공학석사 학위논문

An Empirical Implementation of I/O Separation Scheme for Burst Buffers in HPC Systems

고성능 컴퓨팅 시스템에서 버스트 버퍼를 위한 I/O 분리 기법의 실증적 구현

2019 년 8 월

서울대학교 대학원
컴퓨터 공학부
구 동 훈

Abstract

To meet the exascale I/O requirements in the High-Performance Computing (HPC), a new I/O subsystem, named *Burst Buffer*, based on non-volatile memory, has been developed. However, the diverse HPC workloads and the bursty I/O pattern cause severe data fragmentation to SSDs, which creates the need for expensive garbage collection (GC) and also increase the number of bytes actually written to SSD. The new multi-stream feature in SSDs offers an option to reduce the cost of garbage collection. In this paper, we leverage this multi-stream feature to group the I/O streams based on the user IDs and implement this strategy in a burst buffer we call *BIOS*, short for Burst Buffer with an I/O Separation scheme. Furthermore, to optimize the I/O separation scheme in burst buffer environments, we propose a stream-aware scheduling policy based on burst buffer pools in workload manager and implement the real burst buffer system, *BIOS framework*, by integrating the BIOS with workload manager. We evaluate the BIOS and framework with a burst buffer I/O traces from Cori Supercomputer including a diverse set of applications. We also disclose and analyze the benefits and limitations of using I/O separation scheme in HPC systems. Experimental results show that the BIOS could improve the performance by 1.44× on average and reduce the Write Amplification Factor (WAF) by up to 1.20×, and prove that the framework can keep on the benefits of the I/O separation scheme in the HPC environment.

Keywords: Burst Buffer, Multi-streamed SSD, I/O Separation, Stream-Aware, Evaluation

Student Number: 2017-20331

Contents

List of Figures

Chapter 1

Introduction

Recently, the data-intensive scientific domains have become a new application field, and the I/O becomes more and more challenging due to the increasing volume of scientific data up to several petabytes and complexity of scientific workload. Consequently, existing HPC Parallel File Systems (PFSs) based on disk cannot satisfy the I/O requirements. In response to the demand for high I/O performance in an HPC environment, most supercomputers have been introduced the SSD-based file systems. For example, Cray has developed a Burst Buffer[3], and deployed it at NERSC (National Energy Research Scientific Computing Center) since 2015[4]. As a high-performance storage layer, burst buffers have effectively handled the bursty I/Os in many commercial HPC systems.

Unfortunately, burst buffers have the problem arising from a characteristic of the NAND flash memory: Garbage Collection (GC) overheads caused by a difference in operation unit between write/read (page-level) and erase (block-level). The GC overhead is an additional copy operation for preserving valid pages in the GC operation that secures the empty block, which affects adversely performance and lifetime of a flash device when GC operation occurs frequently[5][6]. In particular, SSDs in burst buffer cope with a large amount of concurrent and complex I/Os from thousands of users and hundreds of thousands of scientific application, because burst buffers are used for absorbing the bursty I/O traffic as a shared resource in HPC systems[7]. In HPC environments, SSDs of burst buffer, as a result, would be exposed to frequent GC operations, which can lead the losses of performance and endurance[8].

In relation to research on burst buffer, they mainly have focused on study for addressing the I/O bottleneck problems in HPC systems[3, 4, 9], a study on burst buffer in local nodes for scalable write performance[10, 37], a study on I/O scheduling in burst buffer[11, 8], and study on burst buffer resource management[12]. To the best of our knowledge, no existing study focused on improving performance and endurance of the burst buffer itself except for our previous work[13]. To mitigate the GC overheads for SSDs in burst buffer, our previous work proposed the user-level I/O isolation by using multi-streamed SSD[14] which allocates same flash block for I/Os in the same stream ID. In that work, we uncovered the expected performance reduction in an SSD-based burst buffer and demonstrated that effectiveness of user-level I/O isolation in burst buffer. However, this is not enough to verify the effect of user-level I/O isolation in real burst buffer environments in that actual burst buffer is not implemented and results are based on a single device.

In this paper, we implement the burst buffer supporting user-transparent I/O separation scheme, named *BIOS*, to verify the effect of multi-stream feature in real burst buffer environments. Further, to optimize I/O separation scheme in burst buffers, we integrate the BIOS with workload manager considering multi-stream, as a real burst buffer system, called *BIOS framework*. Based on implemented system, we explore the benefits and limitations of the BIOS and framework with extensive experiments. To implement the burst buffer with I/O separation scheme, we leverage the multi-streamed SSDs to group the I/O streams based on the user-stream mapping. This burst buffer is to assign the flash-memory blocks to each user exclusively, being performed transparently to users. In HPC burst buffer environments, we found that benefits of I/O separation scheme can be reduced due to user ID-based stream allocation and a limited number of available streams (e.g., 4 to 16) in multi-streamed SSD[14],[15],[16]. To overcome these problems, we propose managing the burst buffer resource as burst buffer pools and the stream-aware scheduling policy in workload manager and finally, implement the framework by integrating these with the BIOS. This framework optimizes the benefits of I/O separation scheme by alleviating the interference caused by data striping and the problem of skewed stream allocation.

To validate the effectiveness of the BIOS and framework in realistic HPC environments, we evaluate these not only using synthetic workload but also using real HPC applications and burst buffer I/O traces obtained from the Cori supercomputer at NERSC. In our experiments, the BIOS shows the up to 1.44× increased performance and up to 1.20× reduced write amplification factor (WAF) compared to an existing burst buffer. With the evaluation of the framework, it shows that framework can keep on the benefits of the BIOS and can be deployed in the HPC systems.

Our main contributions are as follows: (1) We implement the burst buffer supporting an user-transparent I/O separation scheme in the real system composed of multiple devices and multiple nodes; (2) To optimize the benefits of I/O separation scheme, we implement the framework integrated with workload manager managing the burst buffer resource efficiently based on stream-aware scheduling policy; (3) Through extensive experiments such as HPC applications and burst buffer I/O traces from the Cori supercomputer, we evaluate the BIOS and framework, thereby uncovering and analyzing the benefits and limitations of the I/O separation scheme in real burst buffer environments.

In the rest of this paper, Section 2 presents the background and challenges of the work presented in this paper and Section 3 and 4 describes the I/O separation scheme and framework. Section 5 shows the experiment results to demonstrate the effectiveness of proposed scheme and framework compared with legacy burst buffer. Section 6 summarizes findings and insights from evaluation. Section 7 and 8 show the discussion and related work respectively, and finally Section 9 summarizes our conclusion and future work.

Chapter 2

Background and Challenges

2.1 Burst Buffer

As the performance of computing resource increases dramatically and emerging of massive HPC systems, traditionally HPC systems with disk-based PFSs are in trouble with satisfying the I/O requirements. As the result of demand for better I/O performance in HPC environments, the burst buffer technology has emerged in recent years. The burst buffer as a high-performance SSD tier efficiently handles bursty I/O that is not handled by the PFSs, located between the compute node and the PFS. In accordance with the purpose of the burst buffer, it handles the traditional HPC checkpoint-restart workloads which have large streaming I/O efficiently and is also being used for temporary staging space and in-transit data processing recently[17]. However, due to these various uses, burst buffer faces a performance challenge for small files and varied I/O patterns[4]. In HPC burst buffer, currently, there are two representative burst buffer architectures: local burst buffer and shared burst buffer. One is to be implemented in each compute node as a local burst buffer. The other is to be located in a shared resource such as dedicated burst buffer nodes. With benefits from ease of maintenance and deployment, shared burst buffer architecture is commonly used in state-of-the-art commercial burst buffers such as Cray's DataWarp[18] and DDN's IME[19]. Specifically, on Cori supercomputer at NERSC, DataWarp burst buffer is treated as high-speed storage resources and managed by a batch scheduler, SLURM[29]. Through the workload manager, the shared burst buffer is allocated to users on a per-job or short-term basis.

2.2 Write Amplification in SSDs

The NAND flash memory used as a storage for burst buffer has an inherent characteristic that erase has coarser granularity (block-level) compared to granularity (page-level) of write and read operations. To secure the free blocks, the garbage collection is performed in SSD, causing a redundant copy operation for keeping valid pages. The characteristics of out-place update and differences in block erase and page write/read operations in flash memory cause the superfluous copy operations. So the amount of data written to the storage media is amplified in SSD when GC operation occurs. This phenomenon is called *write amplification*[20] and the amplification ratio is denoted as write amplification factor (WAF). This write amplification factor is calculated as follows:

$$\mathrm{WAF} = \frac{\text{the amount of data written in NAND flash}}{\text{the amount of data sent from the client}} \tag{2.1}$$

The cost of the erase operation is considerably expensive due to the write amplification caused by GC operation. When SSD being excessively used, an SSD incurs more frequent GC operation at the same time, which degrade the write performance. Therefore, SSDs in burst buffer that used as a shared resource and handles the bursty I/O can be more affected the effect of GC overheads.

2.3 Multi-streamed SSD

A multi-streamed SSD has been proposed to reduce the garbage collection(GC) overheads in flash memory by mapping the data with different lifetime to different stream. To map the data to stream ID, a multi-streamed SSD makes the host system to send write request with stream ID to SSD since the host can provide adequate information about data lifetime. Therefore, the stream ID may be a hint about data lifetime, which allowing an SSD to place the data with same stream ID into the same flash blocks. With this mechanism, all data associated with stream is expected to be invalidated at the same time in the same flash blocks, and thus this feature increases the probability of the data within the blocks to be removed together, which reduces the probability of a number of copy operations during the GC operation. A multi-streamed SSD, as a result, provides not only an enhanced device lifetime but also improved performance and constant latency via the multi-stream mechanism. In the NVMe 1.3 specification[38], the multi-stream feature is introduced as a form of directives. This feature is defined for write commands, allowing the host system to carry the stream to the controller by using NVMe commands. As the multi-stream feature is officially adopted in the NVMe interface, the multi-stream feature is expected to be available in many future NVMe devices.

2.4 Challenges of Multi-stream Feature in Burst Buffers

In this work, we aim at implementing a robust burst buffer from a large amount of I/Os using a multi-stream feature. In other word, by separating the user's

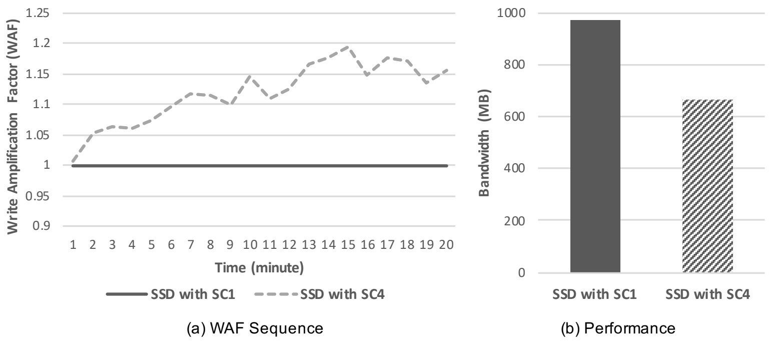


Figure 2.1: Impact on multi-streamed SSD when a number of stripe count is different for the same load

I/Os from the others' I/Os via multi-streamed SSDs, we target to minimize the GC overheads in SSDs of burst buffer. The number of streams supported by the device, therefore, is important to reduce the GC overheads. For example, if multi-streamed SSD can provide an unlimited number of streams, all I/O streams can be allocated exclusive flash blocks, which can remove the GC overheads completely. Unfortunately, a multi-streamed SSD supports number of 4 to 16 streams due to implementation constraints related to a write buffering mechanism in an SSD[21]. In HPC environments, as a result, each stream ID is shared by multiple I/O streams, resulting in data mixed in flash block associated with same stream IDs and eventually weakens the benefits of the multi-stream feature. To address this problem, many research on multi-streamed SSD have tried to devise a method for stream allocation which maps data with a similar lifetime to the stream IDs[15],[22],[16],[23]. In this paper, we present intuitive and effective criteria for stream allocation considering the burst buffer environments.

In this paper, we consider a shared burst buffer system that uses striping I/O for high performance and is located in a dedicated node. Assuming we build a burst buffer using multi-streamed SSDs in this system, in theory, we can use a number of streams equal to # of devices $\times$ # of supported streams to completely isolate the I/O stream from other I/O streams in this burst buffer. However, the number of streams that can actually be used would be decreased since the striping I/O segments the file and stores each segment on different SSDs, each file uses a stream assigned to that file in all multi-streamed SSDs, which brings the effect like using only the number of streams supported by a single multi-streamed SSD when all SSDs participate in striping I/O.

To demonstrate the impact of data striping to multi-streamed SSD, we perform the workload with 8 FIO[32] threads in each of the 4 nodes to local SSD and grouped SSDs with RAID 0 respectively, and these results about the local SSD (SSD with SC1) and one of the grouped SSDs (SSD with SC4) after preconditioning to warming up the devices are showed in figure 2.1. Since we use 8 stream IDs to isolate the I/O threads, the local SSD perfectly separates the I/O threads in each node, while each SSD grouped with RAID 0 handles the 32 I/O threads due to striping I/O; all stream IDs in SSD is shared by 4 I/O threads each. As a result, WAF in SSD with SC4 is increased as time passed, adversely affecting the performance despite using the multi-stream feature. In this context, it showed that simply applying the stream allocation strategy in burst buffer is not sufficient to keep on maximum benefits of the multi-stream feature.

Chapter 3

I/O Separation Scheme in Burst Buffer

In this section, we first present the intuitive and effective criterion to distinguish the data lifetime in burst buffer environments and using this criterion, we implement the burst buffer providing a multi-stream feature to user transparently, named *BIOS*. Before addressing the challenge of multi-stream feature in the burst buffer environment, it is important the implementation of an effective burst buffer must be preceded to maximize multi-stream capability.

3.1 Stream Allocation Criteria

The mapping method between data and stream ID is the most important factor in optimizing the multi-stream mechanism. In shared burst buffer systems, burst buffer is assigned to user on a per-job basis through the workload manager. The output data of the job are stored in the burst buffer while the job is running, and burst buffer and output data are deleted together when the job finishes.

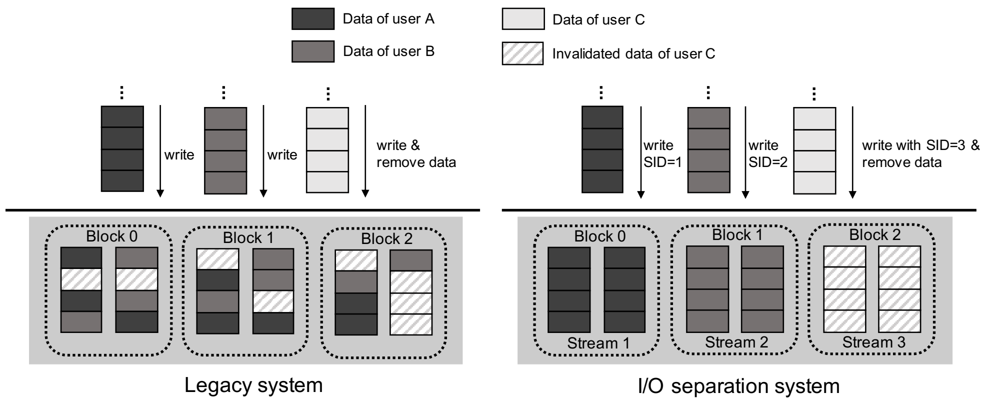


Figure 3.1: Legacy System v.s I/O Separation System

Namely, the lifetime of the user's data stored in the burst buffers is generally equal to the job execution time[26]. The user ID can be the key to intuitively and effectively distinguish the data lifetime in the burst buffer environments. Therefore, we take the *user ID* as a classification unit for mapping the data with a different lifetime to disparate stream IDs. Based on this insight, we present user ID-based stream management in burst buffer, I/O separation scheme.

To prove the effectiveness of an I/O separation scheme, we assume the situation that a user's data is deleted while other users are still writing data to burst buffer; this situation is common in shared burst buffer environments. Figure 3.1 illustrates the effect of an I/O separation scheme compared with the legacy system. More specifically, it shows the different flash-memory block layout in a legacy and an I/O separation system when GC operation is performed on block #2 respectively. In case of the legacy system, to perform the GC operation for block #2, it needs 4 copy and 1 erase operation. On the other hand, the I/O separation system can complete the GC operation with just 1 erase operation. It's intuitive that the I/O separation system can reduce the GC overheads efficiently compared to the legacy system in burst buffer environments.

3.2 Implementation

To implement the burst buffer with I/O separation scheme, we modified the open-source based distributed file system, BeeGFS [28], to allow it to allocate stream IDs by leveraging user IDs, uid in Linux, and to pass the stream IDs by using *fadvise()* which passes it down to SSDs through file inode; a stream allocation is performed per file descriptor when a file is opened.

The logic for implementing I/O separation scheme is represented by algorithm 1 that describes the user ID-based stream management in the BeeGFS storage daemon located in each node. In the algorithm 1, the stream and access time mapping table are used for stream management; these are data structure for managing the stream IDs efficiently, returning the stream ID and access time corresponding to user ID respectively. When a user writes the data, a BeeGFS checks the list whether the user ID is registered in the stream mapping table. If not, the BeeGFS tries to find the idle stream ID first. If all stream IDs are in use, it finds the least used stream ID as an alternative. And then, the BeeGFS registers the (user ID, stream ID) pair to stream mapping table. After stream allocation process, the selected stream ID is applied to file descriptor through the modified *fadvise()*. And then, user's stream ID access time is updated in the access time mapping table and a threshold value is incremented; these variables are used as a criterion for whether or not the stream ID will be reclaimed and periodically to perform a retrieval function respectively. When the threshold value reaches the setting value, the retrieval function is performed. The retrieval function checks the all registered user's stream ID access time, removing all data related to corresponding the user ID such as the (user ID, stream ID) pair in the stream mapping table when user has not used the stream ID for a certain period of time. After retrieval function is finished, the threshold value

Algorithm 1 An User ID-based Stream Management

```
1: streamID = getStreamID_from_table(userID)
2:
3: if $streamID = NULL$ then
4:     if $Find\_idle\_streamID = True$ then
5:         Allocate the $streamID$
6:     else
7:         Allocate the least used $streamID$
8:     end if
9:     Register the (userID, streamID) pair to table
10: end if
11:
12: fadvise(fd, streamID)
13: update_access_time(userID)
14: threshold++
15:
16: if $threshold > setting\_value$ then
17:     Check all registered user's stream ID access time
18:     if $Not\_recently\_accessed$ then
19:         Retrieve the streamID
20:         Remove the (userID, streamID) pair in the table
21:     end if
22:     Init threshold
23: end if
```

is initialized.

In summary, the overall process of data flow from BeeGFS to multi-streamed SSDs is as follow. When the user writes the file to a BeeGFS, it divides the file into chunks for data striping, sending the chunks to nodes which belong to BeeGFS. Then, a BeeGFS storage daemon which services the storage function in all nodes belonging to BeeGFS assigns the stream ID for chunk using algorithm 1. The chunks with stream ID are passed down to a multi-streamed SSD in each node. And then, each SSD stores the received chunk into the flash blocks associated with stream ID. This system, named *BIOS*, that provides an I/O separation scheme offers a multi-stream feature to users transparently, thus all users in the BIOS can experience these benefits without modifying the application's code.

3.3 Limitations of User ID-based Stream Allocation

As discussed above, a user ID is the most suitable information about data lifetime when we considered burst buffer environments. However, the problem can occur when the same user submits multiple jobs because burst buffer supporting user ID-based stream allocation assigns the same stream ID to all jobs from the same user. In this case, skewed stream allocation can occur in the BIOS. In summary, the problems of skewed stream allocation along with the reduction of available stream incurred by data striping are seen as factors in reducing the benefits of I/O separation scheme.

Chapter 4

BIOS Framework

In this section, we propose the burst buffer system, *BIOS Framework*, which optimizes the I/O separation scheme by integrating the BIOS with workload manager. Through integrating with the workload manager, we can treat burst buffer as high-speed storage resource, optimizing the use of burst buffer (e.g., Cray DataWarp). To improve the limitations of I/O separation scheme in the BIOS framework, we organize resource unit for burst buffer allocation by grouping the devices together in a burst buffer pools and design the stream-aware scheduling policy which evenly balances the load to SSDs while decreasing the contention of stream resource in workload manager.

4.1 Support in Workload Manager

In existing HPC systems such as supercomputer, the workload manager is essential component for cluster/resource management and job scheduling. Most of resources including the burst buffer in HPC systems are managed and assigned to a user for an amount of time by workload manager. For example, in the case of ephemeral burst buffers as of NERSC Cori, the allocation process is mainly managed by the SLURM workload manager. Therefore, supporting burst buffer in the workload manager is essential for implementation of the real burst buffer system.

To implement the framework that supports the BIOS, we used the SLURM workload manager with some modification. Specifically, in order for SLURM to support the BIOS, we added the SLURM directive for the BIOS. As an argument of the directive, it includes a directory name, *dirname*, and size of burst buffer, *capacity*. In the *dirname* argument, it's used for directory name of burst buffer; directory name for job is created as a "user/dirname" to prevent the problem of name duplication between the users. A *capacity* is used for the size of burst buffer and for a baseline of setting the stripe count which is the number of SSDs to use. In current burst buffer on Cori, the allocation granularity is 20G, for example, when the *capacity* is set to 80GB, the stripe count will be 4 (four burst buffer nodes will be allocated). In our BIOS framework, we configured the maximum stripe count as 4. By specifying the BIOS directive, *#BIOS*, in a SLURM batch script, users can be assigned the ephemeral burst buffer with an I/O separation scheme from the workload manager.

4.2 Burst Buffer Pools

In order to reduce the contention caused by data striping and to prevent skewed stream allocation when a lot of jobs are submitted from the same user, we utilized the storage pools function [31] in BeeGFS to split the burst buffer resources into different burst buffer pools, as shown in figure 4.1. By utilizing the burst buffer pools, we can limit the range of data striping and choose the physical

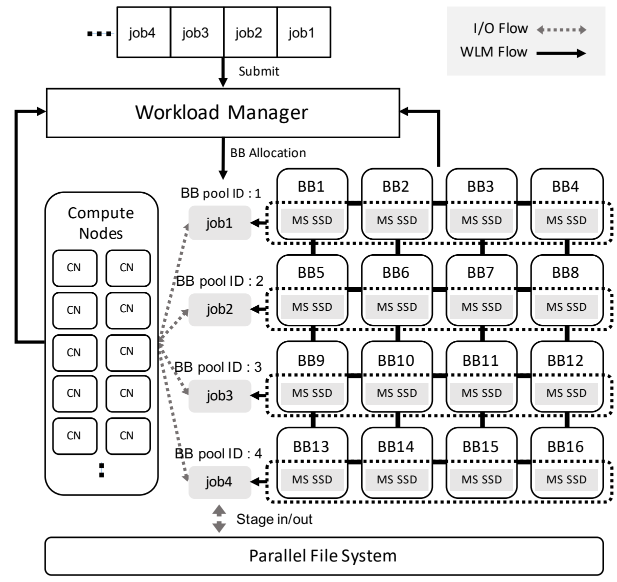


Figure 4.1: An overview of BIOS framework

storage device for burst buffer, which gives a chance to mitigate the contention from data striping and to solve the skewed stream allocation via scheduling technique. Figure 4.1 shows the BIOS framework with 4 burst buffer pool IDs composed of 16 storage devices. In this framework, the workload manager determines the pool ID first (see next subsection for details) and then creates the burst buffer within the pool for each job.

But, using a burst buffer pools may bring the inefficient use of burst buffer resources when demand for burst buffer is usually low. Because burst buffer resource per job is limited in burst buffer pool even if there are idle resources remaining. However, since the utilization of the burst buffer has recently been increasing, the inefficiency from low utilization is negligible. Next we discuss the stream-aware scheduling policy based on burst buffer pools.

4.3 Stream-Aware Scheduling Policy

Ultimately, we implement the real burst buffer system which manages the burst buffer resource by workload manager. To efficiently manage the resource, it is important to use an appropriate scheduling scheme. For example, in burst buffer system, a round robin scheduling is used to assign it to the user[35]. In resource scheduling, the most important thing is to ensure resources are used evenly. In order to design the scheduling policy to optimize the I/O separation scheme in the burst buffer system, we first consider distributing the load evenly to the SSDs in burst buffer pools. Moreover, we also consider the stream ID-use-state to mitigate the problem of skewed stream allocation and contention in stream ID. Based on these considerations, we propose the stream-aware scheduling policy which considers not only load balancing but also user ID-based stream allocation in the burst buffer system.

The algorithm 2 describes a stream-aware scheduling policy in the workload manager. When jobs are submitted, the workload manager first tries to find an idle pool and creates the burst buffer on it. If an idle pool does not exist, the workload manager allocates a pool that is not used the same user ID by checking each stream ID-use-state. As discussed in Section 3.1, we use the user ID-based stream allocation in the BIOS, thus jobs from the same user are assigned the different pool IDs, guaranteeing that each job uses their own stream IDs on different SSDs. If all pool IDs are being used from the same user, the workload manager assigns the least used pool ID to the job in order to reduce interference as much as possible.

Algorithm 2 A Stream-Aware Scheduling Policy

```
1: pool_ID = get_idle_pool_id()
2:
3: if $pool\_ID \neq NULL$ then
4:     Return $pool\_ID$
5: else
6:     user_ID = get_uid_from_job_id(jobID)
7:     pool_ID = not_overlap_pool_id(userID)
8:
9:     if $pool\_ID = NULL$ then
10:        pool_ID = min_used_pool_id()
11:        Return $pool\_ID$
12:    else
13:        Return $pool\_ID$
14:    end if
15: end if
```

4.4 Workflow of BIOS Framework

Figure 4.1 shows the overall structure of a BIOS framework which includes the BIOS, workload manager, and parallel file system (PFS). The BIOS framework also supports the staging function: stage-in/out which move the data between burst buffers and PFSs. When jobs are submitted to the workload manager, it figures out the user ID of job and the all pool IDs use-state, assigning a burst buffer to the job. Before starting the job execution, stage-in is performed when the user demands this function; stage-in is necessary if files of frequently accessed or large size are needed. After stage-in, a job starts to run. When output files are written to allocated burst buffer after job execution, BeeGFS storage daemon in each node assigns the stream ID to output files by identifying the user ID. And then, when the job is terminated, the workload manager performs the stage-out transferring the files stored in the burst buffer to the PFS and terminates the job by removing the burst buffer. The above process is simply performed by specifying a BIOS directive in the SLURM batch script. As a result, the BIOS framework we implement provides the service of a complete burst buffer system.

Chapter 5

Evaluation

5.1 Experiment Setup

For conducting the experiments, we used the testbed which consists of 4 nodes. Each of its nodes has an Intel Xeon E5-2620 v4 processor with 2-way 8-core and 64GB RAM. For multi-streamed SSD, we used four 960GB Samsung PM963 SSD supporting 8 configurable streams with modified firmware to support the multi-stream feature. Before the start of every experiment, we initialized all SSDs using NVMe command: *nvme format*. To collect the steady-state I/O throughput and WAF, we measured the results after 30 minutes in all experiments. To measure I/O throughput and WAF, we used *nmon* [39] and nvme command, *nvme smart-log*, respectively.

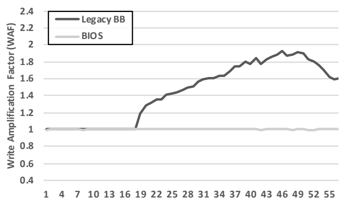
5.2 Evaluation with Synthetic Workload

In order to understand the effect of I/O separation scheme in burst buffer, we generated the synthetic workload with bursty I/O by using FIO [32] benchmark and compared to existing burst buffer, Legacy BB, on various metrics. The synthetic workload simulates 8 users on each node, 2 threads per user; our testbed consists of 4 nodes, so total users are 32. Each user is assigned a different burst buffer capacity and each thread writes the 64MB file repeatedly during the runtime. If user exceeds the allocated burst buffer capacity, 15% files are removed from the oldest one; therefore, we can consider users who are allocated a similar size of burst buffer capacity have similar data lifetime.

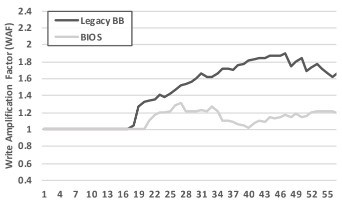
As mentioned before, we consider shared burst buffer architecture which aggregates the bandwidth across multiple devices via data striping. We configured a shared burst buffer with four nodes having a single multi-streamed SSD. Despite using four multi-streamed SSDs supporting 8 streams, we can actually use not 32 streams but 8 streams in this burst buffer due to data striping. Each stream ID, therefore, is shared by 4 users in this experiment, thus it is important how users who have various data lifetime are grouped into stream IDs. Depending on the degree of grouping the users who have similar data lifetime into the same stream ID, we configured three grouping configurations described below.

- Configuration 1; all users in each stream ID have the same data lifetime
- Configuration 2; some users in each stream ID have the same data lifetime
- Configuration 3; all users in each stream ID have the completely different data lifetime

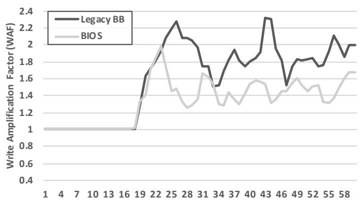
Under these configurations, we evaluated the BIOS and Legacy BB on synthetic workload for an hour respectively. For each configuration, figure 5.1 illustrates the WAF sequence of Legacy BB and the BIOS. On the Legacy BB, WAF is increased up to almost 2 or more as time goes on regardless of grouping configurations. Actually, in perspective of Legacy BB, grouping the user who has



(a) Configuration 1



(b) Configuration 2

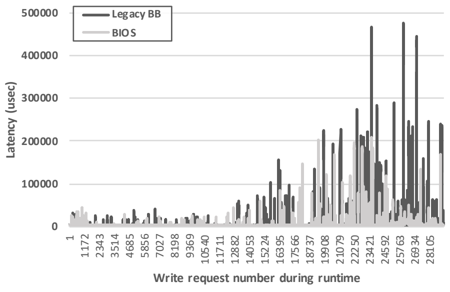


(c) Configuration 3

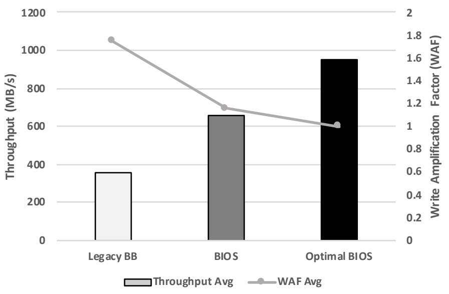
Figure 5.1: WAF sequence of Legacy BB and BIOS with different configuration

similar data lifetime is meaningless because all data is eventually mixed into the SSD block regardless of these configurations. It indicates that the Legacy BB constantly suffered GC overheads, which hurts the SSD's endurance and performance. On the contrary, the BIOS shows low and stable WAF value except to result in configuration 3. But, even though WAF is increased in configuration 3, its average WAF is lower than all case of Legacy BB because the contention of SSD blocks is less than in SSD blocks of Legacy BB shared by 32 users. The I/O separation scheme which decreases the degree of block sharing from multiple users in SSD brings the benefits for improving the endurance in an SSD. Besides, if stream ID mapping is well optimized by considering the data lifetime, its benefits can be maximized.

Another benefit of the BIOS is to reduce the long-tail latency. To present the effect of the I/O separation scheme for tail latency, we measured the latency for every 64MB write request in this workload. Figure 5.2a shows the results of latency on Legacy BB and BIOS with configuration 2. As shown in Figure 5.2a, write requests in the BIOS become 1.72×, 1.93× and 2.04× faster compared to Legacy BB, at 90th, 95th and 99th percentiles, respectively. Since the GC operation blocks the processing of incoming I/O requests until this operation is done[5], long-tail latency occurs in Legacy BB with severe GC overhead in



(a) Write latency between the BIOS and Legacy BB during runtime



(b) Average performance and WAF on Legacy BB, BIOS and Optimal BIOS

Figure 5.2: Results of long-tail latency and average performance and WAF

this experiment. As a result, burst buffer with I/O separation scheme which mitigates the GC overheads provides up to 2 times improved long-tail latency compared with existing burst buffer system.

To assess the I/O separation scheme in burst buffer from a throughput point of view, we measured the throughput in these experiments performed to Legacy with configuration 2, BIOS with configuration 2, and BIOS with configuration 1 as a Legacy BB, BIOS, and optimal BIOS respectively, and calculated the average throughput after 30 minutes from the start of experiment to understand the degree of impact of GC overheads on performance. Figure 5.2b shows the results of average throughput and WAF from the time GC operation is generated in earnest. These results indicate that applying the I/O separation scheme to conventional burst buffer can improve the endurance and performance by 1.51× and 1.83× and if stream mapping is optimized by considering data lifetime, then the BIOS can improve the benefits of endurance and performance up to 1.74× and 2.66× compared to Legacy BB.

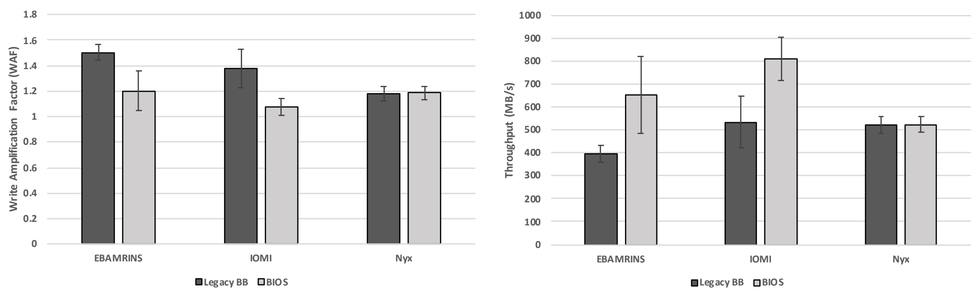
5.3 Evaluation with HPC Applications

As a next step for a more realistic evaluation, we used three HPC applications: EBAMRINS [1], IOMI [40] and Nyx [2]. Among them, EBAMRINS and IOMI are built on Chombo which is a high-performance block-structured adaptive mesh refinement framework for solving partial differential equations in complex geometries. Nyx is N-body hydrodynamic cosmological simulation application that uses a massively parallel AMR code for computational cosmology. These are representative scientific applications in HPC environments. The output files generated by each application are described as follows.

- EBAMRINS : single checkpoint and plot file with 41MB and 65MB size respectively
- IOMI : single hdf5 file around 292MB size
- Nyx : checkpoint and plot directory consisting of 13 and 10 files, total size of each directory is 131MB and 117MB respectively

To load the enough I/O to burst buffer, we configured workloads to generate output files after each step of computation. We assumed the 8 users per node in total four nodes and performed the experiments three times for each application. Therefore, total 32 users perform the striping I/O to the burst buffer which consists of four nodes and each stream ID is shared by 4 users in that our multi-streamed SSD supports the 8 streams. For the method of file deletion and burst buffer capacity allocation, it is the same as we used it in the evaluation with the synthetic workload as described in Section 5.2. In the case of EBAMRINS experiments, we used 3 threads per user in order to give sufficient I/O load.

Figure 5.3 compares the BIOS with Legacy BB in terms of average WAF and write throughput on representative HPC applications. The results in EBAM-



(a) Average WAF (b) Average Throughput

Figure 5.3: Average throughput and WAF while performing experiments using HPC applications

RINS show that average WAF and throughput for Legacy BB are 1.5 and 395MB/s whereas WAF for BIOS are 1.2 (1.25×) and 652MB/s (1.65×) respectively. In EBAMRINS, the BIOS shows relatively higher performance deviation compared with other application's results since we used more total thread than other application experiments; stream ID is shared by more threads in EBAMRINS experiment. Therefore, depending on how well the users with similar data lifetime can be grouped together, the BIOS performance in EBAMRINS may show variable performance trends. Despite large performance deviations, the BIOS guarantees better performance than Legacy BB even in the worst case. In IOMI application, the BIOS shows that 1.07 and 810MB/s for WAF and throughput respectively, whereas Legacy BB shows results of 1.37 and 534MB/s. The BIOS in IOMI represents the far high performance compared to experimental results in other applications due to simple I/O characteristic of IOMI that generates a large single file, which allows the BIOS to keep on the benefits even though each stream ID is shared by multiple users. On the other hand, the Legacy BB results indicates that existing burst buffer can be troubled in handling the concurrent I/Os from many users despite simple I/O pattern. Consequently, the BIOS compared to Legacy BB improves WAF and throughput by 1.27× and 1.51× on average respectively. In the case of Nyx experiment, both burst buffers show approximately the same experimental results for WAF and throughput. In Nyx experiment, we observed that GC overheads on the BIOS go up to the same level as Legacy BB since Nyx application generates multiple small files frequently and concurrently. The BIOS did not show any advantage compared to Legacy BB in Nyx application. However, we expect this case to be improved in the BIOS framework. Overall, these results show that the average WAF and I/O throughput in all application experiments are improved by 1.17× and 1.37× in the BIOS when compared to the Legacy BB.

5.4 Evaluation with Emulated Workload

For a more realistic evaluation beyond the synthetic workload, we evaluated the BIOS with Darshan logs of Cori's burst buffer provided by NERSC. Darshan can profile the application's POSIX and MPI-IO function calls with minimum overheads. The Darshan log can accurately report the I/O pattern and I/O cost over the job's lifetime. More importantly, the emulated workload can represent the real I/O pattern on a production burst buffer at a national HPC facility. We selected 32 workloads which are write-intensive from the Darshan logs; the average size of write operations ranged from 1KB to 10MB. To replay these workloads, we used the workload emulator [13] and assume that 32 users randomly select a workload and they are granted the burst buffer with different capacity. Since the darshan log doesn't record the remove operation which triggers by the workload manager instead of the application itself, we simply remove the data in the same way as in the synthetic tests. With the combination of different workloads for each test, we could reveal the different HPC

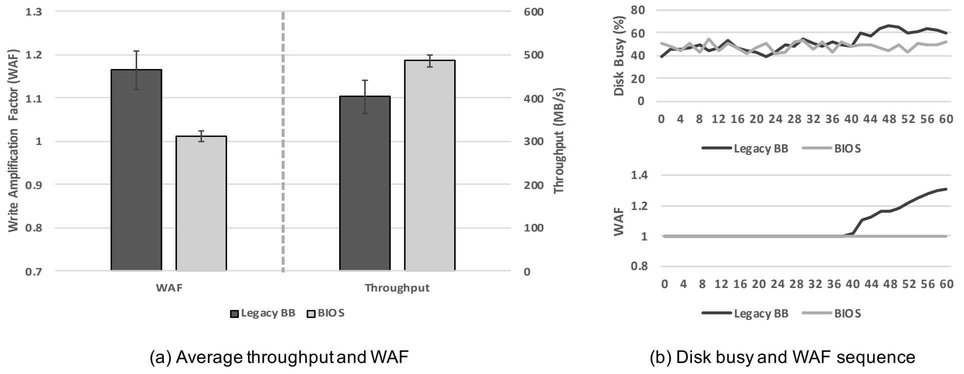


Figure 5.4: Average throughput and write amplification factor when emulating supercomputing workload and disk-busy and WAF sequence of Legacy BB and BIOS during runtime

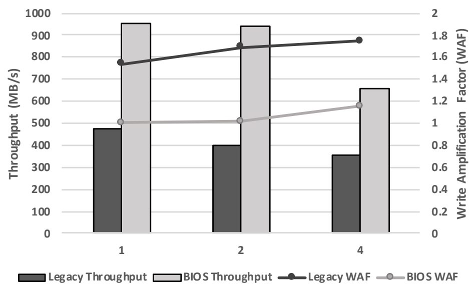
workload pattern. In this subsection, we formed four combinations of workloads and conducted the experiments separately.

Figure 5.4a illustrates the average WAF and throughput on emulated workloads. In the case of WAF results, average WAF in Legacy BB is 1.16 and it is improved to 1.01 by the BIOS. Moreover, I/O throughput is improved by 1.20× with the BIOS. Even though the emulated workloads do not have enough I/O to saturate the burst buffer bandwidth, we observed that the WAF starts to increase in Legacy BB even when the disk-busy is only around 40% to 50%, as shown in Figure 5.4b. This phenomenon is aroused from the complex I/O patterns in HPC and the concurrent I/Os from multiple users. These results indicate the BIOS prevents the degradation of performance and the device lifetime by reducing the GC overheads in such complex I/O workloads. Currently, the burst buffer has been introduced and experimentally operated in the HPC systems without being fully utilized yet[36]. However, the complete introduction and generalization of the burst buffer technology in the near future will result in more bursty I/O, and the benefits of the BIOS will become more significant.

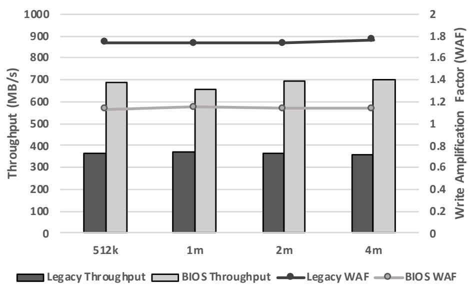
5.5 Evaluation with Different Striping Configuration

In general, most burst buffers use the data striping which is the ability to stripe data across multiple storage devices for providing high-performance. Although striping I/O provides high-performance, it accelerates the data fragmentation in the SSD blocks, which can adversely affect the performance and lifetime of SSDs in burst buffer. To identify the impact of data striping on the burst buffer, we performed the experiments by changing the stripe settings about stripe count and stripe size. In order to give the same I/O load to each SSD regardless of the stripe count, we configured the ratio of the number of SSDs to the number of I/O users equally; 8 users per SSD perform I/O. In experiment for stripe count, we set the stripe size and RDMA buffer size to the value of 512KB and 768KB separately, and for stripe size experiment, we fixed the value of stripe count and RDMA buffer size to 4 and 16MB respectively.

Figure 5.5a illustrates the average throughput and WAF while changing the stripe count. On the Legacy BB, as stripe count increases to 1,2 and 4, WAF is increased to 1.53, 1.69 and 1.74 and throughput shows that it is decreased to 477MB/s, 397MB/s and 358MB/s respectively. Each SSD handles the same amount of I/O regardless of the stripe count, but as the stripe count increases, the number of users processed by each SSD increases, resulting in more data being mixed in the SSD block. As a result, this leads to an increase of GC overheads as the stripe count increase. For the same reason, when the stripe count is 4 in the BIOS, WAF is increased up to 1.15 and its throughput is also decreased. Nevertheless, the WAF and throughput in the BIOS show 1.53×, 1.66× and 1.51×, and 2×, 2.3× and 1.83× improvement in case of 1,2 and 4



(a) Results on Stripe Count



(b) Results on Stripe Size

Figure 5.5: Results of performance and WAF with different stripe configuration

stripe count respectively.

Figure 5.5b shows the experimental results when the stripe size is 512KB, 1MB, 2MB and 4MB. Both results in Legacy BB and the BIOS show around the same WAF and throughput regardless of stripe size. Although we did not present the results for different RDMA size in this paper, we also performed the same experiment with 768KB RDMA size. But, RDMA size also did not affect performance. These results demonstrate that stripe size and RDMA size show little impact on SSD performance and lifetime in bursty I/O environments.

5.6 Evaluation on BIOS Framework

Until now, we have evaluated the BIOS directly with extensive experiments. From those experiments, we verified that I/O separation scheme is effective and also has a limitations incurred by a limited number of streams. In this subsection, we confirm whether BIOS framework optimizes the I/O separation scheme in burst buffer environments.

To evaluate the BIOS framework, we configured the realistic supercomputing environments using multi-component workload composed of five work-

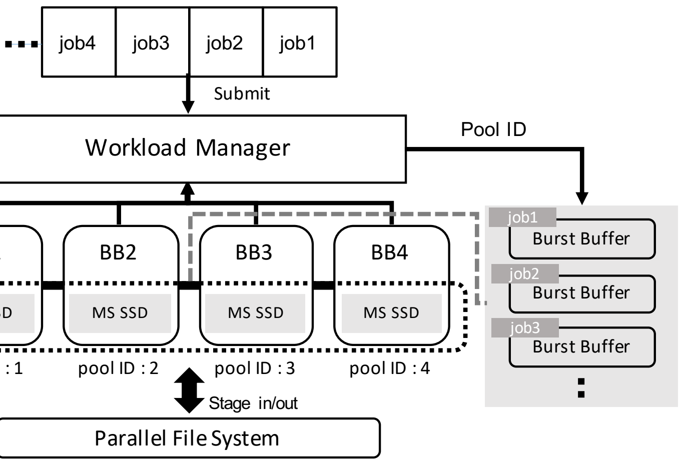
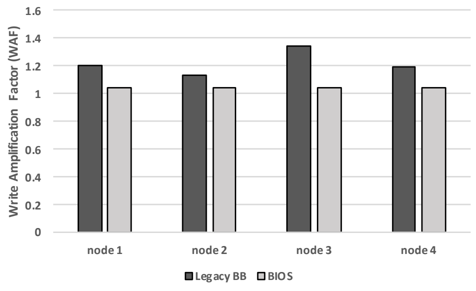


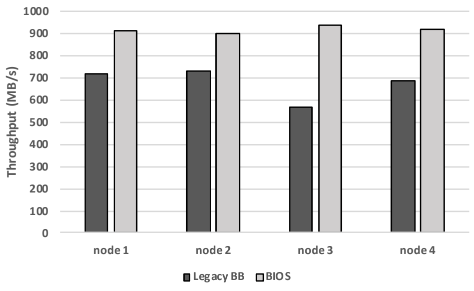
Figure 5.6: BIOS framework on our testbed

loads: Nyx-MiniSB, Nyx-Santabarbara, Chombo-EBAMRINS, Chombo-IOMI, and IOR[33]. We assume the 8 users with submitting the 4 jobs to workload manager; total 32 jobs are submitted and running on BIOS framework together while performing the IOR applications with large files in order to fill the burst buffer capacity. As shown in Figure 5.6, we built the BIOS framework on our testbed which has four nodes. Due to the limited number of nodes in our testbed, we configured the four burst buffer pools using four SSDs; each burst buffer pool consists of single SSD, so each job only uses a single one in this framework. For example, when a job is submitted to the workload manager, BIOS framework assigns the burst buffer pool ID based on stream-aware scheduling policy and provides burst buffer using the single SSD in allocated pool ID.

Figure 5.7 illustrates the average WAF and throughput for all SSDs in the framework. In Figure 5.7a and 5.7b, the WAF and throughput in BIOS show 1.04 and 914MB/s on average while presenting the 1.21 and 675MB/s in Legacy BB respectively. The BIOS compared with Legacy BB improves WAF and throughput by 1.17× and 1.35× respectively, which indicates that the BIOS



(a) Average WAF



(b) Average Throughput

Figure 5.7: Average WAF and throughput for the storage device of each node in diverse HPC applications

framework effectively works as designed in realistic supercomputing environments. Under the BIOS framework, each job can exclusively use the SSD without interference from other jobs, so the BIOS removes the GC overheads in the SSD despite running with a lot of jobs, providing consistent benefits of I/O separation scheme.

In Figure 5.7, the performance results appear a slight difference between nodes. This is because each node handles the different combination of applications due to the pool ID allocation by the workload manager. In addition, interesting things is that the WAF in the BIOS is not 1.00 but 1.04 even though each job uses the own stream ID. 0.04% copy overheads come from the different timing which marks invalid state about deleted data between the file system and an SSD. When a user delete the data, the file system marks data block as not use in, but the SSD does not know which SSD's page should be marked as invalid until the operating system notifies. This results in unnecessary copy operation for a worthless data, and it becomes the GC overheads despite using the SSD completely alone. However, such a degree of overheads is negligible and can be completely removed by using trim command [24].

Chapter 6

Summary and Lessons Learned

In this section, we summarize the findings and insights from an extensive evaluation performed in burst buffer with an I/O separation scheme and the BIOS framework.

6.1 An I/O Separation Scheme in Burst Buffer

6.1.1 Evaluation with Synthetic Workload

Applying I/O separation scheme to burst buffer can mitigate garbage collection overheads efficiently, improving I/O throughput, device lifetime, service level objective. When data grouping is precisely grouped considering data lifetime, the benefits of I/O separation scheme are maximized.

6.1.2 Evaluation with HPC Applications

In real HPC applications with diverse I/O patterns, the I/O separation scheme demonstrates that GC overheads are mitigated in SSDs of burst buffer. Although the burst buffer with I/O separation scheme provides inconsistent benefits depending on I/O patterns, it ensures better performance than existing burst buffers.

6.1.3 Evaluation with Emulated Workload

Currently, the I/O load in real supercomputing workload is not enough to saturate the SSDs of burst buffer, representing 40% to 50% disk-busy since burst buffers have been introduced and experimentally operated in HPC systems. Despite insufficient load, the complex I/O patterns bring the GC overheads in existing burst buffers, whereas the BIOS completely eliminates the GC overheads incurred by complex I/O patterns.

6.1.4 Evaluation with Striping Configurations

Striping I/O pattern used in shared burst buffer accelerates the data fragmentation in SSD blocks, undermining the benefits of I/O separation scheme. Management is required to keep on the benefits of I/O separation scheme in burst buffers. Without addressing the problem of GC overheads in SSD, trivial optimization such as stripe size or RDMA buffer size is meaningless for optimizing the I/O separation scheme in burst buffer environments.

6.2 A BIOS Framework

6.2.1 Evaluation with Real Burst Buffer Environments

The real burst buffer system, BIOS framework, demonstrates that I/O separation scheme can be optimized in burst buffer environments through the burst buffer pools and stream-aware scheduling policy. The burst buffer pools reduces interference caused by data striping. The stream-aware scheduling policy solves the skewed stream allocation for jobs of the same user while balancing the load. The BIOS framework can reduce the GC overhead that is no longer reduced by limitations from the BIOS.

Chapter 7

Discussion

7.1 Limited Number of Nodes

In this paper, even though we conducted the evaluation of the BIOS framework on a small-scale cluster consisting of a limited number of nodes, the effectiveness of BIOS can be expanded to large-scale clusters. Because the BIOS framework is to manage the burst buffer resource by burst buffer pools, it can be adopted in any size cluster by adjusting the size of burst buffer pools. For example, if we assume the scaled-up cluster with 16 burst buffer nodes, the BIOS framework will look like Figure 4.1. We can configure the four burst buffer pool IDs grouped into four SSDs considering the scale of the system. Assuming multiple jobs are running in same pool ID in this cluster, we can consider this pool ID is equivalent to the same circumstance in the BIOS experiments of Section 5.2 to 5.4, because these experiments are performed on four nodes using the striping I/O. Although we use a limited number of nodes to demonstrate the effect of BIOS framework, its benefits can be generalized to a large-scale cluster.

7.2 Advanced BIOS Framework

In this paper, we configured the fixed number of storage device for each burst buffer pool ID. The management of the BIOS in uniform pool IDs allows us to make it easy about allocating burst buffer resources and to ensure a certain level of performance. However, as a demand of I/O requirement of each HPC application varies, the framework with uniform pool IDs can lose the chance to use more efficiently such as grouping the I/O streams with similar characteristic into pool IDs. Therefore, by organizing the pool IDs with the variable number of SSDs, we expect our framework will manage storage more efficiently and further improve overall system performance.

We can also utilize the information of job's lifetime in workload manager in order to predict when the data of the job will actually be erased. If we use this information suitably, we can ideally group the data into the stream IDs; Theoretically, all data in the same stream will be erased at the same time, which will show the BIOS result shown in Figure 5.1a. We expect that utilizing the job's lifetime information enable to make BIOS more effective.

Chapter 8

Related work

As a beginning of the burst buffer study, Liu *et al.* explored the potential of burst buffer and demonstrated its effectiveness in a large-scale HPC system. In subsequent studies, the study for handling the I/O bottleneck problem in HPC systems[4, 9], the study of new burst buffer architecture[10, 37] and the study of scheduling policy for I/O between burst buffer and PFSs, or burst buffer resource[8, 11, 12] have been progressed. Our work is based on the burst buffer architecture represented in Bhimji *et al.*[4], and we improve the performance and endurance problem that could be aroused in this system by mitigating the GC overheads.

The multi-streamed feature has introduced for mitigating the GC overheads of SSD. A large body of prior research has been conducted on how to effectively leverage multi-stream mechanism. There are some strategies to leverage the multi-stream feature, which typically includes mapping data from applications level [15], [25], file systems layer [22], and the block layer [16]. In case of application-level customization, the multi-stream feature can be optimized via the understanding of data lifetime in application although it comes to cost as a compatibility issue to all applications requiring the multi-stream feature. The others using abstracted information support multi-stream feature with transparency to applications but have limitations in optimizing all applications compared to application-level customization. A recent study by Kim *et al.* has proposed the automatic stream management based on application-level information, program context. Our work also provides automatic stream management in burst buffer by utilizing intuitive and effective data lifetime information based on burst buffer I/O characteristics.

Chapter 9

Conclusions

With emerging burst buffers, the HPC systems with disk-based PFSs could satisfy the I/O requirement. However, the burst buffers also cannot completely meet the I/O requirement since some I/O characteristics of the HPC environment may cause the write amplification in SSDs of burst buffer, leading to the performance degradation of the entire burst buffer system. To address this problem, we have proposed the BIOS, a Burst Buffer with an I/O separation scheme based on multi-streamed SSDs and a framework managing the BIOS efficiently in burst buffer environments. By assigning the stream to each user transparently, the BIOS provides the illusion that user uses their own SSDs; actually, each user uses exclusive NAND blocks in the same SSD, this mitigates the write amplification in the SSDs, possibly enhancing the performance by an average of 1.44$\times$ and reducing WAF by 1.20$\times$, as shown in our extensive experiments. In addition, the framework manages the BIOS using stream-aware scheduling policy based on burst buffer pools, optimizing the I/O separation scheme regardless of cluster scale. The results of all those tests demonstrate a very promising future of using the BIOS framework in HPC environments. As future work, we plan to enhance our framework by improving the stage in/out using dcp[34] or gnu parallel[27] utility for large files, and by supporting the the transparent viewing, we will ultimately implement the BIOS framework supporting transparent caching mode.

Bibliography

[1] G. Miller and D. Trebotich, “An embedded boundary method for the navier–stokes equations on a time-dependent domain,” *Communications in Applied Mathematics and Computational Science*, vol. 7, no. 1, pp. 1–31, 2011.

[2] A. S. Almgren, J. B. Bell, M. J. Lijewski, Z. Luki'c, and E. Van Andel,” *Nyx: A massively parallel amr code for computational cosmology*, The Astrophysical Journal, vol. 765, no. 1, p. 39, 2013.

[3] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume,and C. Maltzahn, “On the role of burst buffers in leadership-class storagesystems,” in *Mass Storage Systems and Technologies (MSST*, 2012 IEEE28th Symposium on. IEEE, 2012, pp. 1–11.

[4] W. Bhimji, D. Bard, M. Romanus, D. Paul, A. Ovsyannikov, B. Friesen,M. Bryson, J. Correa, G. K. Lockwood, V. Tsulaiaet al., “Accelerating science with the nersc burst buffer early user program,” , 2016

[5] S. Yan, H. Li, M. Hao, M. H. Tong, S. Sundararaman, A. A. Chien,and H. S. Gunawi, “tiny-tail flash: Near-perfect elimination of garbage collection

tail latencies in nand ssds," in *ACM Transactions on Storage(TOS)*, vol. 13, no. 3, p. 22, 2017.

[6] Y. Luo, Y. Cai, S. Ghose, J. Choi, and O. Mutlu, "Warm: Improving nand flash memory lifetime with write-hotness aware retention management," in *2015 31st Symposium on Mass Storage Systems and Technologies(MSST)*, IEEE, 2015, pp. 1–14.

[7] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock, "I/O performance challenges at leadership scale," in *High Performance Computing Networking, Storage and Analysis, Proceedings of the Conferenceon*, IEEE, 2009, pp. 1–12.

[8] S. Thapaliya, P. Bangalore, J. Lofstead, K. Mohror, and A. Moody, "Managing i/o interference in a shared burst buffer system," in *201645th International Conference on Parallel Processing (ICPP)*, IEEE,2016, pp. 416–425.

[9] J. Lofstead, I. Jimenez, C. Maltzahn, Q. Koziol, J. Bent, and E. Barton, "Daos and friends: a proposal for an exascale storage system," in *SC'16:Proceedings of the International Conference for High Performance-Computing, Networking, Storage and Analysis*, IEEE, 2016, pp. 585–596.

[10] J. Bent, S. Faibish, J. Ahrens, G. Grider, J. Patchett, P. Tzelnic, andJ. Woodring, "jitter-free co-processing on a prototype exascale storagestack," in *012 IEEE 28th Symposium on Mass Storage Systems andTechnologies (MSST)*, IEEE, 2012, pp. 1–5.

[11] K. Tang, P. Huang, X. He, T. Lu, S. S. Vazhkudai, and D. Tiwari, "Toward managing hpc burst buffers effectively: Draining strategy to regulate bursty i/o behavior," in *2017 IEEE 25th InternationalSymposium on*

Modeling, Analysis, and Simulation of Computer andTelecommunication Systems (MASCOTS), IEEE, 2017, pp. 87–98.

[12] W. Liang, Y. Chen, J. Liu, and H. An, "Contention-aware resource scheduling for burst buffer systems," in *Proceedings of the 47th Inter-national Conference on Parallel Processing Companion*, ACM, 2018,p. 32.

[13] J. Han, D. Koo, G. K. Lockwood, J. Lee, H. Eom, and S. Hwang, "Accelerating a burst buffer via user-level i/o isolation," in *2017 IEEEInternational Conference on Cluster Computing (CLUSTER)*, IEEE,2017, pp. 245–255.

[14] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho, "The multi-streamed solid-state drive," in *HotStorage*, 2014.

[15] F. Yang, K. Dou, S. Chen, M. Hou, J.-U. Kang, and S. Cho, "Optimizing nosql db on flash: A case study of rocksdb," in *Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and TrustedComputing and 2015 IEEE 15th Intl Conf on Scalable Computing andCommunications and Its Associated Workshops (UIC-ATC-ScalCom) 2015 IEEE 12th Intl Conf on*, IEEE, 2015, pp. 1062–1069.

[16] J. Yang, R. Pandurangan, C. Choi, and V. Balakrishnan, "Autostream: automatic stream management for multi-streamed ssds," in *Proceedings of the 10th ACM International Systems and Storage Conference*, ACM, 2017, p. 3.

[17] C. S. Daley, D. Ghoshal, G. K. Lockwood, S. Dosanjh, L. Ramakrishnan, and N. J. Wright, "Performance characterization of scientific workflows for the optimal use of burst buffers," in *Future Generation Computer Systems*, 2017.

[18] D. Henseler, B. Landsteiner, D. Petesch, C. Wright, and N. J. Wright, “Architecture and design of cray datawarp,” in *Cray User Group CUG*, 2016.

[19] W. Schenck, S. El Sayed, M. Foszczynski, W. Homberg, and D. Pleiter, “Early evaluation of the ”infinite memory engine” burst buffer solution,” in *International Conference on High Performance Computing*, Springer, 2016, pp. 604–615.

[20] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, “Write amplification analysis in flash-based solid state drives,” in *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, ACM, 2009, p. 10.

[21] T. Kim, D. Hong, S. S. Hahn, M. Chun, S. Lee, J. Hwang, J. Lee, and J. Kim, “Fully automatic stream management for multi-streamed ssds using program contexts,” in *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 2019, pp. 295–308.

[22] E. Rho, K. Joshi, S.-U. Shin, N. J. Shetty, J.-Y. Hwang, S. Cho, D. D. Lee, and J. Jeong, “Fstream: managing flash streams in the file system',' in *16th USENIX Conference on File and Storage Technologies*, 2018, p.257.

[23] T. Kim, S. S. Hahn, S. Lee, J. Hwang, J. Lee, and J. Kim, “Pcstream: automatic stream allocation using program contexts,” in *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

[24] J. Kim, H. Kim, S. Lee, and Y. Won, “Ftl design for trim command,” in *The Fifth International Workshop on Software Support for Portable Storage*, 2010, pp. 7–12

[25] T.-D. Nguyen and S.-W. Lee, "Optimizing mongodb using multistreamed ssd," in *Proceedings of the 7th International Conference on Emerging Databases*, Springer, 2018, pp. 1–13.

[26] Using the Burst Buffer on Cori, ttps://www.nersc.gov/assets/Uploads/Burst-Buffer-tutorial.pdf

[27] GNU Parallel: The Command-Line Power Tool, https://www.usenix.org/system/files/login/articles/105438-Tange.pdf

[28] BeeGFS Documentation, https://www.beegfs.io/wiki/TableOfContents

[29] Slurm: A Highly Scalable Workload Manager, https://github.com/SchedMD/slurm

[30] BeeGFS On Demand, https://www.beegfs.io/wiki/BeeOND

[31] BeeGFS Storage Pools, https://www.beegfs.io/wiki/StoragePools

[32] Flexible I/O Tester, https://github.com/axboe/fio

[33] Parallel filesystem I/O benchmark, https://github.com/LLNL/ior

[34] distributed file copy program, https://github.com/hpc/dcp

[35] How to use the Burst Buffer, https://www.nersc.gov/users/computational-systems/cori/burst-buffer/example-batch-scripts/

[36] Burst Buffer Early User Program, http://www.nersc.gov/users/computational-systems/cori/burst-buffer/burst-buffer-early-user-program/

[37] T. Wang, W. Yu, K. Sato, A. Moody, and K. Mohror, "Burstfs: Adistributed burst buffer file system for scientific applications," LawrenceLivermore National Lab.(LLNL), Livermore, CA (United States), Tech.Rep., 2016.

[38] A. Huffman, “Nvm express revision 1.3 specification,” NVM Express, Inc., 2012.

[39] N. Griffiths, “nmon performance: A free tool to analyze aix and linux performance,” 2003.

[40] D. Devendran, S. Byna, B. Dong, B. Van Straalen, H. Johansen, N. Keen, and N. F. Samatova, “ Collective i/o optimizations for adaptive mesh refinement data writes on lustre file system,” 2016.

초록

고성능 컴퓨팅의 엑사스케일 I/O 요구 사항을 충족시키기 위해 비휘발성 메모리 기반의 새로운 I/O 서브시스템인 버스트 버퍼가 등장하였다. 그러나 HPC 환경의 다양한 HPC 워크로드 및 버스티한 I/O 패턴의 특성은 고성능 낸드 디바이스로 구성된 버스트 버퍼에 심각한 데이터 단편화를 유발시킨다. 이는 SSD에 빈번한 가비지 컬렉션 연산을 발생시켜 실제 SSD에 기록되는 바이트 수를 증가시키게 된다. 최근 SSD의 새로운 기능인 멀티 스트림은 가비지 컬렉션 비용을 절감 시킬 수 있는 옵션으로 등장하였다. 본 논문에서는 멀티 스트림 기능을 기반으로 사용자 ID를 이용하여 I/O를 분리시키고, BIOS라 부르는 I/O 분리 스킴을 지원하는 버스트 버퍼를 구현한다. 또한 버스트 버퍼 환경에서 I/O 분리 스킴을 최적화하기 위해 워크로드 매니저에서 버스트 버퍼 풀을 기반으로 하는 스트림 인식 스케줄링 정책을 제안하고 BIOS를 워크로드 매니저와 통합하여 실제 버스트 버퍼 시스템인 BIOS 프레임 워크를 구현한다. 우리는 다양한 응용 프로그램을 포함하여 Cori 슈퍼컴퓨터의 버스트 버퍼 I/O 로그파일을 사용하여 BIOS 및 프레임 워크를 평가하였다. 우리는 또한 HPC 시스템에서 I/O 분리 스킴의 이점과 한계를 확인하고 분석한다. 실험 결과, BIOS는 평균 144%의 성능 향상 그리고 쓰기 증 지수 (WAF)는 최대 120%까지 줄일 수 있었다. 또한 프레임 워크가 HPC 환경에서 I/O 분리 스킴의 이점을 계속 유지할 수 있음을 보였다.

주요어: 버스트 버퍼, 다중 스트림 디바이스, 입출력 분리, 스트림 인식, 평가
학번: 2017-20331