, 46%, and 71%, respectively, while the I/O throughputs are 208.1 MB/s, 211.76 MB/s, 175.8 MB/s, 159.29 MB/s, and 132.68 MB/s, respectively. It is apparent that the I/O throughput decreases as the randomness of the workload increases.

2.3.2 Determining the Threshold Adaptively. To determine the random requests, we introduce a new parameter, the threshold of the random percentage, denoted by *threshold*. If *percentage* of the current request stream (the concept of a request stream is defined in Section 2.1) is higher

Fig. 6. The changes in I/O throughput and random percentage of the request streams as the number of processes increases in OrangeFS.

than *threshold*, then the requests are deemed random and the incoming requests are redirected to SSD. If *percentage* is lower than *threshold*, then the next request stream (i.e., incoming requests) is regarded as sequential, and the subsequent requests are redirected to HDD. In the previous version of this work, SSDUP, we set the “high-water mark” and “low-water mark” thresholds (more specifically, 45% and 30% in our prototype, respectively). When the *percentage* is greater than the “high-water mark” threshold, the request stream is considered random, and the upcoming requests are redirected into SSD. If the *percentage* is less than a “low-water mark” threshold, then the request stream is treated as sequential, and the subsequent requests are redirected to HDD.

However, as the number of jobs that issue the I/O requests concurrently change over time, the I/O workload at the server side is dynamic. Simply setting a static, empirical threshold is not always accurate. Therefore, we propose a traffic-aware adaptive algorithm, which is able to dynamically adjust the threshold according to the current workload level.

The adaptive algorithm is designed to dynamically adjust *threshold* as new request streams arrive. When a new request stream arrives, the *percentage* of the stream is inserted into a list, called *PercentList*, in increasing order. A *percentage* is then selected from the *PercentList* and used as the *threshold*. When the access pattern of the workload changes, the *PercentList* will be emptied so that the access pattern of previous jobs does not interfere with the redirection of the request streams issued by new jobs. The following presents how to detect the access pattern in the implementation of SSDUP+. When SSDUP+ processes each request stream, it checks whether the difference between the randomness of the last k (e.g., 10) request streams and the threshold are greater than a certain value (e.g., 0.3). If the randomness of more than $x\%$ (e.g., 70%) of the k request streams (i.e., seven request streams) is greater than the threshold by 0.3, then the I/O access pattern is regarded as being changed and the current *PercentList* is emptied.

Equation (2) is used to calculate the element in *PercentList* that is selected as *threshold*, where *avgper* is the average over all elements in *PercentList*, as calculated in Equation (3). *avgper* is used as the basis for selecting the element in *PercentList*. N is the number of elements in the *PercentList*.

The rationale behind the selection of the elements from *PercentList* is explained as follows. When the values of recent random percentages are small, it indicates the randomness of the current workload is small and, therefore, the element with a bigger index in *PercentList* should be selected. Consequently, a smaller proportion of the incoming requests will be redirected to SSD. Otherwise, an element with a small index in *PercentList* should be selected, so that the random percentage of

Fig. 7. The random percentage of the request streams that are directed to SSD and HDD.

more request streams will be higher than *threshold* and therefore more requests will be redirected to SSD.

$$threshold = PercentList[(1 - avgper) * (N - 1)], \quad (2)$$

$$avgper = \frac{\sum_{i=0}^{N-1} PercentList[i]}{N}. \quad (3)$$

The adaptive algorithm that calculates the threshold is outlined in Algorithm 1.

Note that *percentage* is calculated based on the latest 128 requests that SSDUP+ has received and that the comparison between *percentage* and *threshold* is used to guide the direction of the upcoming (future) requests. This method is effective, because many HPC applications manifest the stable access pattern or smooth change in access pattern [41, 45]. It is very rare that the access pattern of a HPC application changes abruptly. Moreover, the data redirector module and the redirection algorithm work by tracking and using the properties of the data (i.e., the offsets and the sizes of the requests), not the data itself. SSDUP+ does not change the data accessing behaviour in any way.

We present the following case study to illustrate the adaptive determination of *threshold*. When we run the IOR instance, we record *Percentage* of the latest 10 request streams in *PercentList*. The set of percentages in the list is 0.3937, 0.5433, 0.5905, 0.6299, 0.6062, 0.5826, 0.622, 0.622, 0.622, and 0.6771. The thresholds that we calculated are 0.5, 0.5433, 0.5433, 0.5433, 0.5905, 0.5826, 0.5826, 0.5905, 0.5905, and 0.6062 as the request streams arrive. It is clear that the thresholds vary from the random percentages of the request streams. The request streams that are directed to SSD are those with the percentages of 0.6299, 0.6062, 0.5826, 0.622, 0.622, and 0.6771. The decision of directing the data to SSD is deemed a correct decision when the percentage of the request stream is greater than the average threshold. In this case study, the IOR instance contains 512 request streams. The proportion of successful directions is 79.48%.

To better illustrate the effect of this redirection, we conduct two sets of experiments. Figure 7(a) shows that most requests are of low randomness, while in Figure 7(b), most requests have high random percentages. We plot the direction of each request stream as a scatter plot, where a red or blue dot indicates that the request stream is written to the SSD or the HDD. It can be seen from Figure 7(a) that almost all requests under the horizontal line (the threshold) are directed to HDD. In the conventional burst buffer scheme, these requests will be cached in SSD and, consequently, consume more SSD space. It can also be seen that above the horizontal line, some requests are cached in SSD and some are still written to HDD directly. The reason some requests whose

ALGORITHM 1: Redirection Algorithm**Input:** *requests*Send *requests* to HDDs**repeat** Group *requests* into requests stream Sort the offsets of *requests* in requests stream Calculate S (sum of RF) Calculate *percentage* of RF Insert *percentage* into *PercentageList* **if** *percentage* > *threshold* **and** *current requests being sent to HDDs* **then** | Send *reqs* of next requests stream to SSDs **end** **if** *percentage* < *threshold* **and** *current requests being sent to SSDs* **then** | Send *requests* of next requests stream to HDDs **else** | Send *requests* of next requests stream to the current device **end****until** *No incoming reqs*;

random percentage is greater than the threshold are still written to HDD is because we use the random percentage of current request stream to predict the random percentage of the next request stream. In some cases, the prediction is always accurate, especially when the randomness of the current request stream is around the threshold line. As can be seen from Figure 7(b), most requests whose randomness is higher than the threshold line are cached in SSD. These results show that SSDUP+ can set the threshold of the randomness adaptively and direct the requests to SSD or HDD accordingly.

In SSDUP+, there is no need to set the thresholds manually thanks to our traffic-aware adaptive algorithm. Comparing with SSDUP, whose thresholds take the empirical values, the SSD utilization can be further improved (i.e., less SSD capacity is required for achieving the same level of I/O performance).

2.4 The Pipeline Scheme

2.4.1 Performance Improvement with the Pipeline Scheme. In SSDUP+, we divide the total area of SSD into two equal-sized regions. Both regions are empty initially. When an I/O request is forwarded to SSD, SSDUP+ first selects an empty region, called *Region1* ($R1$), to buffer the incoming data. When $R1$ is filled up and a new request is forwarded to SSD, SSDUP+ then picks the other free region, called *Region2* ($R2$), to buffer the data. At the same time, $R1$ starts to flush its data to HDD. Namely, SSDUP+ flushes the data in $R1$ and writes the data to $R2$ simultaneously. When $R2$ is filled up, it is likely that the space of $R1$ has already been released (if $R1$ has not been released yet and a new request arrives, the I/O request will block until $R1$ is released). With the pipeline scheme, data flushing overlaps data writing. Consequently, the I/O throughput can be improved. From another perspective, it is more likely that there is the free SSD space when the requests arrive. Therefore, we can achieve the same I/O throughput with a smaller SSD capacity. We will conduct the detailed analysis about the effectiveness of the pipeline scheme in next subsection.

2.4.2 Further Optimization with the Traffic-aware Pipeline Strategy. As the data are being flushed from SSD to HDD in the pipeline scheme, the data from other jobs may be written to HDD at the same time. The HDD writing from these two sources (i.e., data flushing from SSD flushing and data writing from jobs) may interfere each other, causing more disk seeking movements

and, consequently, degrading the I/O throughput. To further optimize the I/O throughput, we propose a traffic-aware flushing strategy in the pipeline scheme, which judiciously flushes the data to reduce the I/O interference discussed above.

The traffic-aware flushing strategy is designed based on the following reasoning. We can determine the randomness of I/O traffic by random percentage. When the data need to be flushed, the traffic-aware flushing strategy checks the random percentage of current I/O traffic. If its random percentage is high, then it suggests most requests are directed to SSD and, consequently, the traffic directed to HDD is low. Therefore, the data are flushed as normal. On the contrary, if the random percentage is low, then it indicates that the traffic directed to HDD is high. Therefore, data flushing is paused to avoid the I/O interference. Data flushing stays paused until the random percentage becomes high again. Another advantage of this strategy is that we can make the data flushing phase to overlap the computing phase more so as to hide the flushing overhead.

2.4.3 Effectiveness Analysis of the Pipeline Scheme. To better analyze the advantages of the pipeline strategy, we simulate and analyze the costs in different scenarios through mathematical modeling. Assume the I/O phase is divided into n stages, and the size of data transmission in each stage is the same. T_{HDD} and T_{SSD} denote the time of writing one stage of data to HDD and SSD, respectively. Also assume that the entire capacity of the SSD is sufficient to accommodate m stages of data ($m < n$). The total I/O time without the pipeline mechanism, denoted by T_1 , can be calculated as Equation (4),

$$T_1 = m * T_{SSD} + (n - m) * T_{HDD}. \quad (4)$$

In the pipeline mechanism, the SSD is divided into two equal-sized regions. Each region can handle $m/2$ I/O stages. So except for the first and final $m/2$ stages, all other stages (of n stages) are handled in pipeline (i.e., while one region is flushing a stage of data, the other region is buffering the next stage of data). Therefore, there are in total $n - 2 * m/2$ stages that are handled in pipeline.

Assume the first $m/2$ stages of data are written to the SSD region $R1$. While the next $m/2$ stage of data are written to the region $R2$, the data in $R1$ are flushed to HDD (i.e., these stages of data are handled in pipeline). Except for the first and last $m/2$ stages, all other stages are fully handled in pipeline. Thus, the total number of I/O stages handled by the pipeline is $(n - 2 * m/2)$. When the stages are handled in pipeline, the time spent by one stage is determined by the maximum between the time of flushing one stage of data (denoted by T_f) and the time of buffering one stage data (denoted by T_b). Therefore, the I/O time spent in writing n stages of data with the pipeline mechanism, denoted by T_2 , can be calculated by Equation (5),

$$\begin{aligned} T_2 &= \frac{m}{2} * T_{SSD} + \left(n - 2 * \frac{m}{2}\right) * \max \{T_f, T_b\} + \frac{m}{2} * T_{SSD} \\ &= m * T_{SSD} + (n - m) * \max \{T_f, T_b\}. \end{aligned} \quad (5)$$

In Equation (5), T_b is actually T_{SSD} in Equation (4). As writing the data to SSD is faster than to HDD typically, T_f is larger than T_b . For this reason, in some occasions, both regions may be filled up (when the amount of random writes is larger than the SSD capacity). In this case, the system waits until a region becomes empty (the data in that region have been flushed). So T_2 becomes

$$T_2 \approx m * T_{SSD} + (n - m) * T_f. \quad (6)$$

In the flushing stage, the data are written back to HDD in a well-ordered fashion. Therefore, T_f is the time of writing a stage of data to HDD sequentially. T_{HDD} is the time of writing the data without ordering to HDD, which is typically larger than T_f . Therefore, T_1 is larger than T_2 , which shows the pipeline mechanism reduces the I/O time of writing the data.

Fig. 8. Metadata management with the AVL tree.

Equation (6) effectively models the I/O time spent in writing the data generated by a single application. In Equation (5), T_f in the second term is the time for flushing one stage of data in SSD. However, as discussed in Section 2.4.2, as the number of I/O applications increases, there may be I/O interference when the data are flushed to HDD. Therefore, the flush time T_f will increase. T'_f denotes the time of flushing one stage of data under I/O interference. We have $T'_f > T_f$. So the total I/O time under the I/O interference (denoted by T'_2) is calculated by Equation (7), which is larger than T_2 . This shows that the I/O interference reduces the I/O performance, which is the reason why a traffic-aware pipeline strategy is designed in SSDUP+ to avoid the performance degradation caused by the I/O interference,

$$T'_2 \approx m * T_{SSD} + (n - m) * T'_f. \quad (7)$$

2.5 Buffer Management Using the AVL Tree

To reduce the performance loss caused by the write amplification [36] of SSD, SSDUP+ is designed to write the data to SSD in a log-structured way, i.e., append the data to the end of the cached files. However, in the log-structured mode, the logical order of the data in the file is disrupted. As shown in Figure 8, the logical order of the data blocks in a file is $\{\#1, \#2, \#3, \dots, \#10\}$. But the data blocks are requested in sequence $\{\#1, \#7, \#8, \dots, \#9\}$, which is then the order in which the data is written to SSD.

To recover logical data order, we need a mechanism to manage the metadata of the cached data, i.e., to record the relation between the accessing order of a data block and its logical order in a file. Normally, a hash table is a desired choice because of its $O(1)$ time complexity 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 sorting algorithm such as quicksort takes the time of $O(n \log n)$. In this work, 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, which takes the time of $O(\log n)$ for the basic operations. When SSDUP+ writes the data to the SSD, it records the original metadata (including the original offset and size) and new metadata (including new offset and size). Both original and new metadata of the same data are stored in a leaf node. Each value requires 8 bytes, which adds up to 24 bytes for one node. The storage overhead of the AVL tree is low. Only about 3 MB of storage space is needed to store a AVL tree for a file of 40 GB (the request size is 256 KB). The nodes are sorted based on the original data offset. As shown in Figure 8, in node $(\#2, *4)$, $\#2$ represents the original offset while $*4$ represents the new offset in SSD. Each AVL tree stores the metadata of one file. This way, the data sequence can be maintained while buffering, which saves an unnecessary sorting phase.

When the data are flushed from SSD to HDD, the data blocks are read from SSD and written to HDD in their logical order by looking up the AVL tree. A significant advantage of using the AVL tree is that when the data are flushed to HDD, SSDUP+ only needs to conduct an ordered traversal of the AVL tree. Note that when traversing the AVL tree, we can ensure the data are in

their original sequence. However, as the way in which the data are written changes the original data layout, it is the random read to retrieve the data from the SSD. Since the SSD has the nearly zero seek delay, the random read from SSD does not hurt the performance.

Additionally, writing to the SSD sequentially can avoid the write amplification when the SSD is heavily occupied. At present, only a very small number of SSDs do not suffer from the writing amplification problem. Intel Optane SSD based on 3D XPoint is one of them. However, Optane SSD is much more expensive than NAND-based SSDs and is still too early to be deployed in the I/O systems at a large scale. For the current mainstream HPC systems (e.g., Sunway TaihuLight), the NAND flash-based SSD is most widely used. Therefore, our work in avoiding the write amplification is still very useful.

We now discuss what happens when the data are re-written in our data management scheme. When the data need to be accessed, an I/O request will be issued. There are two situations to be consider: (1) the file that contains the data to be accessed has been flushed to HDD and (2) the file that contains the data to be accessed has not been flushed yet (including the case (i) where the flush of the file that contains the data to be accessed has not started yet and (ii) the case where the file that contains the data to be accessed is being flushed). In the first situation, if the rewrite request is regarded as a low random request, then the data will be rewritten directly into HDD. If the data rewrite request is regarded as a highly random request, then the data rewrite request will be cached in SSD. The data in HDD will be rewritten when the data rewrite request is flushed to HDD. In the second situation, when the file (part of the file or the entire file) is in SSD, the file is in the state of “open.” Therefore, the data access operation will block until the file has been flushed to HDD. Once the file has been flushed to HDD, it becomes the first situation, as discussed above.

3 EVALUATION

We have conducted extensive experiments to validate the design of SSDUP+ and to evaluate its performance. We present the results and analyses in this section.

3.1 Experimental Setup

The experiments were conducted on a cluster of 10 nodes, in which 8 nodes are compute nodes and 2 nodes are I/O nodes. Each compute node is equipped with 16 Intel Xeon E5-2670 CPU processors, 64-GB RAM, and a 300-GB SATA hard drive. Each I/O node is equipped with 16 Intel Xeon E5-2670 CPU processors, 8-GB RAM, a 300-GB SATA hard drive (Toshiba model MBF2300RC), and a 240-GB SSD (INTEL® SSD DC S3520 SERIES). 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 while the default I/O scheduler for SSD is NOOP [2]. All nodes are connected via a Gigabit Ethernet. MPICH-3.0.2 release [26], compiled with ROMIO, is installed on the compute nodes.

Although there are many hybrid data storage schemes between SSD and HDD in literature, many of them are the computing node-oriented schemes as we discussed in Section 2.1 while our SSDUP+ is a server-oriented scheme. Comparing with computing node-oriented scheme does not help validate the benefits of the design considerations in SSDUP+. Moreover, most of the solutions are not open source, and it is difficult to find a similar system for comparison. Thus, we simulated the systems with different writing strategies based on the original OrangeFS. We focus on comparing SSDUP+ with two contemporary file systems: the original OrangeFS-2.9.3 and OrangeFS equipped with the conventional server-oriented bursty buffer scheme (i.e., all traffics are buffered in SSD (without differentiating random or sequential accesses), and there is no the two-stage pipeline mechanism as in SSDUP+). In addition, we also compare SSDUP+ with the previous version of this work, SSDUP [34]. By comparing with such a conventional scheme and SSDUP, we

can demonstrate the benefit of detecting randomness of the data and the pipeline mechanism in SSDUP+.

OrangeFS adopts a client/server model. It has been widely used as both an experimental platform and a production platform in HPC areas. Files are striped across multiple I/O nodes to enable parallel I/O with high aggregated bandwidth [31]. In our experiments, we deployed OrangeFS on all I/O nodes to manage the server-side SSDs as a generic remote-share Burst Buffer that we called OrangeFS-BB. In addition, to assess the potential of SSDUP+ for HPC applications, we use three standard HPC I/O benchmarks, IOR, MPI-TILE-IO, HPIO, to cover most I/O patterns.

3.2 IOR Benchmarking and Analysis

We used IOR-2.10.3, a parallel file system benchmark developed at Lawrence Livermore National Laboratory, as one of the evaluation benchmarks. 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 conducted three sets of test using three different access patterns to simulate different execution stages. The first, second, and third test set use the access patterns of *segmented-contiguous*, *strided*, and *segmented-random*, respectively.

The size of each I/O request is 256 KB, and the first two sets of test write the data to a shared 16-GB file. The final set of tests write the data to a shared 8-GB file, because a completely random pattern is relatively rare.

3.2.1 Performance with Different Numbers of Processes. In the first set of evaluation, we ran the IOR instances (segmented-contiguous, strided, and segmented-random patterns) with 8, 16, 32, 64, 128, 256, and 512 processes, aiming to investigate the performance and the SSD usage of SSDUP+, the original OrangeFS, OrangeFS-BB, and SSDUP. In this set of experiments, the capacity of SSD is set to be large enough to hold all the data.

Figure 9 shows the impact of the number of processes on the performance and the SSD usage of SSDUP+. As the number of processes increases from 8 to 32, the performance of the original OrangeFS increases slightly because of the positive effect of parallel I/O. However, as the number of processes increase further (from 64 to 512), the performance of the original OrangeFS starts to decline, which is due to the increased I/O competition caused by too many processes. The performance of OrangeFS-BB shows a different trend. Its performance improves greatly from 8 to 16 processes and then improves slightly from 32 to 512 processes and eventually maintains at the high performance. This is because OrangeFS-BB uses SSD to cache all data.

The performance of SSDUP+ improves gradually as the number of processes increases from 8 to 32. It reaches the same performance as the original OrangeFS with 32 processes. Moreover, SSDUP+ only buffers 25% of the data in SSD, which is 75% less than OrangeFS-BB. This is because SSDUP+ uses the *random factor* to filter the workloads and only buffer random requests in SSD. When the number of processes is small, only the requests with the segmented-random pattern are redirected to SSDs, which account for 25% of the total data size.

From 64 to 512 processes, the performance of SSDUP+ always maintains at almost the same level as OrangeFS-BB (only 2.15%, 4.99%, and 2.53% lower than OrangeFS-BB, respectively). In contrast, SSDUP+ only buffers 40%, 66%, 84.5%, and 97% of the data, which is less than OrangeFS-BB by 55%, 34%, 15.5%, and 3%, respectively. This is because when the number of processes increases, more requests with the segmented-contiguous and the strided pattern are identified as random requests due to the I/O interference between the different processes [49] and therefore are redirected to SSD.

In addition, although SSDUP is almost comparable to SSDUP+ in performance when the number of processes increases to and beyond 64 processes, SSDUP uses much more SSD space than SSDUP+

Fig. 9. I/O throughput with different numbers of processes for the IOR benchmark, with the original OrangeFS, OrangeFS-BB, SSDUP, and SSDUP+.

Fig. 10. I/O throughput with different sizes of CFQ queue for the IOR benchmark for the original OrangeFS and SSDUP+.

to cache the data. As shown in Figure 9, SSDUP+ uses 41.5%, 33%, 15.5%, and 3% less SSD space than SSDUP. This is because SSDUP uses a static empirical threshold to determine whether the requests are random, but SSDUP+ uses an adaptive algorithm to identify the random requests.

3.2.2 Performance with Different CFQ Queue Sizes. We also measured the impact of the CFQ queue size on the SSDUP+ performance. We conducted three groups of experiments with the size of the CFQ queue being 32, 128, and 512. The IOR instances was run with 32 processes. SSDUP+ achieved 59.7%, 41.5%, and 12.3% performance improvement, respectively, as shown in Figure 10. Note that the performance depicted in the figure is the average I/O throughput of a single I/O node. The SSD device we used for the experiments is INTEL SSDDCS3520SERIES. This SSD is designed for data centers. Its theoretical peak performance is 250 MB/s according to its specification. Also note that the default size of the CFQ queue is 128. When the size was changed to 32, CFQ became more sensitive to the concurrent accesses with interferences, which resulted in the decrease of the system I/O bandwidth to 124 MB/s. We also adjusted the length of the request stream to increase the percentage of random factor, which caused more data to be identified as random accesses and be redirected to SSD. The ratio of data directed to SSD became 92%, which is greater than the ratio in the case where the queue size is 128. When the CFQ queue size became 512, the average throughput of the original system increased to 179 MB/s. With a larger queue size, the CFQ scheduler has more opportunities to merge adjacent requests and achieves better locality. Just because of this reason, the throughput is less sensitive to random accesses [47]. In these cases, SSDUP+ only achieved 12.3% improvement and a small portion of segmented-random data were redirected to SSD.

3.2.3 Performance with Limited SSD Capacity. To verify the performance improvement of different pipeline strategies in multi-load scenarios, we tested the performance of different loads under different flushing strategies with the SSD of a small capacity. The reason why we chose a small SSD is because in the HPC area, the amount of data generated in a computing phase is very likely to exceed the Burst Buffer size. Therefore, we would like to test the performance of SSDUP+ in the scenarios where the SSD capacity is constrained.

First, we set the SSD capacity as 8 GB. Note that in OrangeFS-BB, the 8 GB is used as an entire space, while in SSDUP+, SSD is divided into two 4-GB regions. Then we run the following workloads in the tests: *workload*₁ includes two IOR instances, and its access pattern is segmented-contiguous and random; *workload*₂ also contains two IOR instances, but its access pattern is segmented-random. Each IOR instance writes 8 GB of data, and the request size is 256 KB. As

Fig. 11. Performance comparison between OrangeFS-BB, SSDUP, and SSDUP+ with limited SSDs capacity.

Fig. 12. Performance comparison between SSDUP+ and OrangeFS-BB with limited SSDs capacity and different computing time.

shown in Figure 11, in which a bar in the figure corresponds to the performance of an IOR instance, OrangeFS-BB achieves the I/O bandwidth of 73.04 MB/s and 72.71 MB/s for these two IOR instances in *workload₁*. However, SSDUP+ achieves the bandwidth of 90.21 MB/s and 90.49 MB/s for two IOR instances, which accounts for 23.98% of improvement. This is because SSDUP+ uses the two-stage pipeline buffering strategy. When one region is full, the random requests are served by the other region. In this test, SSDUP+ performed three flush operations. The total delay of the first two flush operations was 17 and 19 s, respectively. However, due to the SSD being full, OrangeFS-BB can only write the data to HDD while SSD is flushing the data to HDD, which leads to intense I/O contention in HDD between the two streams of data writing. This is one of the reasons why the performance of SSDUP+ is better than OrangeFS-BB. The percentages of *RF* of *workload₁* is about 70%, which suggests that a portion of requests in the workloads are still handled by HDD. It is not always good to perform the flush operation immediately when a region of SSD is full. The flush operation should be performed when the load in HDD is low, so that the I/O contention and, consequently, the performance loss can be reduced. Since SSDUP+ can accurately determine the write path of each request stream, it can accurately identify the current load in HDD. Besides, the I/O bandwidth achieved by SSDUP+ for two IOR instances of *workload₂* is 97.32 MB/s and 98.38 MB/s, which is 8.3% higher than that of *workload₁*. This is because the requests of *workload₂* are all random, and there are almost no requests written directly to HDD. When a region of SSD is full, the strategy of flushing immediately to HDD is used, and no delay is required. The performance of *workload₂* are 71.16 MB/s and 71.54 MB/s, respectively, which are almost the same as *workload₁*.

In addition, we also tested SSDUP, which does not have the traffic-aware flushing scheme. As shown in Figure 11, SSDUP achieved the I/O bandwidth of 67.85 MB/s and 66.15 MB/s for the two IOR instances of *workload₁*, which is 34.8% lower than the bandwidth achieved by SSDUP+. This is because that SSDUP starts flushing immediately once a region of SSD is full without taking into account the current load in HDD, which may lead to intense I/O contention. OrangeFS-BB achieves slightly better performance for *workload₁* than SSDUP, because OrangeFS-BB uses SSD to handle all data while much less data are handled by SSD in SSDUP (hence a SSD of a much small capacity is needed by SSDUP).

However, the performance of *workload₂* is the same in SSDUP+ as in SSDUP, because the access pattern of *workload₂* is very random and, consequently, almost all data are written to SSD. The workload written to HDD directly is almost zero. Therefore, the flush operation will not experience interference, which is the similar effect as when *workload₂* is processed by SSDUP+.

3.2.4 Performance with Different Computing Times. We have also investigated the impact of different computing times on the system performance. When the SSD capacity is less than the amount of data being written, which is very common in practical applications, the I/O operations of the applications and the flushing operating of SSD will collide, resulting in performance degradation. Traditional Burst Buffer attempts to solve this problem by overlapping the computing stage and the flushing stage. However, the computing time is difficult to predict. Therefore, we would like to evaluate the impact of different computing times between two consecutive I/O phases on the overall performance of OrangeFS-BB and SSDUP+. We ran two identical IOR instances sequentially and adjusted the interval times between these two instances from 0 s to 30 s. Each IOR instance was executed with the segmented-random access pattern and produced a 8-GB shared file. Moreover, the capacity of one region of SSD was set to 2 GB in an I/O node managed by SSDUP+, and the capacity of SSD was set to 4 GB in an I/O node managed by OrangeFS-BB. The total size of SSD accounts for 50% of the size of data to be written.

As shown in Figure 12, the I/O throughput of OrangeFS-BB is improved gradually when the computing time increases. This is because after the first IOR instance fills the SSD buffer, OrangeFS-BB starts the flushing phase, during which the SSD buffer cannot process the new requests. When the flush phase is over, the SSD buffer can continue to process the second IOR instance. So when the interval between the two IOR instances is very short, the second IOR instance conflicts with the flush phase, causing a sharp drop in performance. SSDUP+ outperforms OrangeFS-BB by 11.91%, 10.65% and 9.92% with different computing times. In addition, when the computing time is 0 s, the performance of SSDUP+ is only 20% lower than the peak performance, while that of OrangeFS-BB is 34% lower. Notably, the performance of SSDUP+ with a computing time of 10 s reaches the peak performance of OrangeFS-BB when the computing time is 30 s. This indicates that SSDUP+ can tolerate the increase in computing time much better than the traditional Burst Buffer.

3.3 HPIO Benchmarking and Its Analysis

We also used the HPIO benchmark [9] to evaluate and compare SSDUP+, the original OrangeFS, OrangeFS-BB, and SSDUP in terms of both I/O throughput and the SSD usage. HPIO is a benchmark developed at the Northwestern University and has been widely used to evaluate the performance of non-contiguous I/O accesses. Similarly to IOR, HPIO can be run in different access patterns too. Four parameters were used in this test: region size, region count, region spacing, and non-contiguous test array. A region is a piece of contiguous data in a file that will be accessed by a process. The region count is the number of regions accessed by a process. Region spacing is the distance between two adjacent regions. Non-contiguous test array indicates whether the file access is continuous or random.

We set the region size from 32 KB to 256 KB and the number of processes is set to 32. The region count varied from region size to keep the file size to be around 8 GB. In addition, we set the region space as 0. To fully evaluate the potential of SSDUP+, we run two HPIO instances concurrently to simulate complex workloads, one of which is of the continuous pattern with the non-contiguous test array set to 1000 (c-c). The other is of the random pattern, and the non-contiguous test array was set to 0010 (c-nc).

As shown in Figure 13, the performance of OrangeFS-BB and SSDUP is roughly the same, because both buffered 100% of the data in SSD. The performance of SSDUP+ is slightly lower than that of SSDUP but not by more than 6% (actually 2.84%, 5%, 5.2%, and 1.4%, respectively). This is because SSDUP+ cached less data in SSDs than SSDUP, and the saved SSDs space is 17.39%, 16.07%, 13.6%, and 19.86%. This means that in this scenario, SSDUP+ sacrifices less than 6% of the performance compared with OrangeFS-BB and SSDUP but saves an average of more than 15% of the SSD space.

Fig. 13. I/O throughput with different region sizes for the HPIO benchmark, with the original OrangeFS, OrangeFS-BB, SSDUP, and SSDUP+.

Fig. 14. Comparison of the original OrangeFS, OrangeFS-BB, SSDUP, and SSDUP+ in terms of I/O throughput for the MPI-Tile-IO Benchmark with different number of processes.

3.4 MPI-Tile-IO Benchmarking and Analysis

MPI-Tile-IO benchmark [30] is a member of the Parallel I/O Benchmarking Consortium, which is a test application that implements the tile access to a two-dimensional dense dataset. Each process accesses a tile, the size of which is based on the number of elements in a tile and the size of one element. In this test, we run two MPI-Tile-IO instances concurrently with 16, 32, 64, and 128 processes. The first instance was set as a one-dimensional dense dataset whose x direction was set to 1 and the y direction set to the number of processes. In the second instance, as a normal two-dimensional dataset, the x direction is set as the square root of the number of processes, while the product of the y direction and x direction is kept to the number of processes. In addition, the size of each element is set to 4 KB and each instance generated 16 GB of data.

As shown in Figure 14, the I/O throughput achieved by OrangeFS drops as the number of processes increases. This is because the I/O contention between different MPI-TILE-IO instances increases as the number of processes increases, resulting in a much greater randomness for mixed loads. OrangeFS-BB maintains the peak performance by writing all data directly to SSD. SSDUP+ and SSDUP have the same performance as OrangeFS when there are 16 processes. This is because there is no request identified as the random request, and therefore the SSD space used is 0. When the number of processes is 32, the performance of SSDUP+ and SSDUP is almost the same as OrangeFS-BB, but only 46.87% of the data is written to SSD in SSDUP+, while 95% of the data is written to SSD in SSDUP. Nearly 50% of the SSD space is saved in SSDUP+. As the number of processes increases further, SSDUP identifies all data as random requests and writes them to SSD, just as in OrangeFS-BB. However, SSDUP+ saves 27.5% and 15% of SSD space, respectively, comparing with SSDUP and OrangeFS-BB and always maintains as high performance as OrangeFS-BB.

3.5 Overhead Analysis

The overhead of SSDUP+ primarily comes from two aspects: the cost of grouping and sorting the requests in a request stream (called *grouping cost*) and the cost of keeping the AVL tree balanced and in order and travelling the AVL tree (called *AVL cost*). We use the IOR benchmark to study and analyze these overheads. The total size of the accessed data is 2 GB, and the request size varies from 32 KB to 512 KB. The SSD capacity was set to be 2 GB. The IOR benchmark was executed with the segmented-random pattern, and all requests were sent to the SSD.

Table 1. System Overhead

Request Size	Total Time	Group Cost	AVL Cost
32 KB	15.5 s	29.1 ms	93.4 ms
64 KB	12.5 s	12.5 ms	44.2 ms
128 KB	11.8 s	15.9 ms	41.8 ms
256 KB	11.54 s	9 ms	21.2 ms
512 KB	11.9 s	6.1 ms	9.5 ms

The system overheads was measured to be 0.13% (in the case of 512 KB requests) and 0.79% (in the case of 32 KB requests) of the total execution time, which can be ignored compared to the performance gain. As shown in Table 1, these two types of overhead increase as the request size becomes smaller. This is because when the request size becomes smaller, the number of I/O requests increases. The overheads in the cases of 128-KB and 64-KB requests are close, because the request is striped across two data servers when the request is larger than the default stripe size. In addition, the depth of the tree grows at a logarithmic scale with the adopted data structure of the AVL tree. The time overhead of the AVL tree remains small as the amount of data grows. When the request size is 256 KB and the dataset size is 416 GB, the overhead caused by the AVL tree only accounts for 0.268% of the total I/O time, which is not obviously larger than that with the dataset of smaller size. This is because the time overhead of the AVL tree increases logarithmically as the amount of data written to SSD increases. Also, the size of the AVL tree is limited by the size of a single SSD region, because when a SSD region is full, the corresponding AVL tree stops growing. The storage space needed to store the AVL tree increases linearly with the data to be stored. Since we only need to store the metadata of the requests (e.g., offset address and request size). It only consumes 56 Bytes for one request. When the request size is 256 KB, the AVL tree only consumes 216 MB for 1-TB data, which causes little storage stress given that the current memory space is typically tens or hundreds of gigabytes.

These experimental results show that SSDUP+ improves the write performance when using 40% of the total SSD space (the ratio of random accesses is 20%) for the IOR benchmark, 50% of the total SSD space (the ratio of random accesses is around 20% to 100%) for the HPIO benchmark, and 90% of total SSD space (the ratio of random accesses is in the range of 80% to 95%) for the MPI-Tile-IO benchmark. The overhead is negligible, less than 1% of the total execution time. The average ratio of random accesses in HPC applications has been reported to be around 50% [50], which is consistent with the ratio setting in our experiments. Compared with burst buffer, SSDUP+ can save up to 50% SSD space. In addition, our approach does not introduce more writes to SSD. Instead, SSDUP+ only buffers random I/O requests and reduces the number of writes to SSD. As the result, SSDUP+ can extend the lifetime of SSD compared with the design of conventional burst buffer. Since IOR, HPIO, and MPI-Tile-IO are typical benchmarks that represent the common access patterns (segmented-contiguous, segmented-random, strided, noncontiguous and nested-stride) of scientific applications, we believe that SSDUP+ can achieve similar performance gains for realistic workloads.

3.6 Analysis of SSD Garbage Collection

To evaluate the impact of garbage collection on I/O performance when the SSD buffer is almost full, five different sizes of datasets are used in the experiments, which are 16, 104, 208, 312, and 416 GB. Since the SSD capacity configured on an I/O node is 220 GB, it can reflect the scenario where the SSD buffer is full when the dataset size is 416 GB. Note that all the data accesses in the experiments are random, and therefore all data are written to SSD. As shown in Figure 15, there is no noticeable fluctuation in I/O performance as the size of the dataset changes. This shows

Fig. 15. The I/O throughput of IOR with different data sizes.

that even if the SSD buffer is exhausted, the SSD GC will not cause noticeable degradation of I/O performance in the way that SSDUP+ uses the SSD.

4 RELATED WORK

Many research studies have been conducted to improve the performance of the I/O system for HPC applications [37, 45, 48]. In this section, we discuss the existing work in four areas: (i) addressing the random access problem caused by concurrent access to the hard disk, (ii) identifying the critical data and access patterns, (iii) the storage systems with SSD, and (iv) the extension from the previous version of this work, SSDUP.

4.1 The Problem of Concurrent Access

Wang et al. introduced an IBCS method [42], 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 [10] in each iteration, which prevents multiple processes from competing for one disk head. Because only one process accesses the disk at a time, the system throughput is significantly improved due to the reduced cost of disk seeking. Chen et al. [8] analyzed the difference between the data layout at the client side and the data layout in parallel file systems. The difference causes the processes to interfere each other when they access the hard disk. Zhang et al. [46] reported that the file striping strategy of a parallel file system may jeopardize the locality of individual programs when multiple programs were served concurrently by a data server. Zhang et al. [49] studied a common pattern of HPC applications, that is, (a) the offsets of individual processes are continuous, but (b) multiple processes may compete for a disk head when the processes handle the requests concurrently, which makes the disk head move back and forth. They propose a data replication method that copies the data of the same process to the same I/O node and ensures that each I/O node serves as few processes as possible. All these approaches will operate on the data itself (merge or migrate, etc.), and the system we designed does not need to operate on the data but only analyzes the meta-data to obtain the data access pattern. And the historical access record is used to guide the writing of next requests. So our design is simple and efficient, and the system overhead is extremely low.

4.2 Recognizing Critical Data and Access Patterns

S4D-Cache [13] introduces a new technique to identify the data that are critical to performance and redirect these data to SSD. S4D-Cache migrates the data between SSD and HDD according to the temporal locality. The method is able to keep the data with strong temporal locality in the SSD longer. Byna's prefetching technique [5] prefetches the data into the memory by detecting a stable

local access pattern. The LBSC [6] redirects the I/O requests by differentiating the randomness of requests to achieve load balancing of SSDs and HDDs. By classifying the I/O workload of applications, ReCA [32] divides all I/O workload types into five categories and then configures the use of SSDs and HDDs according to different types. Both S4D-Cache and I/O prefetching analyze the access pattern from the perspective of a single process. However, multiple processes that access the same server can cause competition and random accesses. More processes are involved, more random the IO accesses can be. It is difficult to calculate the cost accurately from a single process's point of view. LBSC simply differentiates I/O randomness based on the type of application and the number of consecutive requests. ReCA does not fully consider the characteristics of SSD and HDD in the I/O pattern classification but only classifies the I/O behavior. Therefore, the results of the classification are insufficient for the use of SSD and HDD. Our proposed method traces the access pattern in a global view, which can measure the random access caused by both an application's native behavior and the competition between multiple processes. SSDUP+ fully considers the characteristics of SSD and HDD when analyzing I/O pattern and performs I/O redirection according to the advantages of different devices. Thus, it can be immediately perceived when the I/O pattern changes and is handled by a unified dynamic caching strategy.

4.3 Storage Systems with Burst Buffer

Burst buffer [22] inserts the I/O forwarding nodes equipped with SSD/DRAM in clients or between the clients and the storage nodes. By doing so, the data are flushed from the memory to the persistent storage quickly and the execution flow can return to the computation phase as soon as possible. A number of Burst Buffer systems focus on using the high performance of DRAM to accelerate bursty writing (e.g., checkpointing). For example, CRUISE [29] and BurstMem [39] are using distributed memory file system to speed up burst write and Wolfram Schenck et al. used the infinite memory engine to accelerate Burst Buffer [33], but their drawbacks are obvious. On the one hand, small-capacity memory cannot achieve the purpose of acceleration and the cost of large-capacity memory is unbearable; on the other hand, when the system crash or power failure, the data in memory will be directly lost. However, our SSDUP+ is optimized for systems with Burst Buffer whose mainly physical hardware is SSD. The price-to-capacity ratio of the SSD is much lower than that of the DRAM, and the data in SSD are not easily lost when system crash or power off.

In addition, some works related to the Burst Buffer use SSD as the main cache device. Hystor [7] identifies the data that cause long latencies and stores them in SSD. However, it needs to maintain an overall data map for entire system, which is very expensive. Moreover, Hystor only considers the I/O access pattern but does not take into account the characteristics of applications in high-performance computing. In addition, Hystor is designed only for stand-alone and embedded kernels, not for parallel environments. BurstFS [40] carefully analyzes the data characteristics of scientific applications to access the data efficiently. But its goal is to pursue the optimal performance, without considering the efficiency and cost-effectiveness of SSD.

Kun Tang et al. [35] adjusted the I/O scheduling strategy to make the flush time of Burst Buffer more reasonable, so that different applications use BB to perform peak shifts to improve the overall I/O bandwidth. However, the schemes of this kind will have to make a significant change to the I/O behavior of the job, which greatly increases the complexity of the I/O subsystem. Also, the adjustment of the I/O scheduling will fail when the application type of the system changes. Jaehyun Han et al. [12] designed a user-isolated scheme to improve the performance of Burst Buffer, so that the number of users using the same SSD is as small as possible and the problem of write amplification caused by random writing can be avoided. However, this design does not consider the practical application scenario of Burst Buffer in HPC. It is not uncommon for different users to

compete for the same SSD. Furthermore, the methods of Feng Ye et al. [44] and Sai Huang et al. [16] are designed to improve the lifetime of SSDs when they are simply used as the HDD buffer. These designs all reduce the amount of data written in the SSD to increase the lifetime of SSD. However, these designs do not consider the application scenario of HPC and only optimize the data-caching for SSD on a single machine.

SSDUP+ addresses the problems in the above designs. First, by tracing I/O from a global perspective, we can detect the random accesses caused by various situations, which makes the I/O accesses more efficient and increases the lifetime of SSD. Second, by designing a pipeline mechanism and the AVL tree, we can use the SSD space in a more efficient way, the result of which is to use less space to buffer more requests and not to rely on the accurate prediction of computation time to overlap the flushing stage. In addition, we do not interfere with the I/O behavior of application. Therefore, even if the I/O pattern of the applications changes, it will not affect effectiveness of our design.

5 CONCLUSION

In this article, we propose a traffic-aware SSD burst buffer scheme, called SSDUP+. We carefully analyze three common access patterns in the HPC environment. The proposed I/O-traffic detection method is able to identify random/irregular I/O accesses caused by various access patterns. SSDUP+ is designed to buffer the random data into SSD. SSDUP+ is able to identify the random data in the incoming I/O requests based on the proposed metric of *random factor*. Then, an adaptive algorithm is proposed to dynamically adjust the threshold of the random factor. The data with the randomness higher than the threshold will be cached to SSD. Moreover, the requests in SSD are sorted and flushed to HDD using a pipeline mechanism. Given the fact that multiple applications may generate a complex I/O access pattern, a traffic-aware flushing strategy is further designed to reduce the I/O interference between the flushed data and the data being written to HDD directly. To maintain the data sequence, an AVL tree data structure is used to manage the buffered data. With all above techniques, SSDUP+ can achieve the desired I/O performance with a much smaller SSD space.

We have implemented a prototype of SSDUP+ in OrangeFS to validate the design and evaluate its performance benefits. The methodology proposed in SSDUP+ is generic and is applicable in other file systems. Because our scheme is designed to be deployed at the server side, we do not need to consider the I/O patterns of individual applications.

We conduct the experiments with three widely used benchmarks, IOR, HPIO, and MPI-Tile-IO. The experimental results show that SSDUP+ can save 50% of SSD space on average while delivering similar performance as other common burst buffer systems. SSDUP+ can also save 20% of SSD space on average, comparing with SSDUP. Especially, when the SSD capacity is insufficient to accommodate all data, SSDUP+ can improve the performance (I/O throughput) by 20% to 30%, comparing with the common burst buffer scheme and SSDUP.

In the near future, we plan to broaden the adoption of SSDUP+. We plan to port SSDUP+ to the Lustre [4] file system and make full use of SSDUP+ for HPC storage systems. We also plan to apply the design methodologies in SSDUP+ to new devices such as non-volatile memory and new hybrid storage architectures.

REFERENCES

- [1] Jens Axboe. 2003. *Completely Fair Queueing (CFQ) Scheduler*. Retrieved from <http://en.wikipedia.org/wiki/CFQ>.
- [2] Jens Axboe. 2003. *Noop Scheduler*. Retrieved from <http://en.wikipedia.org/wiki/Noop>.
- [3] 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 *Proceedings of the International Conference on Supercomputing (SC'09)*. 21.

- [4] Peter Braam. 2003. *Lustre File System*. Retrieved from http://wiki.lustre.org/Main_Page.
- [5] 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 *Proceedings of the International Conference on Supercomputing (SC'08)*. 44.
- [6] Hsung-Pin Chang, Jhih-Cheng Luo, and Da-Wei Chang. 2016. A load-balancing data caching scheme in multi-tiered storage systems. In *Proceedings of the 2016 IEEE 18th International Conference on High Performance Computing and Communications, the IEEE 14th International Conference on Smart City, and the IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS'16)*. IEEE, 124–127.
- [7] Feng Chen, David A. Koufaty, and Xiaodong Zhang. 2011. Hystor: Making the best use of solid state drives in high performance storage systems. In *Proceedings of the International Conference on Supercomputing*. ACM, 22–32.
- [8] Y. Chen, X.-H. Sun, R. Thakur, H. Song, and H. Jin. 2010. Improving parallel I/O performance with data layout awareness. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'10)*. 302–311.
- [9] 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 *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'06)*. 49–49.
- [10] 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 *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing (HPDC'09)*. 31–38.
- [11] H. Fu, J. Liao, J. Yang, L. Wang, X. Huang, C. Yang, W. Xue, F. Qiao, W. Zhao, X. Yin, C. Hou, W. Ge, J. Zhang, Y. Wang, and G. Yang. 2016. The Sunway TaihuLight supercomputer: System and applications. *Inf. Sci.* 59, 7, Article 072001 (2016), 16 pages.
- [12] Jaehyun Han, Donghun Koo, Glenn K. Lockwood, Jaehwan Lee, Hyeonsang Eom, and Soonwook Hwang. 2017. Accelerating a burst buffer via user-level i/o isolation. In *Proceedings of the 2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 245–255.
- [13] S. He, X.-H. Sun, and B. Feng. 2014. S4D-Cache: Smart selective SSD cache for parallel I/O systems. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS'14)*. 514–523.
- [14] 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 *Proceedings of the IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'13)*. 111–120.
- [15] H. Huang, W. Hung, and K. Shin. 2005. FS2: Dynamic data replication in free disk space for improving disk performance and energy consumption. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'05)*. 263–276.
- [16] Sai Huang, Qingsong Wei, Dan Feng, Jianxi Chen, and Cheng Chen. 2016. Improving flash-based disk cache with lazy adaptive replacement. *ACM Trans. Stor.* 12, 2 (2016), 8.
- [17] IBM. 2006. *Boost Application Performance Using Asynchronous I/O*. Retrieved from <https://www.ibm.com/developerworks/library/l-async>.
- [18] Hartford N. Gunn Jr. 1970. *PBS*. Retrieved from <http://pbspro.org>.
- [19] Lawrence Livermore National Laboratory. 2010. *Slurm Workload Manager*. Retrieved from <http://slurm.schedmd.com>.
- [20] Tonglin Li, Xiaobing Zhou, Kevin Brandstatter, Dongfang Zhao, Ke Wang, Anupam Rajendran, Zhao Zhang, and Ioan Raicu. 2013. ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table. In *Proceedings of the IEEE 27th International Parallel and Distributed Processing Symposium (IPDPS'13)*. IEEE, 775–787.
- [21] Xiangke Liao, Liquan Xiao, Canqun Yang, and Yutong Lu. 2014. MilkyWay-2 supercomputer: System and application. *Front. Comput. Sci.* 8, 3 (2014), 345–356.
- [22] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Gridler, A. Crume, and C. Maltzahn. 2012. On the role of burst buffers in leadership-class storage systems. In *Proceedings of the International Conference on Massive Storage Systems and Technology (MSST'12)*. 1–11.
- [23] Y. Liu, R. Gunasekaran, X. Ma, and S. Vazhkudai. 2014. Automatic identification of application I/O signatures from noisy server-side traces. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'14)*. 213–228.
- [24] LLNL. 2003. *Interleaved Or Random (IOR) Benchmarks*. Retrieved from <https://github.com/LLNL/ior>.
- [25] 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 *Proceedings of the ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC'15)*. 33–44.
- [26] University of Chicago. 2002. *High-performance and Widely Portable Implementation of the Message Passing Interface (MPI) Standard*. Retrieved from <http://www.mpich.org>.
- [27] Oracle. 2012. *Sun Grid Engine*. Retrieved from https://en.wikipedia.org/wiki/Oracle_Grid_Engine.
- [28] Michael Papka, Susan Coghlan, Eric Isaacs, Mark Peters, and Paul Messina. 2013. *Mira: Argonne's 10-petaflops Supercomputer*. Technical Report. Argonne National Laboratory, Argonne, IL.

- [29] Raghunath Rajachandrasekar, Adam Moody, Kathryn Mohror, and Dhabaleswar K. Panda. 2013. A 1 PB/s file system to checkpoint three million MPI tasks. In *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing*. ACM, 143–154.
- [30] Rob Ross. 2001. Evaluating the performance of an underlying MPI-IO and file system implementation under a non-contiguous access workload. Retrieved from <http://www.mcs.anl.gov/research/projects/pio-benchmark/>.
- [31] Robert Ross. 2015. *Orange File System*. Retrieved from <http://www.orangefs.org>.
- [32] Reza Salkhordeh, Shahriar Ebrahimi, and Hossein Asadi. 2018. ReCA: An efficient reconfigurable cache architecture for storage systems with online workload characterization. *IEEE Trans. Parallel Distrib. Syst.* 29, 7 (2018), 1605–1620.
- [33] Wolfram Schenck, Salem El Sayed, Maciej Foszczynski, Wilhelm Homberg, and Dirk Pleiter. 2017. Evaluation and performance modeling of a burst buffer solution. *ACM SIGOPS Operat. Syst. Rev.* 50, 2 (2017), 12–26.
- [34] 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 the Industrial Control Systems Cyber Security Conference (ICS'17)*. 27:1–27:10.
- [35] Kun Tang, Ping Huang, Xubin He, Tao Lu, Sudharshan S. Vazhkudai, and Devesh Tiwari. 2017. Toward managing hpc burst buffers effectively: Draining strategy to regulate bursty i/o behavior. In *Proceedings of the 2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'17)*. IEEE, 87–98.
- [36] L. Tang, Q. Huang, W. Lloyd, S. Kumar, and K. Li. 2015. RIPQ: Advanced photo caching on flash for Facebook. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'15)*. 373–386.
- [37] R. Thakur, W. Gropp, and E. Lusk. 1999. Data sieving and collective I/O in ROMIO. In *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation*. 182–189.
- [38] Teng Wang, Adam Moody, Yue Zhu, Kathryn Mohror, Kento Sato, Tanzima Islam, and Weikuan Yu. 2017. Metakv: A key-value store for metadata management of distributed burst buffers. In *Proceedings of the 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS'17)*. IEEE, 1174–1183.
- [39] Teng Wang, Sarp Oral, Yandong Wang, Brad Settlemyer, Scott Atchley, and Weikuan Yu. 2014. Burstmem: A high-performance burst buffer system for scientific applications. In *Proceedings of the IEEE International Conference on Big Data (Big Data'14)*. IEEE, 71–79.
- [40] Teng Wang, W. Yu, K. Sato, A. Moody, and K. Mohror. 2016. *BurstFS: A Distributed Burst Buffer File System for Scientific Applications*. Technical Report. Lawrence Livermore National Laboratory, Livermore, CA.
- [41] Y. Wang and D. Kaeli. 2003. Profile-guided I/O partitioning. In *Proceedings of the Industrial Control Systems Cyber Security Conference (ICS'03)*. 252–260.
- [42] Z. Wang, X. Shi, H. Jin, S. Wu, and Y. Chen. 2014. Iteration based collective I/O strategy for parallel I/O systems. In *Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid'14)*. 287–294.
- [43] N. Xiao, Z. Chen, F. Liu, M. Lai, and F. Long. 2011. P3Stor: A parallel, durable flash-based SSD for enterprise-scale storage systems. *Inf. Sci.* 54, 6, Article 1129 (2011), 1129–1141 pages.
- [44] Feng Ye, Jianxi Chen, Xuejiao Fang, Jieqiong Li, and Dan Feng. 2015. A regional popularity-aware cache replacement algorithm to improve the performance and lifetime of SSD-based disk cache. In *Proceedings of the 2015 IEEE International Conference on Networking, Architecture and Storage (NAS'15)*. IEEE, 45–53.
- [45] 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 *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'13)*. 345–356.
- [46] X. Zhang, K. Davis, and S. Jiang. 2010. IOrchestrator: Improving the performance of multi-node I/O systems via inter-server coordination. In *Proceedings of the International Conference on Supercomputing (SC'10)*. 1–11.
- [47] X. Zhang, K. Davis, and S. Jiang. 2012. iTransformer: Using SSD to improve disk scheduling for high-performance I/O. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'12)*. 715–726.
- [48] X. Zhang, K. Davis, and S. Jiang. 2012. Opportunistic data-driven execution of parallel programs for efficient I/O services. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'12)*. 330–341.
- [49] X. Zhang and S. Jiang. 2010. InterferenceRemoval: Removing interference of disk access for MPI programs through data replication. In *Proceedings of the Industrial Control Systems Cyber Security Conference (ICS'10)*. 223–232.
- [50] X. Zhang, K. Liu, K. Davis, and S. Jiang. 2013. iBridge: Improving unaligned parallel file access with solid-state drives. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'13)*. 381–392.
- [51] Dongfang Zhao, Zhao Zhang, Xiaobing Zhou, Tonglin Li, Ke Wang, Dries Kimpe, Philip Carns, Robert Ross, and Ioan Raicu. 2014. Fusions: Toward supporting data-intensive scientific applications on extreme-scale high-performance computing systems. In *Proceedings of the IEEE International Conference on Big Data (Big Data'14)*. IEEE, 61–70.

Received February 2019; revised December 2019; accepted December 2019