Title

Design and Implementation of Burst Buffer Over-Subscription Scheme for HPC Storage Systems

Authors

Bang, Jiwoo

Sim, Alexander

Lockwood, Glenn K

et al.

Publication Date

2023

DOI

10.1109/access.2022.3233829

IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 7 December 2022, accepted 23 December 2022, date of publication 3 January 2023, date of current version 11 January 2023.

Digital Object Identifier 10.1109/ACCESS.2022.3233829

RESEARCH ARTICLE

Design and Implementation of Burst Buffer Over-Subscription Scheme for HPC Storage Systems

JIWOO BANG[1], ALEXANDER SIM[2], (Senior Member, IEEE), GLENN K. LOCKWOOD[2], HYEONSANG EOM[1], AND HANUL SUNG[3]

[1]Department of Computer Science and Engineering, Seoul National University, Seoul 08826, South Korea
[2]Lawrence Berkeley National Laboratory, Computational Research Division, Berkeley, CA 94720, USA
[3]Department of Game Design and Development, Sangmyung University, Seoul 03016, South Korea

Corresponding author: Hanul Sung (sunghanul817@gmail.com)

This work was supported in part by the National Research Foundation through the PF Class Heterogeneous High Performance Computer Development under Grant NRF-2016M3C4A7952587; in part by the BK21 FOUR Intelligence Computing through the Department of Computer Science and Engineering, SNU, funded by the National Research Foundation of Korea (NRF) under Grant 4199990214639; in part by the Seoul Business Agency through the Optimizing GPGPU Scheduling and Processing in Multi-GPU Environments Based on Data Locality under Grant NRF-2021R1F1A1063438; in part by the Korea Institute of Science and Technology Information; in part by NRF funded by the Korean Government (MSIT) under Grant NRF-2022R1G1A1011433; in part by the Office of Advanced Scientific Computing Research, Office of Science, through the U.S. Department of Energy under Contract DE-AC02-05CH11231; and in part by the National Energy Research Scientific Computing Center (NERSC).

ABSTRACT Burst Buffer is widely used in supercomputer centers to bridge the performance gap between computational power and the high-performance I/O systems. The primary role of Burst Buffer is to temporarily absorb the bursty I/O and reduce the heavy access on Parallel File System (PFS). However, the job resource manager on High-Performance Computer (HPC) systems prefers to use a dedicated Burst Buffer allocation approach, which eventually leads to the severely underutilized Burst Buffer resource. To improve the efficiency of using the expensive Burst Buffer resource, we analyze the I/O patterns on Burst Buffer in depth. We propose Burst Buffer over-subscription allocation method, which improves Burst Buffer utilization by allowing each job to access Burst Buffer only during its I/O phases so that the jobs can overlap each other. Furthermore, we develop a new I/O congestion-aware scheduler and a transparent data management system between Burst Buffer and PFS. Our approach also reduces the memory overhead and improves the data persistence of the data management system by adapting the persistent memory. With the proposed approach, not only the Burst Buffer utilization can be improved, but also HPC applications can achieve high I/O performance by exploiting the powerful Burst Buffer hardware capabilities. Experimental results show that BBOS can improve Burst Buffer utilization by up to 120% while more stable and higher checkpoint performance is guaranteed even under high I/O loads compared to other state-of-the-art schedulers. Besides, our approach can improve the hit ratio of restart requests by up to 96.4% and provides up to 210% higher restart throughput on Burst Buffer.

INDEX TERMS Burst buffer, checkpoint, demotion, over-subscription, parallel file system, restart.

I. INTRODUCTION

As computational capability has grown over one petaflop, a large number of system components have been deployed in HPC systems, thereby resulting in increased overall system failures [1], [2], [3]. For a fail-safety purpose, HPC applications tend to aggressively utilize checkpoint and restart strategy, which is the most common fault tolerance mechanism. The checkpoint and restart mechanism inevitably generates bursty I/O, occupying 75%∼80% of total I/O traffic of overall

The associate editor coordinating the review of this manuscript and approving it for publication was Alberto Cano.

HPC system [2], [4]. Since PFS consists of cheap HDDs or low-end flash SSDs, heavy I/O accesses generated by checkpoint and restart operations makes it difficult for PFS to handle and eventually leads to low I/O performance. To alleviate the overhead on PFS and speed up the I/O performance, Burst Buffer composed of high-end flash SSDs (e.g., 3D XPoint SSD and NVMe SSD) [5], [6] has been introduced as a new storage tier located between compute nodes and PFS. Due to the substantial performance difference delivered by Burst Buffer and PFS, HPC users prefer to allocate dedicated Burst Buffer resource for a whole lifetime of the submitted jobs.

However, current dedicated Burst Buffer allocation mechanism leads to severely underutilized Burst Buffer resource for the following two reasons. First, HPC users prefer to request overabundant amount of Burst Buffer resources (e.g., up to six times [3]). According to the real-world log data collected from the NERSC (National Energy Research Scientific Computing Center) Cori supercomputer system, only 5% of overall Burst Buffer resource is actively used. The reasons for over-requesting the Burst Buffer resource include preventing possible I/O errors, improving the I/O performance, and avoiding the complicated data movement between Burst Buffer and PFS. Second, since HPC applications use Burst Buffer only during the I/O phases, Burst Buffer stays idle for the rest of the time. Although checkpoint dominates the I/O traffic of the HPC system, long interval exists between two successive checkpoint operations. As a result, allocated Burst Buffer resource is wasted for most of the time during the application lifetime.

In this paper, we propose an efficient HPC storage management approach using the Burst Buffer over-subscription allocation method, called BBOS (Burst Buffer Over-Subscription). To support the Burst Buffer over-subscription method in the HPC storage systems, we transparently manage data movement between Burst Buffer and PFS when scheduling the I/O jobs. The key idea behind BBOS is to utilize the characteristics of checkpoint and restart operations that occupy most of the I/O traffic in HPC storage systems. Since checkpoint and restart mechanism has specific I/O characteristics, using primitive data management approach such as a kernel data management approach between memory and storage layers or commonly used approaches for storage tiers within PFS can result in low performance. In this work, we introduce a new data placement scheduling policy between Burst Buffer and PFS that considers the characteristics of checkpoint and restart operations. To show the improved Burst Buffer utilization and checkpoint/restart performance with BBOS, we evaluate our approach in comparison to Cray DataWarp [7], the current representative HPC scheduler which uses the dedicated Burst Buffer allocation method, and Harmonia [8], which is the Burst Buffer based dynamic I/O scheduler in consideration of Burst Buffer over-subscription method. Compared to DataWarp, Burst Buffer utilization is improved by up to 120% while maintaining stable and high checkpoint performance by using the BBOS framework. Besides, our approach can provide high checkpoint performance and improve restart performance by up to 96.4% on Burst Buffer by utilizing the characteristics of checkpoint and restart operations.

In the previous work [9], we implemented the Burst Buffer over-subscription framework to solve the Burst Buffer under-utilization problem shown in most of the HPC systems. The BBOS framework consists of an I/O engine, a data management engine, and an in-memory key-value store to efficiently handle the checkpoint and restart operations of HPC applications. We extend the original work by using persistent memory on the BBOS framework. We implement an improved version of the framework using NVDIMM on the Redis in-memory key-value database. Specifically, the memory capacity can be increased with low cost and the data persistence is guaranteed even when power failure occurs. We show that there is no I/O performance degradation shown with NVDIMM-applied BBOS that stores most of the data in NVDIMM. We further extend the original work by evaluating the NVDIMM-applied BBOS design, showing the implementation details, and explaining the workflow of checkpoint, restart, and demotion operations when applying the BBOS framework on Burst Buffer.

Our contributions are as follows:

- We adopt the over-subscription Burst Buffer allocation method to efficiently utilize the Burst Buffer resource.
- We analyze the characteristics of checkpoint and restart operations of HPC applications in detail. We find that the existing data management approach does not consider checkpoint and restart characteristics of HPC applications, which results in low I/O performance.
- We propose BBOS, a novel HPC data management approach that provides high Burst Buffer utilization as well as stable and high checkpoint and restart performance. BBOS schedules I/O jobs, adjusts demotion threshold and I/O bandwidth of checkpoint and demotion adaptively, and manages data placement policy between Burst Buffer and PFS.
- We implement a BBOS prototype by adding multiple modules on GlusterFS. We utilize the persistent memory to lower the DRAM overhead and add data persistence when adopting the BBOS framework on Burst Buffer.

II. BACKGROUND AND MOTIVATION

A. UNDERUTILIZED BURST BUFFER

Burst Buffer is located in the intermediate layer between computational nodes and storage systems to absorb bursty I/O in HPC systems [10], [11]. Each Burst Buffer node consists of expensive hardware resources, such as high-speed storage media and high-speed network. Most of the supercomputers, including Cori supercomputer [12] at NERSC and Summit supercomputer at ORNL (Oak Ridge National Laboratory), allocate Burst Buffer resources by using a dedicated Burst Buffer allocation method. The users specify the desired capacity or desired nodes to be used for the applications and the specified space is provided by an HPC

scheduler [13], [14] during the whole lifetime of the applications. However, the dedicated allocation method leads to severe underutilization problem of Burst Buffer as the HPC users normally reserve Burst Buffer space larger than the actual capacity they need for the following reasons.

The application jobs fail with I/O error when there is not enough Burst Buffer capacity to handle the I/O. To avoid failure, the users are recommended by a supercomputer providers to request the surplus amount of Burst Buffer capacity [3]. Not only for the failure, but users may also request a bountiful capacity expecting higher performance as well. Another reason for overabundant requests arises from complicated data management in multi-tier HPC storage systems (i.e., local storage of a compute node, Burst Buffer, and PFS). Since current supercomputers manage Burst Buffer and PFS separately, the users are challenged with redundant and complicated management. For instance, if the users use a limited amount of Burst Buffer capacity for writing only one checkpoint, they should copy data manually from Burst Buffer to PFS at every end of the checkpoint phase to make Burst Buffer space for the next phase [15].

Performing the Burst Buffer reservation process without considering the characteristics of the checkpoint and restart operations is also the critical reason that causes resource underutilization problem. HPC applications perform checkpoint operations during a fixed amount of time [16], [17], [18], called checkpoint period, by repeating compute phase and I/O phase periodically. However, as the checkpoint period lasts from tens of minutes to tens of hours, expensive Burst Buffer resources stay idle during compute phases. Moreover, Burst Buffer needs to be reserved for at least twice as much the capacity for the checkpoint data since old checkpoint file should be kept until a new checkpoint file is completely written safely. If HPC users decide to store multiple versions of checkpoint files in Burst Buffer to increase data durability, Burst Buffer becomes severely underutilized as the rest of the old version files except the latest one are rarely accessed. The addressed problems caused by using the dedicated Burst Buffer allocation method motivate our over-subscription-based HPC storage management approach.

B. CHARACTERISTICS OF CHECKPOINT AND RESTART OPERATIONS

Unlike common applications, HPC applications have checkpoint and restart-related characteristics. To apply the Burst Buffer over-subscription method on the HPC storage system, a novel data management scheme needs to be developed considering the following five checkpoint and restart characteristics.

First, most of the HPC applications solve computationally intensive problems and perform checkpoint operations at a particular cycle. We observe that the total amount of the checkpoint written to Burst Buffer in a certain period, called Data Written Per Period (*DWPP*) in this paper, is kept quite steady. As so, it is possible to predict future *DWPP* of a job using the previous *DWPP* values run.

Second, each application has a specific checkpoint period and an intermediate time interval between two checkpoint operations. Thus, each application accesses the Burst Buffer only during a specific checkpoint period. For instance, HPC applications with short checkpoint periods access Burst Buffer more frequently than ones with long checkpoint periods.

Third, HPC applications tend to keep multiple versions of checkpoint files to increase data durability [19]. HPC users prefer to keep the old versions of checkpoint files without deleting them even though only the latest version of the checkpoint file is required in the restart process. Since users demand different degree of data reliability when they run the jobs, each application job maintains the different number of checkpoint versions.

Fourth, HPC applications have different failure rates. Failure is occurred by individual components, such as processors, disk, memory, power supplies, network, cooling systems, and the physical connections between them [20]. The large number of the components together unavoidably leads to frequent failures [21], [22]. The prior studies show that the Mean Time Between Failure (*MTBF*) on a single node is thousands of hours, while *MTBF* on a large-scale cluster with hundreds of nodes is dozens of hours. In other words, failure rates increase linearly with the number of nodes used by HPC applications [23], [24].

Lastly, there is no data locality across the checkpoint files of HPC applications. Temporal locality does not exist across checkpoint files, because the checkpoint file is accessed only when the failure occurs. Also, spatial locality does not exist across checkpoint files. The checkpoint files will not be accessed unless failure occurs, even if they are stored around the other requested checkpoint file.

C. PROBLEM ANALYSIS

Different from the dedicated Burst Buffer allocation method, the Burst Buffer over-subscription method allocates more space to the applications than the actual capacity. To make this possible, the applications are allowed to access Burst Buffer only during the I/O phases. Applications in the computation phase should yield Burst Buffer to other applications in the I/O phase by moving data from Burst Buffer to PFS. Therefore, an efficient data management approach between Burst Buffer and PFS that does not degrade the overall performance is required. There are several previous works that propose efficient data management policy in the multi-tiered system [3], [8], [25], [26]. However, these approaches are not suitable for the HPC storage system where checkpoint dominates most of the I/O traffic for the following reasons.

The previous works use static demotion threshold without considering the amount of data to be moved between storage tiers. With the prior approaches, demotion is operated only when Burst Buffer is idle. When the total used capacity of Burst Buffer reaches the threshold, demotion has to be operated concurrently with checkpoint operations. Using the over-subscription method, the number of jobs accessing Burst

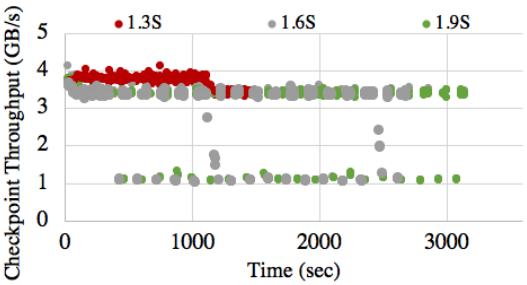

FIGURE 1. Checkpoint performance with different *DWPP*s. Checkpoint files with total 1.3 times, 1.6 times, and 1.9 times the size of Burst Buffer are written in a checkpoint period with 1.3*S*, 1.6*S*, and 1.9*S* *DWPP*s.

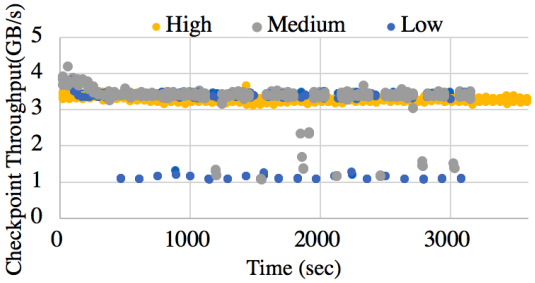

FIGURE 2. Checkpoint performance with different I/O job congestion rates. Time interval of each I/O job is equal and evenly distributed within the period under Low congestion rate. Time interval of each I/O job is halved and tenth of Low case under Med and High congestion rates.

Buffer is increased and Burst Buffer is filled up quickly with checkpoint data. As a result, demotion operations interrupt the checkpoint operations more aggressively.

Figure 1 shows checkpoint performance with different *DWPP*s after setting the demotion threshold to 90% of total Burst Buffer capacity. As S represents the total capacity of Burst Buffer, $1.3S$, $1.6S$, and $1.9S$ write 1.3 times, 1.6 times, and 1.9 times the size of Burst Buffer in a certain period, respectively. Each dot in the figure represents the application job that runs on Burst Buffer as time goes by. We only assume the over-subscription scenarios where the total size of checkpoint files is larger than Burst Buffer size ($DWPP > S$), which can improve the Burst Buffer utilization. When only small number of jobs are reserved to use Burst Buffer ($DWPP < S$), Burst Buffer utilization would remain low with each job having high I/O performance.

When $DWPP$ is $1.3S$, the performance of the jobs gets slightly decreased after 1000 seconds since the number of jobs run concurrently increases. Yet when $DWPP$ increases to $1.6S$, Burst Buffer is fully used in the middle of checkpoint I/O operations and the performance begins to drop over time. In the $1.9S$ case, almost half of the jobs get four times lower performance compared to the others as checkpoint operations have to be stopped and wait for demotion to make free space in Burst Buffer.

Another limitation of the previous works is that the arrival pattern of checkpoint operations is not considered in data management policy. For instance, the HPC jobs issue checkpoint operations with different periods. When the inter-arrival time is long enough, there exists a sufficient amount of Burst Buffer idle time between the checkpoint operations. Then the files can be demoted to PFS making space in Burst Buffer for the next I/O jobs. However, if the checkpoint operations are issued with small inter-arrival time, the lack of Burst Buffer idle time makes it difficult to finish data migration before the next checkpoint operation. This inevitably leads to Burst Buffer capacity depletion.

Figure 2 shows that checkpoint performance is highly related to the I/O job congestion rate under the same *DWPP*. Three I/O job congestion patterns, Low, Med, and High, represent the rate of how busy I/O jobs arrive and the jobs are allowed to use 1.9 times the size of Burst Buffer. The figure shows that the jobs under a Low congestion rate get high performance as there is a sufficient amount of idle time between the I/O operations. On the contrary, when the I/O jobs become to arrive in crowds in Med and High cases, some of the I/O jobs experience low checkpoint performance.

Naively using data eviction policy algorithms including FIFO, LRU, and LFU can leads to low Burst Buffer utilization. HPC applications have specific checkpoint periods and keep different number of checkpoint files to be used for data recovery. When the FIFO algorithm is used, the latest checkpoint file of application with long checkpoint period is considered cold data while old-version checkpoint file of application with short period is considered hot. As a result, the application with long checkpoint period experiences low recovery performance, which makes Burst Buffer inefficient. Also, the checkpoint files do not have data locality and spatial locality and LRU, LRU or other hotness-aware algorithm is not suitable for data eviction policy. To better classify which checkpoint files to be evicted, the failure rates need to be considered. Without taking the failure rates into account, checkpoint files with high failure rates might be chosen as cold data, instead of checkpoint files with low failure rates.

III. DATA MANAGEMENT IN BURST BUFFER

Applications may suffer from severe performance degradation when the characteristics of checkpoint and restart operations are not fully considered. In this paper, we set the demotion threshold on Burst Buffer and adjust the speed of checkpoint and demotion operations in advance to avoid the performance degradation. Also, we develop novel data placement policy that manages the data movement between Burst Buffer and PFS.

A. ADAPTIVE DEMOTION ADJUSTMENT

In order to make free space in Burst Buffer when using over-subscription method, we determine a demotion threshold considering both *DWPP* and I/O job congestion rate, which data can be retrieved from the log history of application jobs. As shown in Figure 1, *DWPP* affects the amount of data to be demoted in a certain period. Large *DWPP* means that there are large amounts of I/O to be written to Burst

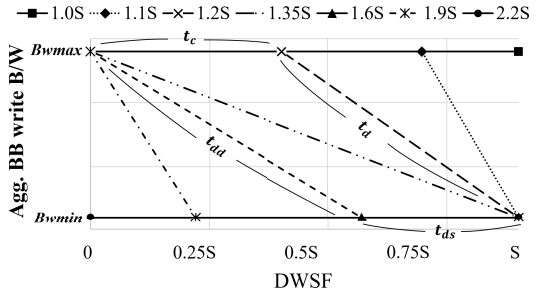

FIGURE 3. Optimal aggregated checkpoint bandwidth on burst buffer under different *DWPP*.

Buffer. In this case, the data needs to be demoted promptly to make free space in Burst Buffer. Burst Buffer may fill up quickly depending on the I/O job congestion rate as well as shown in Figure 2. The checkpoint and demotion throughput is another factor that needs to be considered when configuring the demotion threshold. When checkpoint and demotion operations are performed together, there exists an inverse relationship between write and read bandwidth within Burst Buffer I/O capability. As a result, the minimum demotion throughput (*Brmin*: minimum read throughput provided by Burst Buffer) is determined by the maximum checkpoint throughput (*Bwmax*: maximum write throughput provided by Burst Buffer). The minimum checkpoint throughput(*Bwmin*: minimum write throughput provided by Burst Buffer) is determined by the maximum demotion throughput(*Brmax*: maximum read throughput provided by Burst Buffer) as shown in equation (1). Note that read throughput provided by Burst Buffer is affected by write throughput provided by PFS. Also, m and b valued are decided depending on the device I/O capability. In order to avoid the worst case when Burst Buffer is running out of space, our policy adjusts the checkpoint throughput from *Bwmax* to *Bwmin*, and demotion throughput from *Brmin* to *Brmax* after used Burst Buffer space reaches demotion threshold.

$$BW_w = m \times BW_r + b \quad (m < 0) \tag{1}$$

Figure 3 shows the aggregated Burst Buffer write bandwidth under different *DWPP*. The overall goal is to sustain the maximum checkpoint bandwidth possible while handling the *DWPP* amount of data written to Burst Buffer within the checkpoint period. With S being the capacity of Burst Buffer, we refer to Data Written So Far (*DWSF*) as the amount of data written so far within the period. Within one period, the time given to execute checkpoint operation at *Bwmax* without any demotion is t_c, while t_d is the time required to demote C amounts of data with concurrent execution of checkpoint operations. t_d is composed of t_{dd} and t_{ds}: each representing the time taken for demotion throughput to gradually increase from *Brmin* to *Brmax*, and the time taken when the demotion throughput is fixed to *Brmax* without changing, respectively. We categorize the I/O patterns of the demotion operations into three categories to decide the demotion threshold.

1) PATTERN 1: DEMOTION IS ONLY PERFORMED WHEN BURST BUFFER IS IDLE

When *DWPP* is less than $1.0S$, the checkpoint can be executed with the bandwidth of *Bwmax* without the need for any demotion as shown in Figure 3. When the checkpoint period is finished, demotion can be performed in the Burst Buffer idle time.

2) PATTERN 2: DEMOTION IS PERFORMED TOGETHER WITH CHECKPOINT FOR SOME RANGE OF TIME

When *DWPP* exceeds $1.0S$, some of the data in Burst Buffer needs to be demoted concurrently with checkpoint operations. The time for demotion operations to be started is calculated depending on *DWPP*. For instance, when *DWPP* is $1.2S$, the demotion threshold is calculated as $0.7S$ of *DWSF*. In other words, demotion should be start even when the checkpoint is being executed when 70% of total Burst Buffer space is used. The checkpoint throughput is adjusted in a range between *Bwmax* and *Bwmin* for demotion to be executed and the demotion throughput is also changed in a range between *Brmin* and *Brmax* accordingly.

3) PATTERN 3: DEMOTION IS ALWAYS PERFORMED CONCURRENTLY WITH CHECKPOINT

When *DWPP* exceeds certain point, the demotion needs to be executed concurrently with checkpoint operations all the time. When *DWPP* exceeds $1.35S$ in Figure 3, the demotion has to begin at the very beginning of the checkpoint period. In this case, the demotion operations are executed with maximum throughput, *Brmax*, while checkpoint operations are executed with minimum throughput, *Bwmin*, when more than 60% of total Burst Buffer space is used.

Using the following equation (3) and *DWPP* value, the threshold capacity of Burst Buffer to start demotion and corresponding demotion bandwidth are calculated.

$$\begin{aligned} &t_c > 0, \\ &\quad \int_0^{t_c+t_{dd}} BW_w(t)\,dt \\ &= Bwmax \times t_c + \tfrac{Bwmax+Bwmin}{2} \times t_{dd} = DWPP \\ &\quad \int_0^{t_c+t_{dd}} BW_r(t)\,dt \\ &= Brmin \times t_c + \tfrac{Brmax+Brmin}{2} \times t_{dd} = C \end{aligned} \tag{2}$$

$$\begin{aligned} &t_c = 0, \\ &\quad \int_0^{t_{dd}+t_{ds}} BW_w(t)\,dt \\ &= \tfrac{Bwmax+Bwmin}{2} \times t_{dd} + Bwmin \times t_{ds} = DWPP \\ &\quad \int_0^{t_{dd}+t_{ds}} BW_r(t)\,dt \\ &= \tfrac{Brmax+Brmin}{2} \times t_{dd} + Brmax \times t_{ds} = C \end{aligned} \tag{3}$$

Since all the data on Burst Buffer needs to be demoted in order to avoid interference with checkpoint operations on the next checkpoint period, the minimum required idle

time between two checkpoint periods is calculated using the following equation (4).

$$(idle\_time - (t_c + t_d)) \times Brmax \geq S \tag{4}$$

B. DATA PLACEMENT POLICY

In this work, we develop a novel data placement policy that takes into account the characteristics of checkpoint and restart operations. The new policy keeps the latest checkpoint file on Burst Buffer as long as possible so that there is no need to prefetch data from PFS to Burst Buffer. Specifically, the hotness of the data is determined by considering the version of the checkpoint file and failure rate of HPC application. Old version checkpoint files have the highest priority to be considered as cold data since latest version checkpoint data is also located in Burst Buffer. If there are no old version checkpoint files left in Burst Buffer, we identify the coldness based on failure rates of the applications that write checkpoint files. As HPC users tend to reserve a large number of Burst Buffer nodes to avoid failure, we consider the failure rate of the applications proportional to the number of Burst Buffer nodes used.

C. DIRECT CHECKPOINT ON PFS

The I/O capability of PFS can also be exploited to further improve the Burst Buffer efficiency. Since cold data in Burst Buffer is destined to be in PFS, those data do not need to be written on Burst Buffer first. For this reason, we add Burst Buffer bypassing option on the data management policy once the data is considered to be cold compared to the other data already stored on Burst Buffer. This is possible because the failure rates of all the incoming checkpoint data can be known in advance from the log history. The checkpoint data is always checked whether it is hot or cold by comparing failure rates with the ones of other checkpoint files on Burst Buffer. If the incoming checkpoint data is determined to be cold, the checkpoint is directed to be written on PFS. This reduces the amount of demotion data to be written to Burst Buffer, which also diminishes the concurrent execution of checkpoint and demotion.

IV. DESIGN AND IMPLEMENTATION

We propose Burst Buffer Over-Subscription scheme (BBOS), a novel HPC data management approach that improves both Burst Buffer utilization and maintains high checkpoint and restart performance. Figure 4 shows the overall architecture of the BBOS framework. BBOS is composed of two engines, I/O engine and data management engine, and an in-memory key-value store that helps engine process.

A. I/O ENGINE

On the system with BBOS scheme, I/O operations of HPC application jobs are scheduled by BBOS I/O scheduler. Since the over-subscription method increases the number of the I/O jobs accessing Burst Buffer, an extreme I/O congestion can happen with low I/O performance. The severe

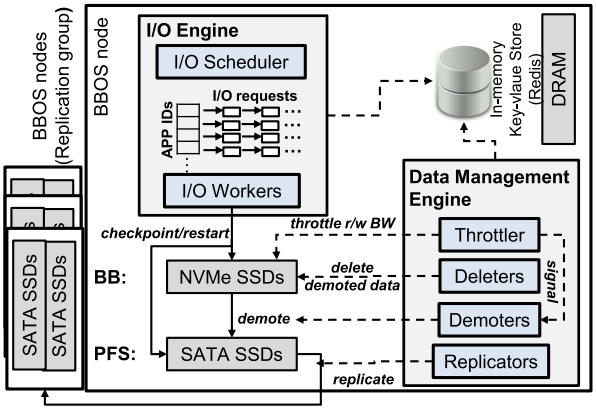

FIGURE 4. Overall architecture of the BBOS scheme.

resource competition and interference among multiple I/O jobs degrades the overall performance of the jobs [27], [28]. Thus, I/O jobs have to be scheduled in a way that they do not overlap each other as much as possible. BBOS I/O engine places multiple I/O queues for each Burst Buffer node and assigns an individual queue to each application. The I/O jobs with different applications access different I/O queues. Then the scheduler retrieves the I/O jobs from I/O queues in a round-robin manner so that the I/O jobs do not overlap each other. I/O engine also manages multiple I/O worker threads to execute I/O jobs. They determine which storage tier the scheduled I/O jobs should access, either Burst Buffer or PFS, with help of the in-memory key-value store.

B. DATA MANAGEMENT ENGINE

Data management engine consists of four modules: *Throttler*, *Demoters*, *Deleters* and *Replicators*. *Throttler* is responsible for dynamically adjusting the bandwidth of the checkpoint and the demotion operations. *Demoters* demote data from Burst Buffer to PFS considering the version of each checkpoint file and failure rate of each application. In the BBOS data management system, demoted data remains in Burst Buffer like a cache unless there is no space left for a new checkpoint file to be written. Whenever Burst Buffer space for new checkpoints is not sufficient, *Deleters* remove the demote-finished data that still exists in Burst Buffer. Finally, *Replicators* transfer checkpoint files from storage devices of local PFS nodes to ones of remote PFS nodes within the same replication group to enforce data consistency.

C. IN-MEMORY KEY-VALUE STORE

We utilize Redis [29], an open-source in-memory key-value store to help the processing I/O and data management engines in the BBOS framework. Since BBOS does not use page cache for checkpoint and restart, the in-memory store utilizes unused memory and facilitates the BBOS engine execution. BBOS stores the data path of the checkpoint files and corresponding metadata information required for data placement in the Redis in-memory database. Based on the stored data, *Demoters* demote the oldest checkpoint files first. If every

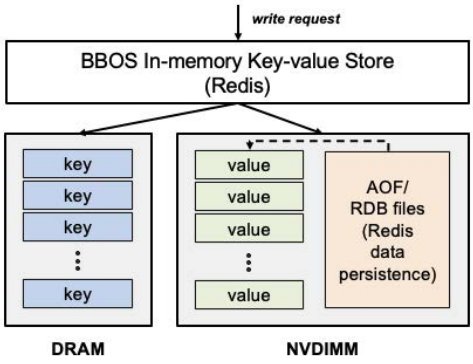

FIGURE 5. BBOS in-memory key-value store with persistent memory.

checkpoint files stored in Burst Buffer are the latest version data, *Demoters* start to demote the files starting from the one having the highest failure rate. *DWPP* and *DWSF* are also stored in the database to decide the demotion threshold and constantly tracked to adjust the checkpoint and demotion bandwidth. Nine key-value pairs used in BBOS in-memory key-value store are shown in Table 1. The detailed explanation of each pair is described in Section IV-F.

D. OPTIMIZED DESIGN FOR STABLE CHECKPOINT AND DEMOTION PERFORMANCE

To provide stable checkpoint, restart and demotion performance, the data management policy is optimized using several techniques. First, Checkpoint and demotion bandwidth are adjusted dynamically with BBOS engines. However, it is difficult to accurately control I/O bandwidth in real-world HPC systems. As the number of I/O operations per second requested from each application job varies, the checkpoint bandwidth may be different from what the data management policy expects to be. Also, the system may not be able to provide stable checkpoint performance due to the inability to demote as much data as it should. For these reasons, we use blkio [30] controller of the cgroup provided by Linux kernel to throttle the speed of checkpoint and restart operations precisely. Second, we utilize *send_file*() system call [31] to maintain stable demotion performance. In the demotion process, data must be read from Burst Buffer and written to PFS. This process incurs context switch and data copy overhead between user and kernel level, which leads to low and unstable demotion performance. Since *send_file*() system call supports zero-copy, demotion overhead can be eliminated. Lastly, checkpoint and restart performance may be degraded due to garbage collection occasionally. To avoid the garbage collection overhead, we periodically request the TRIM command after deleting the files. The TRIM throughput is also controlled by using blkio controller in order to minimize the performance degradation.

E. OPTIMIZED DESIGN USING PERSISTENT MEMORY

We further improve BBOS using persistent memory. A non-volatile dual in-line memory module (NVDIMM) is a new type of memory module that combines DRAM and storage in a DIMM socket. HPC supercomputer systems can get benefits from using NVDIMM as it provides memory-speed I/O performance at a lower cost. BBOS uses Redis in-memory key-value database to track the I/O-related metadata. Whenever the I/O accesses Burst Buffer, Redis updates multiple key-value pairs and uses those information to determine where to locate the checkpoint file or adjust read-write bandwidth. As a result, there are lots of accesses to the memory during I/O operations. In order to ease the memory overhead while improving data persistence, we take advantage of NVDIMM with Redis. We utilize pmem-redis [32], a Redis version that supports persistent memory to provide both high performance and persistence. Among several features that pmem-redis provides, we apply two features to the BBOS framework. The overall architecture of the BBOS in-memory key-value store using NVDIMM is shown in Figure 5.

First, considering NVDIMM as low-cost memory, we store key-value pairs on both DRAM and NVDIMM with a data placement strategy. Most of the HPC applications are memory-intensive workloads and they require large amounts of memory capacity when accessing Burst Buffer resources. When BBOS manages the Redis database, it would increase the memory usage as the data is served from memory. Consequently, Redis has to limit its memory consumption in order to not interfere with the I/O bandwidth of the HPC applications. To increase the memory capacity that Redis can utilize, pmem-redis provides a feature that can store large values in NVDIMM. This is because NVDIMM shows better performance on big and sequential data access pattern than the small and random data access pattern compared to DRAM performance. In this way, the DRAM usage on Redis can be saved while still providing DRAM-like access to large data. Specifically, all values with more than 64B size by default are stored in NVDIMM and the rest including keys and small values are stored in DRAM. Although NVDIMM shows higher latency compared to DRAM, our experiment shows that there is negligible overhead on pmem-redis when applications issue I/O operations on Burst Buffer.

Second, NVDIMM has a hard disk aspect in that data in persistent memory still exists after power failure and restart. The information stored in Redis is an important factor for the Burst Buffer scheduler to work properly. Also, whenever the applications request checkpoint or restart operations, the key-value pairs in Redis help locate the proper file while assuring other I/O requests to get reasonable I/O bandwidth. Default Redis offers two types of data persistence in order to keep data safe in the database. The RDB persistence writes all the data stored in memory to disk periodically, while AOF persistence logs every insert/modify/delete command issued to the servers to disk. These persistence methods have a major drawback in that data has to be written to slow hard disk. This is also a problem when there is a power outage and the data has to be read from a slow disk on the restart process. To improve Redis persistence performance, pmem-redis writes a persistence file in the reserved space of NVDIMM.

TABLE 1. Key-value pairs used in BBOS in-memory key-value store.

	KEY	VALUE	Description
1	FileName	String(path)	Record the file path for every file stored in BB and PFS
2	"VICTIM"	Sorted Set(MTBF, AppID)	Record victim App IDs and sort them by MTBF
3	"CLEAN"	List(AppID)	List up App ID which have demotion-finished files for cleanup when BB needs free space
4	"APP"	List(AppID)	Record list of App IDs having more than two different checkpoint versions stored in BB
5	"DWSF"	String(DWSF)	Record DWSF to throttle the bandwidth of checkpoint/restart operations
6	"REPLICA"	List(FileName)	Record list of files for replication
7	AppID+'"restart"	Sorted Set(MTBF,time)	Record the restart time and new MTBF for every read request
8	AppID+deviceID	Sorted Set(ver, FileName)	Record the version and the name of the checkpoint files of each HPC app for each device
9	AppID+"BB"	NULL	Record App ID if the checkpoint files of the app are in BB

When the persistence data size exceeds the reserved space, the data is evicted using LRU policy. As a result, periodic persistence can get improved write bandwidth and persistence files can be read with DRAM-like bandwidth whenever the power failure occurs.

F. IMPLEMENTATION

In this section, we describe the process flows of each engine in the BBOS framework using the Redis in-memory key-value store in detail. The BBOS framework is implemented by modifying the Gluster file system (GlusterFS), a highly scalable distributed file system. I/O engine and data management engine are added to the GlusterFS so that the engines can be processed interacting with the critical path of workflow. GlusterFS also interacts with the Redis server to collect the information used in the Burst Buffer scheduling policy.

There are total of nine kinds of key-value pairs stored in Redis in-memory key-value store as shown in Table 1. Redis records all the metadata information of the files written to and read from Burst Buffer. First, Redis stores file path for every file written in Burst Buffer and PFS so that the data management engine can have fast access to the files that need to be demoted or read. Every checkpoint files written by HPC applications have a specific application ID used by GlusterFS in the I/O flow. We refer to application-specific metadata as App ID. In order to identify victim checkpoint files to be demoted to PFS when there is not enough space on Burst Buffer, we manage Sorted Set with key name ''VICTIM''. The Sorted Set records App IDs in *MTBF* order, which represents the failure rate of each application. ''CLEAN'' key has a list of applications that have demotion-finished checkpoint files. In this case, the files stored in Burst Buffer can be erased. ''APP'' key manages a list of applications that have more than two different versions of checkpoint files stored in Burst Buffer. When there is not enough space in Burst Buffer, the old version checkpoint files have to be erased for those applications. Also, the ''DWSF'' key records the amounts of data written so far within the checkpoint I/O phase. The checkpoint and demotion bandwidth can be calculated using the current *DWSF* value. ''REPLICA'' key manages the list of files that needs to be replicated on the remote storage nodes. AppID+''restart'' key records the restart time and new *MTBF* calculated accordingly whenever the checkpoint file is read. AppID+deviceID key records the version and the name of the checkpoint files stored in each Burst Buffer device. This helps track the checkpoint files when a file demotion needs to be performed. Finally, AppID+''BB'' key is recorded so that BBOS engines can check whether the checkpoint files of the applications are previously written in Burst Buffer. Using the information stored in the Redis database, the I/O engine, and data management engine of the BBOS framework improves the Burst Buffer utilization while offering stable performance. The rest of this subsection presents implementation detail of the BBOS framework using the nine key-value pairs.

Algorithm 1 Pseudo-Code for Checkpoint

```
1: if freespace ! = enough
2:     Signal to DELETER
3: if get('AppID+"BB"', temp) ! = NULL
4:     put('"APP"', AppID)
5: put('AppID+deviceID', 'FileName')
6: put('FileName', 'path')
7: Execute checkpoint operation
8: put('"DWSF"', 'DWSF + current file size')
```

Algorithm 2 Pseudo-Code for Restart

```
1: get('AppID+"restart"', {prev_MTBF, timestamp})
2: new_MTBF = (prev_MTBF + (current Time - times-
   tamp)) / 2
3: update('AppID+"restart"', {new_MTBF, current Time})

4: if get('AppID+"BB"', temp) ! = NULL
5:     update('"VICTIM"', {new_MTBF, AppID})
6: get('FileName', 'path')
7: Execute restart operation
```

I/O engine schedules I/O jobs and finds appropriate storage tier for each checkpoint file. Algorithm 1 shows pseudo-code for process flow of checkpoint operation. Demoted data can stay in Burst Buffer unless there is not sufficient capacity for a new checkpoint file. Thus, the I/O engine first checks if there is enough space left before processing the checkpoint operation. If there is not enough space left, the engine sends *Deleters* a signal to delete demotion-finished files or outdated files (line 1-2). In addition, the engine checks if there are any outdated checkpoint files of the application on Burst Buffer. After the new checkpoint files are safely written, old versions of checkpoint files do not need to remain in Burst Buffer.

Algorithm 3 Pseudo-Code for Demotion

```
1:  pop('"APP"', AppID)
2:  if AppID != exists
3:     pop('"VICTIM"', {MTBF, AppID})
4:     flush ← TRUE
5:  pop('AppID+deviceID', {ver, FileName})
6:  for each file of files
7:     if(flush == TRUE)
8:        demote file from Burst Buffer to PFS
9:        put('"CLEAN"', AppID)
10:    else
11:       demote old version file from Burst Buffer to PFS
12:       delete file on Burst Buffer
13:    update('FileName', 'path')
14:    put('"REPLICA"', 'FileName')
```

In order to figure out the outdated files, the engine checks the Redis database if there is a key of the application in pair #9. If the key exists, the engine inserts the application to pair #4 so that *Deleters* can handle outdated files later on (line 3-4). The engine enlists checkpoint file names of each application and the device ID they are written to on pair #8 (line 5). Also, the engine saves the file path for each file name in pair #1 (line 6). After checkpoint operations are completed, the engine updates pair #5 with the current file size (line 7-8). The reason for continuously recording the *DWSF* value in the database is that the capacity of Burst Buffer will always remain full as our system keeps demotion-finished data in Burst Buffer until free space is actually necessary. BBOS would not know the actual amounts of data written if *DWSF* is not tracked during the checkpoint phases.

The process flow of restart operation is shown in Algorithm 2. When the system fails and the restart operation is requested, two new values are updated to let *Demoters* choose the victims. At first, the *MTBF* of the application that needs restart operation and the latest restart time is read from pair #7 (line 1). Then the module calculates new *MTBF* and updates pair #7 (line 2-3). At the same time, the engine checks whether the checkpoint file of the application exists on Burst Buffer or not by using pair #9. If the checkpoint file to be read is stored on Burst Buffer, pair #2 is updated with new *MTBF* (line 4-5). After all the process is done, the engine reads checkpoint files of the application with pair #1 (line 6-7).

While the I/O engine schedules the I/O jobs accessing Burst Buffer, the data management engine manages an efficient demotion process between Burst Buffer and PFS using the four modules. First, *Throttler* regulates the bandwidth of the checkpoint and the restart operations by monitoring *DWSF*. *Throttler* obtains *DWSF* from pair #5 and decides whether to start the demotion. When *DWSF* exceeds the demotion threshold, *Throttler* regulates the checkpoint and restart bandwidth to the reconfigured bandwidth. Second, *Demoters* receive a signal from *Throttler* about which device in Burst Buffer needs the demotion. Then, *Demoters* collect information from the in-memory store to execute the demotion. The pseudo-code for the demotion process is described in Algorithm 3. *Demoters* first check for every victim checkpoint file in pair #4 since the oldest version of the checkpoint file should be demoted first (line 1). If there is no victim, the victim file is retrieved from pair #2 which is ordered by *MTBF* (line 2-3). In this case, the victim file has to be demoted even though it is the latest checkpoint file of a certain application. If the victim is found from pair #2, the victim file is not deleted from Burst Buffer right after the demotion is finished. The file has to be stored in both Burst Buffer and PFS to preserve restart performance (line 7-8). However, it is necessary to mark that victim file in pair #3 in order to erase the file when Burst Buffer needs available capacity (line 9). If the victim is retrieved from pair #4, it also means that the application has an old version of the checkpoint file. Since the file of the old version does not need to stay in Burst Buffer, the file can be deleted (line 10-11). Finally, *Demoters* update pair #1 (line 12) and put the name of the file in pair #6 for *Replicators* to handle the replications (line 13). Third, *Deleters* erase demotion-finished files after receiving a signal from I/O workers. Specifically, *Deleters* pop information of the application first which is inserted in pair #3, and delete the files from Burst Buffer using pair #1 and #8. Lastly, *Replicators* replicate checkpoint files from the local storage device to the remote devices within the same replication group. Each storage node has a mount point of PFS which consists of storage nodes in the same replication group except itself. PFS-only low-speed network is additionally installed between each storage node. Thus, *Replicators* transfer the demoted data to the mount point by using pair #6 without hindering Burst Buffer performance.

V. EVALUATION

A. EXPERIMENTAL ENVIRONMENT

We evaluate the BBOS HPC storage management scheme on the small-scale testbed environment consists of eight compute nodes and a single storage node. Burst Buffer and PFS are configured together in the storage node. Four of the compute nodes consist of Intel Xeon Phi CPU 7290 processor with 72 physical cores and others are of Intel Xeon Phi CPU 7250 with 68 physical cores. The storage node consists of dual 12-core Intel Xeon Silver CPU 4115 and 32 GB memory. Burst Buffer is configured using four 800 GB FADU NVMe SSDs provided by a semiconductor start-up company [33], with the sequential write and read performance up to 920 MB/s and 3,200 MB/s. Also, 16GB Dell NVDIMM-N is deployed in the storage node in order to increase memory capacity and improve data persistence for the Redis in-memory database. PFS on the same storage node with Burst Buffer is composed of four 4TB Samsung 860 EVO SATA SSDs. The compute nodes and the storage node are connected with a 100 GbE Mellanox SN2100 switch.

We use GlusterFS [34] version 5.6 each configured for Burst Buffer and PFS and the file system configurations

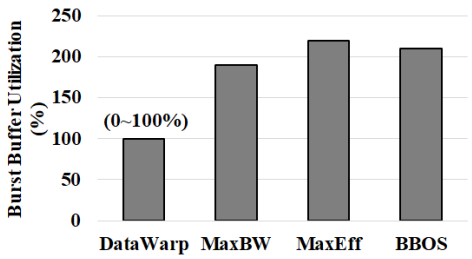

FIGURE 6. Burst buffer utilization.

are tuned for high performance. GlusterFS is modified by adding multiple modules for BBOS scheme. Each variable of the BBOS framework is configured as following by considering the capable I/O bandwidth provided by storage devices: *Bwmax* as 3.56 GB/s, *Bwmin* as 3 GB/s, *Brmax* as 1.6 GB/s, *Brmin* as 0.08 GB/s, and *period* as 3800 seconds. For experiments, we execute large sequential write I/O to simulate checkpoint operations by using a microbenchmark FIO [35]. Since failure rate and *MTBF* have an inverse relationship [36], *MTBF* is used to represent failure rates of the applications in this evaluation.

We compare BBOS with DataWarp, one of the currently deployed HPC schedulers which use the dedicated Burst Buffer allocation, and two scheduling policies presented in Harmonia [26] which is the state-of-the-art scheduler that uses the Burst Buffer over-subscription method. Since Harmonia is not an open-source work, we make an emulation scheduler based on the paper. DataWarp does not perform I/O scheduling, while Harmonia schedules I/O jobs for preventing them from overlapping each other. MaxEff, one of Harmonia's policies, optimizes the Burst Buffer system efficiency by maximizing the Burst Buffer utilization. As the policy aims to maintain the high capacity of free Burst Buffer space, it always demotes data at full speed (*Brmax*) even when the checkpoint is performed concurrently. On the other hand, MaxBW, another policy introduced in Harmonia, aims to provide maximum checkpoint bandwidth to applications. The checkpoint and the demotion cannot be performed at the same time with the MaxBW scheduling policy. The demotion threshold of MaxEFF is 0*S* of *DWSF* while threshold of MaxBW is 1*S* of *DWSF* in Figure 3.

B. BURST BUFFER UTILIZATION

In this section, we evaluate the Burst Buffer utilization with four scheduling policies. We assume that each application requests to write an 80 GB checkpoint file once a period. The Burst Buffer utilization is decided by the number of applications that finish writing the checkpoint file within the period, which also indicates the maximum *DWPP* each scheduler can provide. The Burst Buffer utilization of four scheduling metrics is shown in Figure 6. DataWarp shows 0∼100% of Burst Buffer utilization since it allocates Burst Buffer capacity as much as the users demand with a dedicated allocation method. The best scenario is that the total Burst Buffer capacity is fully used within the checkpoint period even when all users demand Burst Buffer allocation as much as they need. This results in 100% of Burst Buffer utilization. However, Burst Buffer utilization remains low due to overabundant Burst Buffer capacity requests in most cases. On the other hand, Harmonia and BBOS can make Burst Buffer accommodate more I/O requests within the period since they use an over-subscription Burst Buffer allocation method. MaxBW does not allow demotion to be performed together with the checkpoint to ensure maximum checkpoint throughput of the applications. As a result, 190% of Burst Buffer utilization can be achieved using the MaxBW scheduling policy. MaxEff shows 210% of Burst Buffer utilization because demotion is always performed at maximum demotion throughput taking the risk of low checkpoint performance. BBOS is similar to MaxEff in that demotion is performed at any time possible without interfered by checkpoint operations. Hence, Burst Buffer can be utilized by up to 210% with BBOS.

C. CHECKPOINT PERFORMANCE

To evaluate the checkpoint performance on BBOS framework, we conduct experiments under various I/O scenarios with different I/O job congestion rates and *DWPP*s. Since the maximum *DWPP* of DataWarp is equal to the total capacity of Burst Buffer, we evaluate DataWarp with *DWPP* at 1*S* while others with *DWPP* at 1.3*S*, 1.6*S*, and 1.9*S*. We make different I/O job congestion patterns on the following three scenarios:

- Low: Time interval of each I/O job is equal and evenly distributed within the period.
- Med: Time interval of each I/O job is halved of Low case.
- High: Time interval of each I/O job is tenth of Low case.

For instance, if I/O jobs are requested every 50 seconds under the Low I/O congestion rate, I/O jobs arrive every 25 seconds and every 5 seconds under the Med and High I/O congestion rates, respectively. All applications are assumed to request an 80 GB checkpoint file once per period for simplicity.

Figure 7 shows the checkpoint throughput and latency under different I/O scenarios. Checkpoint latency includes 1) the time to wait until the previous job is finished to prevent concurrent execution of I/O jobs (wait time), 2) the time to wait until free Burst Buffer space is reserved (stall time), and 3) the execution time of I/O job (execution time). When there is sufficient amount of idle time between the I/O jobs, data written during checkpoint period can be demoted during idle time with the DataWarp. As a result, DataWarp can provide high checkpoint throughput under the Low I/O job congestion rate. On the contrary, the checkpoint throughput remains low under the High I/O job congestion rate. This is because there is no time to make free space in Burst Buffer as I/O jobs arrive in crowd even when the previous jobs are not finished. As a result, DataWarp provides the lowest average checkpoint throughput and similar average latency even with *DWPP* at 1.0*S* compared to BBOS.

Different from DataWarp, Harmonia and BBOS schedule I/O jobs in a way that mitigate I/O interference across the jobs. Since MaxBW does not allow concurrent execution of

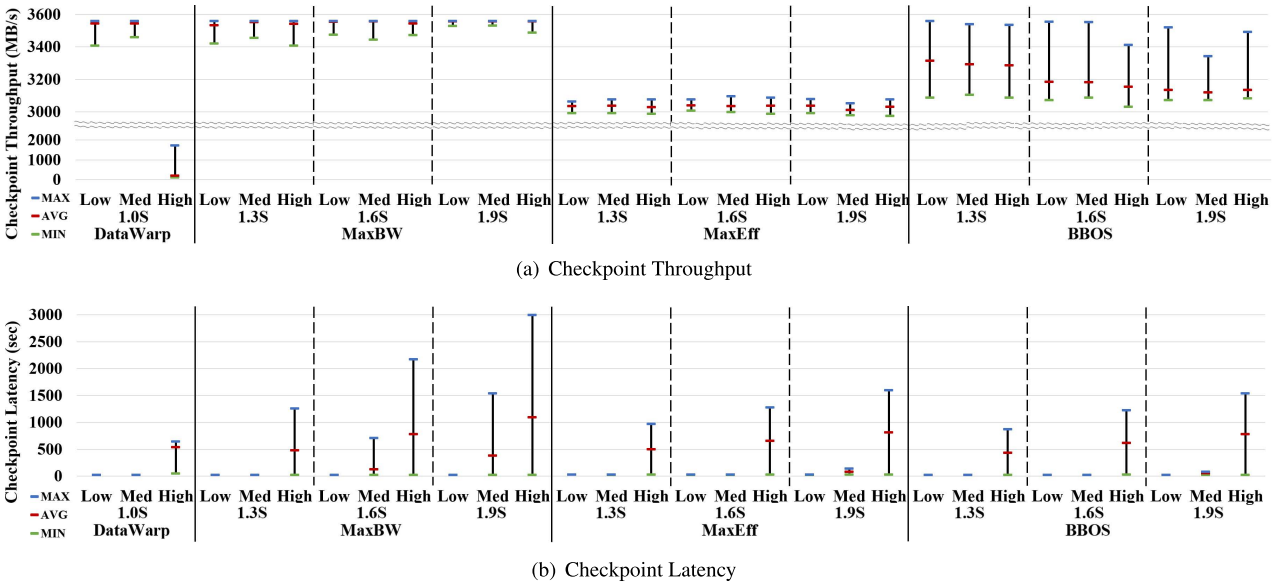

FIGURE 7. Checkpoint throughput and latency. Low, Med, High refer to I/O job congestion rates and *S* refers to *DWPP*.

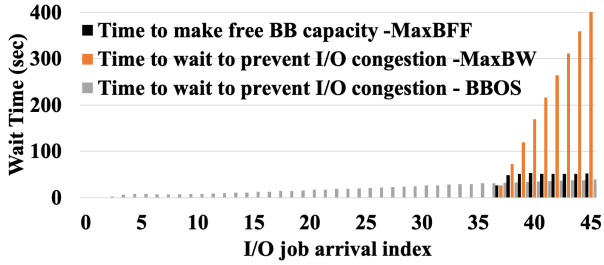

FIGURE 8. Wait time of I/O jobs with MaxBW policy.

demotion and checkpoint, high checkpoint throughput can be ensured all the time. However, some of the I/O jobs still have to stall in order to wait for available Burst Buffer capacity before the execution. Under the Low congestion rate scenario, none of the applications have to wait to avoid I/O interference or to make space in Burst Buffer as there is sufficient idle time between the I/O jobs. As the I/O jobs arrive in crowds and *DWPP* increases, some of the applications begin to experience high latency. Specifically, the checkpoint latency starts to increase with Harmonia and BBOS under Med I/O job congestion rate and *DWPP* over 1.6*S*. The large *DWPP* represents that there is not much idle time between the jobs and the checkpoint latency increases as *DWPP* increases. When the I/O job congestion rate is High, jobs have to wait for the longest time and results in the highest checkpoint latency.

In addition, MaxBW shows extreme performance variance. Figure 8 shows the wait time of the first 45 I/O jobs under the Med I/O congestion rate at *DWPP* of 1.9*S*. The later arrived I/O jobs have to wait for a long time, resulting in severe performance fluctuation. On the other hand, BBOS makes sure that data is demoted in advance so that Burst Buffer always reserve free space for the incoming jobs. The wait time under BBOS scheme gradually increasing from the beginning, preventing a sudden burst in the wait time in any case. In summary, MaxBW provides higher performance compared to BBOS when there is no wait time. When there is not enough Burst Buffer idle time per period or idle time between I/O jobs, MaxBW shows the higher latency and higher performance variance compared to BBOS. This is because BBOS always prepares for the worst case and adjust the checkpoint performance to reserve free space in Burst Buffer in advance.

Both MaxEff and BBOS perform demotion in advance for Burst Buffer not to overflow. MaxEff shows the lowest checkpoint throughput because the data is always demoted at the maximum demotion speed. In this way, relatively large amounts of Burst Buffer capacity can be maintained. Consequently, MaxEff provides lower checkpoint latency compared to MaxBW. BBOS adjusts checkpoint throughput within the range from *Bwmax* to *Bwmin* depending on *DWPP*. The smaller *DWPP*, the higher checkpoint throughput can be achieved by avoiding unnecessary concurrent execution of checkpoint and demotion. When there is enough time to make free Burst Buffer capacity, only the checkpoint throughput affects the latency. Hence, BBOS shows lower checkpoint latency compared to MaxEff when *DWPP* is small. MaxEff shows higher checkpoint latency compared to BBOS when *DWPP* is large, even though MaxEff performs demotion more aggressively than BBOS does. This is because MaxEff always demotes data at full demotion speed. In order to demote data in the maximum speed, the checkpoint throughput has to be decreased. As a result, checkpoint I/O jobs need to wait longer to be scheduled. In our experiments, the difference in latency of MaxEff and BBOS seems small (about tens of seconds) because the difference between *Bwmax* and *Bwmin* is not

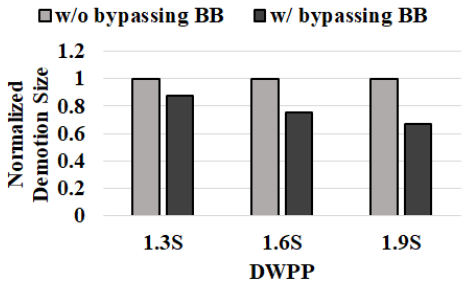

FIGURE 9. Direct checkpoint on PFS by bypassing burst buffer.

large. If the difference gets greater, we expect significantly lower I/O latency with BBOS compared to MaxEff.

Overall, BBOS is the novel approach that takes advantage of and complements the shortcomings of MaxBW and MAX-Eff. By adjusting checkpoint and demotion speed depending on *DWPP* and I/O job congestion rate dynamically, BBOS always provides relatively high checkpoint throughput and low latency compared to the other approaches. Furthermore, our result shows that there is no I/O overhead shown with managing BBOS framework on Burst Buffer. In other words, applications can get expected I/O performance as BBOS throttles the speed of checkpoint and restart operations depending on decisions made by BBOS I/O scheduler. As a result, BBOS can adjust the checkpoint throughput of the jobs in real-world HPC systems with thousands of storage nodes likewise to the checkpoint throughput under single Burst Buffer node system. Since BBOS runs with pre-determined system configurations setting, BBOS always provides stable checkpoint performance within a configured range under hardware limits.

D. DIRECT CHECKPOINT ON PFS

When the difference of the I/O bandwidth provided by PFS and Burst Buffer is not large, bypassing Burst Buffer and directly accessing PFS can eliminate unnecessary demotion overhead. We conduct experiments with three different *DWPP*: 1.3S, 1.6S, and 1.9S. Each application requests an 80 GB checkpoint during one hour and *MTBF* of all the applications are set randomly from 0 to 100 minutes. We optimize the BBOS framework by checking the *MTBF* of the applications that issue checkpoint requests. Before serving the request, the I/O engine first checks whether the Burst Buffer capacity is fully used. Only when the demotion is needed in order to make free space in Burst Buffer, the engine next checks whether the checkpoint file to be written is cold data or not by comparing the failure rates of the applications and the version number among the checkpoint files of the same application. The application with large *MTBF* is considered to write cold data since there is less possibility to get failure. Also, when there are multiple checkpoint files with different version numbers, only the latest file is considered to be hot. When the cold data is to be written when there is not enough Burst Buffer space, optimized BBOS bypass the Burst Buffer and write directly on PFS. Figure 9 shows the normalized demotion data size with optimized BBOS under different *DWPP* scenarios. In the case of *DWPP* being 1.9S, large amounts of checkpoint files that are considered to be cold data are written directly on PFS, which decreases the amounts of demoted data by up to 38%. Since less amount of data is demoted concurrently with checkpoint operations, more applications can experience higher checkpoint throughput.

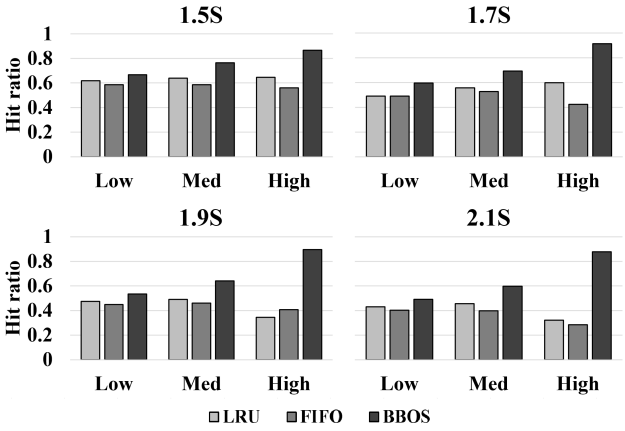

FIGURE 10. Hit ratio of restart requests on burst buffer. Low, Med, High refer to *MTBF* variance of applications.

E. RESTART PERFORMANCE

We evaluate restart performance on Burst Buffer by comparing the hit ratio using different scheduling policy: LRU, FIFO, and BBOS. *DWPP* is configured as 1.5S, 1.7S, 1.9S, and 2.1S, and we randomly set *MTBF* of the applications between the following ranges: 0 to 20 minutes (Low), 0 to 50 minutes (Med), and 0 to 100 minutes (High). The applications that need restart is selected based on the expected *MTBF*, as the failure rate is in the inverse relationship with *MTBF*. All the checkpoint periods are fixed to be equal and the checkpoint size of each application is set to 80 GB.

Figure 10 shows the hit ratio under different configurations. In every case, BBOS shows the highest hit ratio on Burst Buffer. Since the checkpoint files have a higher possibility to be in Burst Buffer with low *DWPP*, the hit ratio increases with low *DWPP* under all three scheduling policies. In the case of LRU and FIFO algorithms, however, cold data is chosen based on the order of data written time. As a result, the variance of the hit ratio of each experiment is high and the result is unrelated to the variance of *MTBF*. In contrast, BBOS shows an increased hit ratio as the *MTBF* variance gets higher. With low *MTBF* variance, the effectiveness of our system is relatively low compared to other policies since failure rates of the applications are similar. On the other hand, the checkpoint files are well distributed on Burst Buffer and PFS, sorted by the failure rates in case of the high variance of *MTBF*. As a result, BBOS provides up to 3.4 times higher hit ratio of restart requests on Burst Buffer compared to the others.

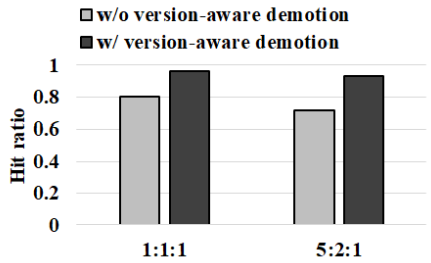

FIGURE 11. Version-aware data placement. 1:1:1 and 5:2:1 refer to the ratio of the number of the applications with 60, 30 and 20 minutes checkpoint period, respectively.

F. VERSION-AWARE DATA PLACEMENT

In order to keep the hit ratio high on Burst Buffer, BBOS uses the version-aware data placement method by identifying outdated checkpoint files as cold data. To demonstrate the effectiveness of the version-aware data placement method, we choose three checkpoint periods for applications as follows: 60 minutes, 30 minutes, and 20 minutes. Each user is to request an 80 GB-size checkpoint data and *MTBF*s of the applications are decided randomly from 0 to 100. We assume that the applications maintain three or more versions of checkpoint files. Thus, the applications with a 60-minute period have one checkpoint version within a one-hour period, two versions for a 30-minute period, and three versions for a 20-minute period. We also arrange the ratio of the number of applications having every three periods as 1:1:1 and 5:2:1 and the *DWPP* is fixed to be $1.9S$.

Figure 11 shows the hit ratio of restart requests on Burst Buffer with and without using the version-aware method. When the numbers of applications with different checkpoint periods are same, all the latest checkpoint files can be stored in Burst Buffer with the version-aware method and results in 96.4% hit ratio in the ideal case. Our result shows the slight decrease in hit ratio because the checkpoint file with the highest *MTBF* has to be demoted even though it is the latest one whenever the free Burst Buffer space is needed for incoming I/O requests. When the version-aware data placement is not used, cold data is decided only based on the *MTBF* of the applications and the latest checkpoint files with high *MTBF* may be stored on PFS while old version files with low *MTBF* stay on Burst Buffer. As a result, only 80.1% hit ratio is shown in our evaluation. In the case of the 5:2:1 ratio, there are large number of applications with low checkpoint period and every checkpoint files cannot all be stored in Burst Buffer. Thus, 92.5% of the restart requests can be handled in Burst Buffer with version-aware placement policy and only 71.7% of the requests can be handled without the policy.

G. PERFORMANCE OF BBOS USING NVDIMM

The BBOS framework is further improved by using NVDIMM on Redis in-memory database. We show two evaluation results in this section: the performance of NVDIMM-aware Redis database and the I/O performance of Burst Buffer with NVDIMM-applied BBOS.

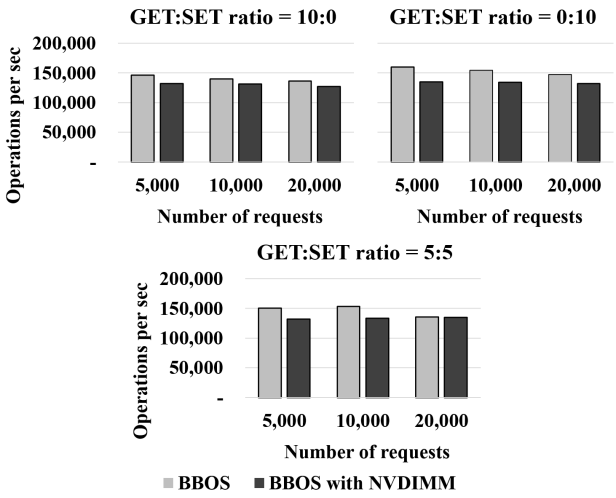

FIGURE 12. Performance overhead of NVDIMM-applied BBOS.

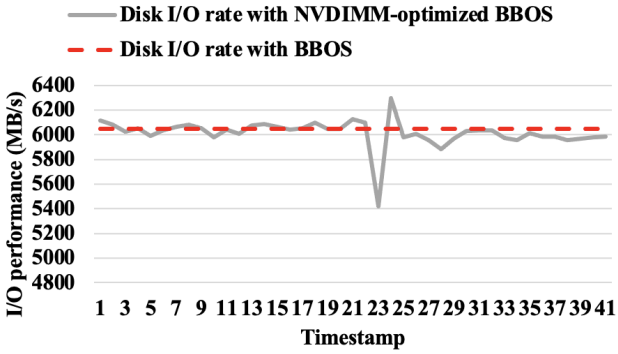

FIGURE 13. Aggregated disk usage rate over time in burst buffer with NVDIMM-applied BBOS. The red dashed line refers to the average I/O performance of burst buffer without using NVDIMM.

First, to focus on the performance of Redis with persistent memory, we use memtier benchmark [37] and compare the performance of default Redis and pmem-redis version Redis. We set the range of the requested data size to be between 64B and 256B, which is chosen empirically as the same size of data is read and written to Redis in the BBOS framework when Burst Buffer serves the checkpoint and restart operations. We set the ratio of GET operations to SET operations as 10:0, 0:10, and 5:5, while the number of requests is set to 5,000, 10,000, and 20,000. Figure 12 shows the transaction rates when using default Redis and Redis with NVDIMM, respectively. Every key is kept in DRAM while value larger than 64B is written to NVDIMM. As every value size is larger than 64B in our configuration, every value is kept in NVDIMM. Although DRAM space is saved using this approach, an additional data copy is required in order to move value data from DRAM to NVDIMM. Also, data written in NVDIMM requires longer latency to read or write compared to that of DRAM. As a result, pmem-redis version of Redis provides 0.4% to 15.7% lower performance compared to default Redis.

Second, we evaluate the I/O performance of Burst Buffer with NVDIMM-applied BBOS. Figure 13 shows the aggregated disk I/O rate over time on a single Burst Buffer node when BBOS runs using Redis with NVDIMM. The red dashed line in the graph refers to the average I/O performance of Burst Buffer without NVDIMM. We do not throttle the I/O performance in this evaluation. The result shows that even when Redis stores most of the data in NVDIMM, the disk I/O rate does not decrease under heavy I/O load. In other words, the optimized version of Redis using persistent memory rarely harms the bandwidth of the I/O workloads. To sum up, we consider the low performance of pmem-redis version of Redis is acceptable as the experimental result shows that the I/O bandwidth of Burst Buffer still reaches the maximum performance limit.

When bursty I/O comes in crowd in Burst Buffer system with BBOS framework, not only the data movement among multiple tiers but data replication and file system processes to manage I/O operations increase memory usage. Under the real-world HPC system environment that consists of thousands of compute nodes and storage nodes, the memory consumption would extremely increases and eventually harms the overall system performance. As such, it is advantageous to maintain a large amount of free memory as possible. Considering that exploiting NVDIMM can easily increase memory capacity with lower cost, our solution is capable of maintaining sufficient amount of memory space with no I/O performance degradation.

VI. RELATED WORK

Burst Buffer has been widely deployed in HPC storage systems in the past few years to improve the I/O performance. A large number of scientific HPC applications can benefit from high I/O performance provided by Burst Buffer [10], [38]. Depending on the architecture designs, Burst Buffer can be either located within the compute nodes or independently located as dedicated Burst Buffer nodes [11], [39], [40]. Common Burst Buffer design used in HPC systems is shared Burst Buffer organization, which shows higher I/O performance compared to local Burst Buffer design [41]. To further optimize the Burst Buffer system, numerous studies have been made in HPC communities with different approaches.

As Burst Buffer works as a cache layer in HPC storage systems, several studies have proposed the novel optimization techniques on Burst Buffer framework. Khetawat et al. [42] designed a simulation framework that can accurately find the best Burst Buffer configuration setting considering the I/O characteristics of real-world HPC workloads. Aupy et al. [43] minimized the I/O contention by sizing and partitioning Burst Buffer using polynomial time algorithms. After Burst Buffer is widely used as high-performance cache layer, researchers have also focused on improving checkpoint and restart performance using Burst Buffer. One of the approaches is to write checkpoint files on multiple layers including compute nodes, Burst Buffer, and PFS [44]. To reduce checkpoint overhead on PFS, Moody et al. [25] developed the multi-level checkpointing mechanism considering the different degree of reliability and the checkpoint cost of each tier in the HPC storage system. Data Elevator implemented by Dong et al. [45] offloads I/O access from Burst Buffer to PFS to reduce the contention on Burst Buffer. Different from Data Elevator that needs users to specify the final destination of the data, BBOS dynamically manages direct checkpointing on PFS when there is not enough free space in Burst Buffer. Since multi-level checkpointing can lead to high failure rates on a large-scale HPC environments, Sato et al. [46] combined the multi-level checkpointing and non-blocking mechanism so that data can be transferred asynchronously on checkpoint operations. Similar to the previous studies, our work also focuses on improving checkpoint and restart performance on Burst Buffer on multi-layered HPC systems.

Burst Buffer can be fully utilized with help of proper I/O scheduling policies, likewise to the policy that exists for PFS [27], [47], [48]. Han et al. [28] observed that the I/O capability of Burst Buffer cannot be fully used when multiple HPC users simultaneously use Burst Buffer. To address the problem, they proposed Burst Buffer with multi-stream SSDs by assigning each user a separate I/O stream to remove the I/O interference. Koo et al. [49] further improved the I/O separation scheme on Burst Buffer by proposing stream-aware scheduling policy on Burst Buffer I/O pools. Thapaliya et al. [47] also reported the I/O interference problem in shared Burst Buffer system and Gainaru et al. [50] attempted to dynamically schedule the I/O jobs based on the past I/O patterns of the jobs. TRIO is the Burst Buffer I/O scheduling policy that efficiently transfer the I/O traffic from Burst Buffer to PFS [2]. Similar to the above works, BBOS introduces novel I/O scheduling policy that efficiently handle data between Burst Buffer and PFS considering the checkpoint characteristics.

Several works claimed that HPC applications have frequently accessed data including checkpoint files [3], [51]. By placing hot data on Burst Buffer, the I/O intensive applications can get benefit from using Burst Buffer. Shin et al. [15] automatically placed data on HPC multi-tiered storage system using goal-driven data management scheme, while Shi et al. [52] regulated I/O traffic on Burst Buffer and PFS by using the write access patterns of the applications. Our work also considers I/O patterns and characteristics of checkpoint operations when making data placement decision.

VII. CONCLUSION

BBOS, the new I/O scheduling framework for Burst Buffer-based HPC storage system, uses the over-subscription scheduling method by allocating Burst Buffer only during I/O phases to improve Burst Buffer utilization. In order to mitigate performance degradation, the Burst Buffer aware I/O scheduler and the data management module are implemented in BBOS. We analyzed and utilized the characteristics of checkpoint and restart operations to design the BBOS modules. Based on the characteristics, data is transferred from Burst Buffer to PFS transparently by dynamically adjusting

the thresholds and the speed of the demotion. We also identified the cold data considering different versions and failure rates of the checkpoint files. All the metadata related to the BBOS framework is handled in the Redis in-memory database, which is improved by using persistent memory. As a result, we improved Burst Buffer utilization by up to 120% compared to the default dedicated Burst Buffer allocation method and guaranteed higher checkpoint throughput without sudden performance reduction. Also, 96.4% of restart requests can be handled in Burst Buffer and provided up to 3.1 times higher restart performance with BBOS framework.

REFERENCES

[1] K. Sato, K. Mohror, A. Moody, T. Gamblin, B. R. D. Supinski, N. Maruyama, and S. Matsuoka, "A user-level infiniband-based file system and checkpoint strategy for burst buffers," in *Proc. 14th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput.*, May 2014, pp. 21–30.

[2] T. Wang, S. Oral, M. Pritchard, B. Wang, and W. Yu, "TRIO: Burst buffer based I/O orchestration," in *Proc. IEEE Int. Conf. Cluster Comput.*, Sep. 2015, pp. 194–203.

[3] T. Xu, K. Sato, and S. Matsuoka, "Explorations of data swapping on burst buffer," in *Proc. IEEE 24th Int. Conf. Parallel Distrib. Syst. (ICPADS)*, Dec. 2018, pp. 517–526.

[4] *The ASC Sequoia Draft Statement of Work*. Accessed: Mar. 8, 2022. [Online]. Available: https://asc.llnl.gov/

[5] 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 storage systems," in *Proc. 28th IEEE 28th Symp. Mass Storage Syst. Technol. (MSST)*, Apr. 2012, pp. 1–11.

[6] O. Yildiz, A. C. Zhou, and S. Ibrahim, "Eley: On the effectiveness of burst buffers for big data processing in HPC systems," in *Proc. IEEE Int. Conf. Cluster Comput. (CLUSTER)*, Sep. 2017, pp. 87–91.

[7] B. R. Landsteiner, D. Henseler, D. Petesch, and N. J. Wright, "Architecture and design of cray datawarp," Cray User Group (CUG), 2016.

[8] X. Meng, C. Wu, J. Li, X. Liang, Y. Bin, M. Guo, and L. Zheng, "HFA: A hint frequency-based approach to enhance the I/O performance of multi-level cache storage systems," in *Proc. 20th IEEE Int. Conf. Parallel Distrib. Syst. (ICPADS)*, Dec. 2014, pp. 376–383.

[9] H. Sung, J. Bang, C. Kim, H.-S. Kim, A. Sim, G. K. Lockwood, and H. Eom, "BBOS: Efficient HPC storage management via burst buffer oversubscription," in *Proc. 20th IEEE/ACM Int. Symp. Cluster, Cloud Internet Comput. (CCGRID)*, May 2020, pp. 142–151.

[10] W. Bhimji et al., "Accelerating science with the NERSC Burst buffer early user program," in *Proc. Cray User Group Meeting*, 2016.

[11] T. Wang, K. Mohror, A. Moody, K. Sato, and W. Yu, "An ephemeral burst-buffer file system for scientific applications," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, Nov. 2016, pp. 807–818.

[12] *Cori Burst Buffer*. Accessed: Jun. 1, 2022. [Online]. Available: https://www.nersc.gov/users/computational-systems/cori/burst-buffer/

[13] *DATAWARP*. Accessed: Jun. 1, 2022. [Online]. Available: https://www.cray.com/products/storage/datawarp

[14] *Slurm Workload Manager*. Accessed: Jun. 1, 2022. [Online]. Available: https://slurm.schedmd.com/publications.html

[15] W. Shin, C. D. Brumgard, B. Xie, S. S. Vazhkudai, D. Ghoshal, S. Oral, and L. Ramakrishnan, "Data jockey: Automatic data management for HPC multi-tiered storage systems," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, May 2019, pp. 511–522.

[16] R. A. Ashraf, S. Hukerikar, and C. Engelmann, "Shrink or substitute: Handling process failures in HPC systems using in-situ recovery," in *Proc. 26th Euromicro Int. Conf. Parallel, Distrib. Netw.-Based Process.*, 2018, pp. 178–185.

[17] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Commun. ACM*, vol. 17, no. 9, pp. 530–531, Sep. 1974.

[18] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Gener. Comput. Syst.*, vol. 22, no. 3, pp. 303–312, Feb. 2006.

[19] G. Lu, Z. Zheng, and A. A. Chien, "When is multi-version checkpointing needed?" in *Proc. 3rd Workshop Fault-Tolerance HPC Extreme Scale*, 2013, pp. 49–56.

[20] S. Gupta, D. Tiwari, C. Jantzi, J. Rogers, and D. Maxwell, "Understanding and exploiting spatial properties of system failures on extreme-scale HPC systems," in *Proc. 45th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, Jun. 2015, pp. 37–44.

[21] N. Naksinehaboon, Y. Liu, C. Leangsuksun, R. Nassar, M. Paun, and S. L. Scott, "Reliability-aware approach: An incremental checkpoint/restart model in HPC environments," in *Proc. 8th IEEE Int. Symp. Cluster Comput. Grid (CCGRID)*, May 2008, pp. 783–788.

[22] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira, "Adaptive incremental checkpointing for massively parallel systems," in *Proc. 18th Annu. Int. Conf. Supercomput.*, 2004, pp. 277–286.

[23] H. Jin, Y. Chen, H. Zhu, and X.-H. Sun, "Optimizing HPC fault-tolerant environment: An analytical approach," in *Proc. 39th Int. Conf. Parallel Process.*, Sep. 2010, pp. 525–534.

[24] F. Petrini, K. Davis, and J. C. Sancho, "System-level fault-tolerance in large-scale parallel machines with buffered coscheduling," in *Proc. 18th Int. Parallel Distrib. Process. Symp.*, 2004, p. 209.

[25] A. Moody, G. Bronevetsky, K. Mohror, and B. R. D. Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *Proc. ACM/IEEE Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, Nov. 2010, pp. 1–11.

[26] A. Kougkas, H. Devarajan, X.-H. Sun, and J. Lofstead, "Harmonia: An interference-aware dynamic I/O scheduler for shared non-volatile burst buffers," in *Proc. IEEE Int. Conf. Cluster Comput. (CLUSTER)*, Sep. 2018, pp. 290–301.

[27] M. Dorier, G. Antoniu, R. Ross, D. Kimpe, and S. Ibrahim, "CALCioM: Mitigating I/O interference in HPC systems through cross-application coordination," in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp.*, May 2014, pp. 155–164.

[28] 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 *Proc. IEEE Int. Conf. Cluster Comput. (CLUSTER)*, Sep. 2017, pp. 245–255.

[29] *Redis in-Memory Database*. Accessed: Mar. 8, 2022. [Online]. Available: https://redis.io/

[30] *Block IO Controller*. Accessed: Mar. 8, 2022. [Online]. Available: https://www.kernel.org/doc/Documentation/cgroup-v1/blkio-controller.txt

[31] *Linux Sendfile*. Accessed: Mar. 8, 2022. [Online]. Available: http://man7.org/linux/man-pages/man2/sendfile.2.html

[32] *Pmem-Redis*. Accessed: Mar. 8, 2022. [Online]. Available: https://github.com/pmem/pmem-redis

[33] *FADU Technology*. Accessed: Mar. 8, 2022. [Online]. Available: http://www.fadu.io

[34] *Gluster File System*. Accessed: Mar. 8, 2022. [Online]. Available: https://www.gluster.org/

[35] *Flexible I/O Benchmark*. Accessed: Mar. 8, 2022. [Online]. Available: https://linux.die.net/man/1/fio

[36] D. Tiwari, S. Gupta, and S. S. Vazhkudai, "Lazy checkpointing: Exploiting temporal locality in failures to mitigate checkpointing overheads on extreme-scale systems," in *Proc. 44th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, Jun. 2014.

[37] *Memtier Benchmark*. Accessed: Mar. 8, 2022. [Online]. Available: https://github.com/RedisLabs/memtier_benchmark

[38] J. Lofstead, I. Jimenez, C. Maltzahn, Q. Koziol, J. Bent, and E. Barton, "DAOS and friends: A proposal for an exascale storage system," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, Nov. 2016, pp. 585–596.

[39] M.-A. Vef, N. Moti, T. Süß, T. Tocci, R. Nou, A. Miranda, T. Cortes, and A. Brinkmann, "GekkoFS—A temporary distributed file system for HPC applications," in *Proc. IEEE Int. Conf. Cluster Comput. (CLUSTER)*, Sep. 2018, pp. 319–324.

[40] T. Wang, W. Yu, K. Sato, A. T. Moody, and K. Mohror, "BurstFS: A distributed burst buffer file system for scientific applications," Lawrence Livermore Nat. Lab.(LLNL), Livermore, CA, USA, Tech. Rep., 2016.

[41] L. Cao, B. W. Settlemyer, and J. Bent, "To share or not to share: Comparing burst buffer architectures," in *Proc. 25th High Perform. Comput. Symp. (HPC)*. San Diego, CA, USA: Society for Computer Simulation International, 2017, pp. 1–10.

[42] H. Khetawat, C. Zimmer, F. Mueller, S. Atchley, S. S. Vazhkudai, and M. Mubarak, "Evaluating burst buffer placement in HPC systems," in *Proc. IEEE Int. Conf. Cluster Comput. (CLUSTER)*, Sep. 2019, pp. 1–11.

[43] G. Aupy, O. Beaumont, and L. Eyraud-Dubois, "Sizing and partitioning strategies for burst-buffers to reduce IO contention," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2019, pp. 631–640.

[44] A. Kougkas, H. Devarajan, and X.-H. Sun, "Hermes: A heterogeneous-aware multi-tiered distributed I/O buffering system," in *Proc. 27th Int. Symp. High-Perform. Parallel Distrib. Comput. (HPDC)*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 219–230.
[45] B. Dong, S. Byna, K. Wu, Prabhat, H. Johansen, J. N. Johnson, and N. Keen, "Data elevator: Low-contention data movement in hierarchical storage system," in *Proc. IEEE 23rd Int. Conf. High Perform. Comput. (HiPC)*, Dec. 2016, pp. 152–161.
[46] K. Sato, N. Maruyama, K. Mohror, A. Moody, T. Gamblin, B. R. de Supinski, and S. Matsuoka, "Design and modeling of a non-blocking checkpointing system," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2012, pp. 1–10.
[47] S. Thapaliya, P. Bangalore, J. Lofstead, K. Mohror, and A. Moody, "Managing I/O interference in a shared burst buffer system," in *Proc. 45th Int. Conf. Parallel Process. (ICPP)*, Aug. 2016, pp. 416–425.
[48] S. Herbein, D. H. Ahn, D. Lipari, T. R. W. Scogland, M. Stearman, M. Grondona, J. Garlick, B. Springmeyer, and M. Taufer, "Scalable I/O-aware job scheduling for burst buffer enabled HPC clusters," in *Proc. 25th ACM Int. Symp. High-Perform. Parallel Distrib. Comput.*, May 2016, pp. 69–80, doi: 10.1145/2907294.2907316.
[49] D. Koo, J. Lee, J. Liu, E.-K. Byun, J.-H. Kwak, G. K. Lockwood, S. Hwang, K. Antypas, K. Wu, and H. Eom, "An empirical study of I/O separation for burst buffers in HPC systems," *J. Parallel Distrib. Comput.*, vol. 148, pp. 96–108, Feb. 2021.
[50] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Y. Robert, and M. Snir, "Scheduling the I/O of HPC applications under congestion," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, May 2015, pp. 1013–1022.
[51] G. Zhang, L. Chiu, C. Dickey, L. Liu, P. Muench, and S. Seshadri, "Automated lookahead data migration in SSD-enabled multi-tiered storage systems," in *Proc. IEEE 26th Symp. Mass Storage Syst. Technol. (MSST)*, May 2010, pp. 1–6.
[52] X. Shi, M. Li, W. Liu, H. Jin, C. Yu, and Y. Chen, "SSDUP: A traffic-aware SSD burst buffer for HPC systems," in *Proc. Int. Conf. Supercomput.*, 2017, pp. 1–10.

JIWOO BANG received the B.S. degree from the Department of Computer Science and Engineering, Seoul National University, in 2017, where she is currently pursuing the integrated Ph.D. degree with the Department of Computer Science and Engineering. Her research interests include distributed file systems, parallel computing, and high-performance computing.

ALEXANDER SIM (Senior Member, IEEE) is currently a Senior Computing Engineer at the Lawrence Berkeley National Laboratory. He has authored or coauthored over 350 technical publications and released a few software packages under open source license. His current research and development interests include data modeling, data analysis methods, learning models, distributed data infrastructure, dynamic resource management, and high performance data systems.

GLENN K. LOCKWOOD received the B.S. degree in ceramic engineering and the Ph.D. degree in materials science from Rutgers University. He is currently the Product Manager of Microsoft, where he develops HPC storage strategy for Microsoft Azure. His background is in system architecture and I/O performance analysis. Previously, he held roles in storage research and development and system operations at the Lawrence Berkeley National Laboratory, National Energy Research Scientific Computing Center. He has a formal research background in silica surface chemistry.

HYEONSANG EOM received the B.S. degree in computer science and statistics from Seoul National University (SNU), Seoul, South Korea, in 1992, and the M.S. and Ph.D. degrees in computer science from the University of Maryland at College Park, MD, USA, in 1996 and 2003, respectively. He was an Intern with the Data Engineering Group, Sun Microsystems, CA, USA, in 1997, and a Senior Engineer with the Telecommunication R&D Center, Samsung Electronics, South Korea, from 2003 to 2004. He is currently a Full Professor with the Department of Computer Science and Engineering, SNU, where he has been a Faculty Member, since 2005. His research interests include distributed systems, cloud computing, operating systems, high performance storage systems, energy efficient systems, fault-tolerant systems, security, and information dynamics.

HANUL SUNG received the B.S. degree in computer science from Sangmyung University (SMU), Seoul, South Korea, in 2012, and the M.S. and Ph.D. degrees in computer science and engineering from Seoul National University (SNU), Seoul, in 2020. She is currently an Assistant Professor with the Department of Game Design and Development, SMU. Her research interests include distributed systems, operating systems, high performance storage systems, and cloud computing.