BBOS: Efficient HPC Storage Management via Burst Buffer Over-Subscription

Hanul Sung⋆, Jiwoo Bang⋆, Chungyong Kim⋆, Hyung-Sin Kim⋄, Alexander Sim†,
Glenn K. Lockwood†, and Hyeonsang Eom⋆
⋆*Department of Computer Science and Engineering, Seoul National University*
†*Lawrence Berkeley National Laboratory*
⋄*Data Science, Seoul National University*

***Abstract*—To avoid access to PFS, dedicated BB allocation is preferred despite of severe BB underutilization. Recently, new all-flash HPC storage systems with integrated BB and PFS are proposed, which speed up access to PFS. For this reason, we adopt BB over-subscription allocation method by allowing HPC applications to use BB only for I/O phase for improving BB utilization. Unfortunately, BB over-subscription aggravates I/O interference and demotion overhead from BB to PFS, resulting in degraded performance. To minimize the performance degradation, we develop an I/O scheduler to prevent I/O congestion and a new transparent data management system based on checkpoint/restart characteristics of HPC applications. With the proposed approach, not only the BB utilization can be improved, but also high performance of applications is achieved. In our experiments, we find that BB utilization is improved at least 2.2x, and more stable and higher checkpoint performance is guaranteed compared to other approaches. Besides, we achieve up to 96.4% hit ratio of restart requests on BB and up to 3.1x higher restart performance than others.**

***Index Terms*—Burst Buffer, PFS, Over-subscription, Checkpoint, Restart, Demotion**

I. Introduction

Parallel File System (PFS), comprised of many HDDs, has been a foundation of storage tier of High-Performance Computing (HPC) systems. As computational capability has grown over one petaflop, a large number of system components have been used in HPC systems, thereby causing increased overall system failures [1]–[3]. For a fail-safety purpose, HPC applications aggressively utilize checkpoint/restart, the most common fault tolerance mechanism. This causes the checkpoint to dominate 75%∼80% of I/O traffic of HPC system [2], [4] and significantly generates bursty I/O, which is difficult for PFS to handle. To alleviate the issue, Burst Buffer (BB) which is composed of high-end flash SSDs (e.g., 3D XPoint SSD and NVMe SSD) [5], [6] and a high-speed network has been introduced as a new storage tier between compute nodes and PFS. Because of the substantial performance differences offered by BB and PFS, HPC users prefer dedicated BB allocation for a whole lifetime of their HPC applications to avoid access to PFS as much as possible. But, this allocation style causes severe underutilization of expensive BB for two reasons. First, some users eagerly request BB resources (e.g., up to six times [7]) for I/O errors prevention, performance scalability, and complicated data movement between BB and PFS, even when they actually utilize only little portion (e.g., 5% of BB per hour is used according to the logs from NERSC Cori). Second, since HPC applications use BB only for I/O phases, BB stays idle for the rest of the time except for the I/O phases. Checkpoint that dominates I/O traffic of HPC system is requested scarcely such as once per hour or some minutes and thereby resources of BB are wasted most of the time.

Recently, there have been many efforts into merging BB with PFS [8], [9]. As the cost-per-bit of flash is falling, it becomes possible to replace HDDs with low-end TLC SSDs [10]. NERSC announced that the next supercomputer with an all-flash storage system, called PERLMUTTER, will be delivered in 2020. Therefore, it is expected that a new all-flash HPC storage system with integrated BB and PFS will be widely adopted in the near future. We believe that the new system has high-end SSDs (e.g., 3D XPoint SSD and NVMe SSD) for BB (performance) and low-end SSDs (TLC SATA SSD) for PFS (capacity) placed on the same node. PFS in the new system consists of a high-speed network and low-end SSDs, which significantly reduces the overhead of PFS access. So it is not worthwhile to use dedicated BB allocation method with penalty of BB underutilization. For this reason, instead of the dedicated BB allocation method, we adopt BB over-subscription allocation method for improving BB utilization. The method allocates BB capacity only to the I/O phase, not for the entire lifetime of HPC applications, resulting in reduced waste of BB resources. But, this method may affects the performance of checkpoint/restart, since a larger BB capacity is allocated to HPC applications than the total capacity of BB. Data management approach between BB and PFS is required to handle this problem, but studies for this have received relatively less attention until now.

In this paper, we propose an efficient HPC storage management approach, called BBOS, with a BB over-subscription allocation method. To support the BB over-subscription in the new HPC storage systems, we transparently manage data movement between BB and PFS and schedule I/O jobs, resulting in high BB utilization and high checkpoint/restart performance. The key idea behind BBOS is to utilize the characteristics of checkpoint/restart that occupy most I/O traffic in HPC storage systems. Because checkpoint/restart has specific characteristics, existing primitive data management approach

Identify applicable funding agency here. If none, delete this.

(e.g., a kernel data management approach between memory layer and storage layer and commonly used approaches for tiers in PFS) results in low checkpoint/restart performance. For this reason, we analyze characteristics of checkpoint/restart and decide data placement between BB and PFS based on the characteristics. To illustrate the effectiveness of BBOS, we evaluate our approach compared to Datawarp, a representative of the current HPC schedulers which uses the dedicated BB allocation method, and Harmonia [11], which is the only approach in consideration of BB over-subscription. Compared to Datawarp, we improve BB utilization considerably while providing high checkpoint performance. Besides, we provide high checkpoint performance and perform up to 96.4% of restart on BB by utilizing the characteristics of checkpoint/restart.

Our contributions are as follows:

- We adopt the over-subscription BB allocation method to handle BB underutilization problem caused by the dedicated BB allocation method.
- We analyze the checkpoint/restart characteristics of HPC applications. We observe that each application has its own checkpoint period and failure rate. In addition, there is no data locality across checkpoint files unlike normal data. We find the characteristics of HPC applications is highly related to low checkpoint/restart performance with existing data management approach.
- We propose BBOS, a novel HPC management approach based on the characteristics of checkpoint/restart to provide high BB utilization as well as high checkpoint/restart performance. BBOS schedules I/O jobs, adjusts demotion threshold and speeds of checkpoint and demotion adaptively and manages data placement between BB and PFS.
- We have implemented a BBOS prototype by adding some modules and modifying GlusterFS, one of the most popular distributed file system. BBOS shows increased BB utilization and improved checkpoint/restart performance than prior works.

II. Background and Motivation

A. *Burst Buffer underutilization*

Burst buffer is introduced for absorbing bursty I/O in HPC systems. Most of the supercomputers, including Cori [12] from NERSC, allocate BB by using a dedicated BB allocation method. The users specify the desired capacity for the applications and the specified space is provided by a HPC scheduler [13], [14] for the whole lifetime of the applications. But, this allocation method causes severe underutilization of BB which is composed of expensive hardware resources, such as high-speed storage media and high-speed network. Generally, the users request more than the actual necessary capacity because the application jobs fail due to I/O error when the allocated capacity is not enough. To avoid failure, the users are recommended by a supercomputer provider to request surplus BB capacity [7]. Not only for the failure, but the users may also require a bountiful capacity for higher performance as well. Since the scheduler decides the number of dedicated BB nodes in proportion to the requested capacity, the users request larger capacity so that they can experience higher performance with more BB nodes and higher parallelism. Along with the performance scalability, another reason for overabundant requests arises from complicated data management in multi-tier HPC storage systems (i.e., local storage of a compute node, BB and PFS). Since current supercomputers manage BB and PFS separately, the users are challenged with redundant and complicated management. For example, if the users have a limited BB capacity for only one checkpoint, they should copy data manually from BB to PFS every end of the I/O phase to make BB space for the next I/O phase. Furthermore, if a workflow of the application is complicated, manual data movement can be a difficult job for the users [15].

BB underutilization is also caused by the characteristics of the checkpoint/restart. HPC applications perform checkpoint with a fixed period [16]–[18], called checkpoint period, by repeating compute phase and I/O phase periodically. Unfortunately, as the checkpoint period ranges from tens of minutes to tens of hours, expensive BB resources keep idle for long compute phases. Moreover, each application requires BB capacity larger than actual checkpoint size in order to preserve the consistency of checkpoints. BB capacity is needed for at least twice as much the capacity for the checkpoints as old checkpoint should be kept until a new checkpoint is all written safely. HPC users also store multiple versions of checkpoint in BB for data durability of checkpoint. Since only the latest version of checkpoint is needed in case of a failure, the rest of the old versions do not actually need to be stored in BB.

These problems caused by the dedicated BB allocation method motivate our HPC storage management approach based on over-subscribing BB.

B. *Checkpoint/Restart Characteristics*

HPC applications have checkpoint/restart related characteristics unlike other applications. For the new HPC storage system with BB over-subscription method, a novel data management needs to be developed based on the characteristics.

First of all, HPC applications run for a long time to solve computationally intensive problems and perform checkpoint at a particular cycle to avoid re-computations from scratch. For this reason, the total amount of the checkpoint written to BB by the applications for a certain period, called Data Write Per Period (*DWPP*) in this paper, is kept quite steady. As so, it is possible to predict future *DWPP* with previous *DWPP* values.

Second, each application has its own checkpoint period. Thus, each application accesses the BB differently during a certain period. HPC applications with short checkpoint period access BB more frequently than ones with long checkpoint period.

Third, HPC applications keep multiple versions of checkpoint for data durability. HPC users have a tendency to keep the old versions of checkpoint without deleting them even though only the most recent version of the checkpoint file is required for restart. According to the paper, multiple versions of checkpoint (three to seventeen) are beneficial for acceptable

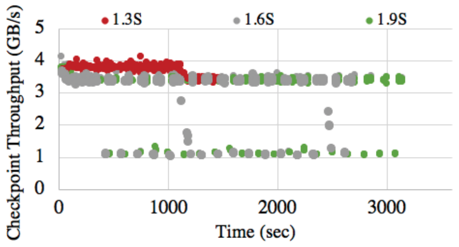


Fig. 1: Checkpoint performance depends on DWPP

error coverage [19]. Since each user requires different reliability, each application has the different number of checkpoint versions.

Fourth characteristic is that HPC applications have different failure rates. Failure is caused by individual components, such as processors, disk, memory, power supplies, network, cooling systems, and the physical connections between them [20]. Since failures of a single component are rare, the large number of the components unavoidably leads to frequent failures [21], [22]. Many prior works have mentioned that the mean time between failure (MTBF) in a single node is about thousands hours, but MTBF of a large-scale cluster with hundreds of node is dozens of hours. Soft errors are also more likely to occur in complicated processing. Therefore, failure rates increase linearly with the number of nodes used by HPC applications [23], [24].

Lastly, there is no locality across the checkpoint files of HPC applications unlike normal data. Temporal locality does not exist across checkpoint files, because the checkpoint file is requested only when in failures. Since a new checkpoint file is created in every checkpoint period, the same checkpoint file is not constantly used unless the application has multiple failures within a single checkpoint period. Also, spatial locality does not exist across checkpoint files. The failure of an application does not affect the failures of the others because each application has different failure rates. So checkpoint files of other applications will not be requested, even if they are stored around a checkpoint file which is accessed due to a failure.

C. Problem Analysis

Unlike the dedicated BB allocation method, the BB oversubscription method allocates more space to applications than the total capacity of BB by allowing the applications to use only in the I/O phase, not the whole lifetime. So applications in the computation phase should yield BB to other applications in the I/O phase via demotion from BB to PFS. Therefore, a data management approach between BB and PFS is required. There are many proposed approaches for multi-tiered system [7], [11], [25], [26], such as an approach between cache, memory and storage and an approach for multi-tier storage of distributed file system. But for the following reasons, these approaches are not suitable for the new HPC storage system where checkpoint dominates most of the I/O traffic.

For the first reason, the existing approaches use static demotion threshold without considering the amount of data to be moved between storage tiers. With the prior approaches, demotion, the process of copying checkpoint files from BB to PFS, is only operated when BB is idle before reaching the threshold. When the used capacity of BB reaches the threshold, demotion is operated concurrently with checkpoint. If the number of users using BB increases, the amount of checkpoint also increases, resulting in increased $DWPP$. The increased $DWPP$ causes a decrease in BB idle time, which reduces the amount of demotion without interrupting the checkpoint. Especially, since the speed of data being stored in BB (write B/W of high-end SSDs) may overwhelm the speed of demotion to PFS (write B/W of low-end SSDs), the BB fills up when there is not enough BB idle time. With the filled BB, applications have to wait until BB has available capacity, leading to significantly low checkpoint performance and high latency. Figure 1 shows checkpoint performance results with different $DWPP$s after setting the same demotion threshold to 90% of total BB capacity. Since S is the capacity of BB, 1.3S, 1.6S and 1.9S write 1.3 times, 1.6 times and 1.9 times the size of BB in a certain period, respectively. With 1.3S, it shows slightly lower performance after 1000 seconds. But, with 1.6S, sometimes the BB is full in the middle of I/O jobs for checkpoint, so performance begins to drop over time. In 1.9S, almost half of the applications get four times lower performance, because they have to be stopped or perform checkpoint with demotion to make available BB capacity.

Another reason is that the existing approaches do not consider the distribution of the I/O jobs for checkpoint. Specifically, even with the same $DWPP$, I/O jobs of applications for checkpoint come in crowds or evenly arrive within the period. When the I/O jobs arrive evenly, there is BB idle time between the job arrivals. So the files are demoted in BB idle time, making enough space in BB for next I/O jobs. However, if the I/O jobs arrive in crowds, lack of BB idle time in between I/O jobs causes little demotion, which leads to BB capacity depletion. As shown in Figure 2, checkpoint performance is highly related to I/O job congestion under the same $DWPP$. There are three I/O job congestion patterns (Low, Med and High) with 1.9S, which represent the rate of how crowded I/O jobs arrived. Low always shows high performance since there is enough idle time between the I/O jobs. On the contrary, when the I/O jobs become to arrive in crowds in Med and High, it results in low checkpoint performance.

Lastly, the existing approaches identify hot and cold checkpoint files using basic algorithms based on data locality, such as FIFO, LRU, and Hotness-aware. (Hot checkpoint files are left in BB, but cold checkpoint files stay in PFS.) However, as mentioned in the section II-B, since checkpoint files across applications do not have the data locality, it is inappropriate to apply the basic algorithms for selecting cold checkpoint files. HPC applications have their own checkpoint period and keep multiple versions of checkpoints. With FIFO algorithm, although old version checkpoints for an application with low checkpoint period are stored in BB, the latest checkpoint with

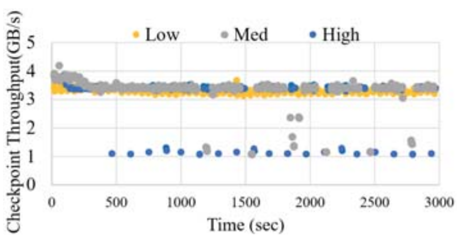


Fig. 2: Checkpoint performance depends on I/O jobs arrival pattern (I/O job congestion)

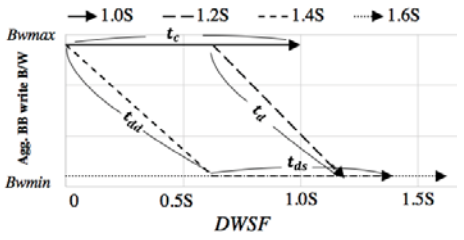


Fig. 3: BBOS demotion management

high checkpoint period can be chosen as the cold data. This makes BB inefficient and leads to low restart performance. Also, a checkpoint file is needed only in case of failure. Since a new checkpoint file is created every checkpoint period, it is very unlikely that the same file will be reused multiple times. Therefore, LRU or Hotness-aware algorithm leads to the low efficiency of BB. Moreover, since there is no spatial locality between checkpoint files, checkpoint files near a failed file do not need to be prefetched or left in BB. The failure rates also need to be considered since HPC applications have their own failure rates. Without taking the failure rates into account, checkpoint files with high failure rate might be chosen as cold data, not checkpoint files with low failure rate.

III. Data management

As mentioned before in the section II-C, because checkpoint/restart characteristics are not fully considered, some applications may suffer from severe performance degradation. To address this problem, we first set the demotion threshold and adjust the speed of checkpoint and demotion depending on $DWPP$. Also we develop data placement policy for the new HPC storage system to improve BB efficiency and restart performance. With our data management approach, checkpoint and demotion are managed for each specific period for expediency. We demote all the data written in this period ($DWPP$) for easy management in the next period.

A. Adaptive Demotion Adjustment

To prevent BB from overflowing, we determine a demotion threshold in consideration of $DWPP$ and I/O job congestion. As shown in the figure 1, since $DWPP$ affects the amount of data to be demoted in a period, the demotion threshold is decided smaller for larger $DWPP$. Even if $DWPP$ is the same, BB may fills up depending on the I/O job congestion shown in the figure 2. At worst, I/O jobs arrive without any idle time for BB. To prepare for the worst, we need to demote data as much as $DWPP$ minus the capacity of BB (S), called C, along with checkpoint execution. In addition, speeds of checkpoint and demotion($Bwmax$, $Bwmin$, $Brmax$ and $Brmin$) also affects the demotion threshold to demote as much as C with checkpoint. Throughput of checkpoint and demotion are is influenced by concurrent execution of write and read operations. When checkpoint and demotion are operated together for C, write and read operations compete for BB resources. Unfortunately, this competition leads to an inverse relationship between write and read bandwidth. As a result, the minimum demotion throughput($Brmin$) is determined by the maximum checkpoint throughput($Bwmax$: maximum write throughput provided by BB). The minimum checkpoint throughput($Bwmin$) is determined by the maximum demotion throughput($Brmax$: maximum write throughput provided by PFS) as shown in equation (1). (m and b values may be different according to various devices.) So, we adjust speed of checkpoint from $Bwmax$ to $Bwmin$, and speed of demotion from $Brmin$ to $Brmax$ after demotion threshold.

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

To calculate the demotion threshold with easy explanation, we show our demotion management by categorizing the patterns of the demotion into three according to $DWPP$ in Figure 3. S is the capacity of BB and $DWSF$ is the amount of data written so far within the period. Within one period, the time given to execute checkpoint at $Bwmax$ without any demotion is t_c, and t_d is the time required to demote C while the checkpoint is going on. t_d is composed of t_{dd} and t_{ds}: the prior is the time where the demotion throughput gradually changes from $Brmin$ to $Brmax$, and the latter is the time when the demotion throughput is $Brmax$ without changing.

1) Pattern 1: Demotion is only performed when BB is idle: As shown in the figure 3, when $DWPP$ is 1.0S, BB does not overflow within the period because $DWPP$ is the same as the capacity of BB. Thus, checkpoint is possible with the $Bwmax$ without concurrent execution of any demotion.

2) Pattern 2: Demotion is performed with checkpoint for some time: As with 1.2S in the figure, $DWPP$ is larger than the capacity of BB(S), resulting in a positive value of C for demotion with checkpoint execution. However, C is not so large, so demotion needs to be performed just for some time with checkpoint. The threshold depends on C. The larger C is, the demotion should start earlier. In the case of 1.2S of $DWPP$, the threshold is 0.7S of $DWSF$. That means that demotion is executed even if checkpoint is performed when $DWSF$ reaches 0.7S. Checkpoint throughput is adjusted in between $Bwmax$ and $Bwmin$ for the demotion. With the checkpoint throughput change, demotion throughput is also adjusted in between $Brmin$ and $Brmax$.

3) Pattern 3: Demotion is always performed with checkpoint: As $DWPP$ increases, C grows large enough that demotion shall be started at the same as the checkpoint. The demotion throughput increases from $Brmin$ to $Brmax$

and the larger C, the faster the demotion throughput reaches $Brmax$. With $1.4S$, the threshold for demotion start is 0 and the threshold for demotion with $Brmax$ is $0.6S$. If the demotion is performed with $Brmax$ from the beginning with the checkpoint like $1.6S$, we can handle the greatest capacity. Therefore, $DWPP$ from this scenario becomes the maximum of $DWPP$.

With the following equation (2), the thresholds to start demotion and to demote with $Brmax$ are also determined according to C. Since it is mandatory to demote all data on BB within the period for the next period, the period is decided as in equation (3).

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

$$
(period - (t_c + t_d)) \times Bdmax \geq S \tag{3}
$$

B. Data Placement Policy

To handle the limitations with existing data placement, we develop a data placement policy based on the characteristics of checkpoint/restart. In our data placement policy, promotion is not required. Because there is no spatial locality across checkpoint files, there is no need to prefetch files around the file which is requested for restart. For high restart performance, we select cold files by considering checkpoint version and failure rates. Old version checkpoint files do not need to be in BB, so they have the highest priority to be cold files. If there is no old version checkpoint files in BB, we identify the cold files based on failure rates. In this paper, we decide failure rates of applications depending on the number of used nodes. But as many prior works have mentioned about causes of failures, the failure rates can be decided using the causes.

C. Direct Checkpoint on PFS

We expect a change of HPC storage system in the future that $Brmax$ be close to the $Bwmin$ since BB and PFS can be placed on the same node. Since cold data from BB are destined to be in PFS, these cold data do not need to be written on BB first, wasting the resource of BB. For this reason, we optimize the data management approach by bypassing BB. Since we know failure rates of the incoming checkpoint, we can classify in advance whether the checkpoint is hot or cold by comparing failure rates with the ones of other checkpoints on BB. If the incoming checkpoint is determined to be cold, the checkpoint is directed to be written on PFS. The optimized approach reduces the amount of demotion, which diminishes the concurrent execution of checkpoint and demotion. Thus we can provide higher checkpoint performance.

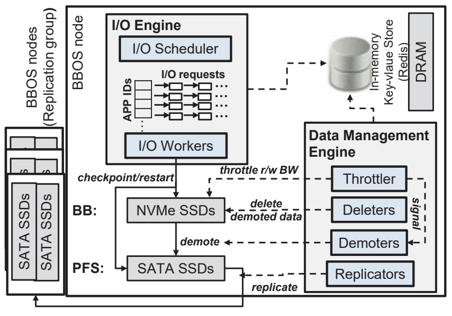


Fig. 4: Architecture of BBOS

IV. Architecture of BBOS

Figure 4 shows the overall architecture of BBOS. BBOS is composed of two engines, *I/O engine* and *Data management engine*, and an in-memory key-value store for efficient engine process.

A. Engines

1) I/O Engine: In this paper, we provide an I/O scheduler for mitigating I/O interference across applications. If we do not schedule the I/O jobs, multiple jobs may arrive simultaneously to BB. This causes resource competition and interferes optimized access pattern of each I/O jobs [27]. In addition, interleaved data from the multiple I/O jobs is saved in the same SSD block, which causes garbage collection overhead [28]. For these reasons, the I/O interference degrades the performance of the applications. The over-subscription method increases the number of the I/O jobs using BB, which may cause I/O congestion more serious. Thus, we schedule I/O jobs so that they do not overlap in *I/O scheduler* of *I/O engine*. We place multiple I/O queues for each BB and assign an individual queue to each application. Then the I/O jobs are transferred to their own queue as shown in Figure 4. Our scheduler operates I/O jobs in the order of the I/O queues so that the I/O jobs across applications do not overlap each other. *I/O engine* also has *I/O workers* to execute I/O jobs for checkpoint and restart. They determine which storage tier the scheduled I/O jobs should access, either BB or PFS, with the help of the in-memory key-value store.

2) Data Management Engine: Data management engine consists of four modules: $throttler$, $demoters$, $deleters$ and $replicators$. $Throttler$ is responsible for dynamically controlling the speed of checkpoint and the demotion. $Demoters$ demote data from BB to PFS by considering checkpoint versions and failure rates. In our management system, demoted data remains in BB like a cache unless there is no space left for a new checkpoint in order to provide high restart performance. Whenever space for new checkpoints is not sufficient, $Deleters$ remove the demote-finish data that still exists in BB. $Replicators$ transfer checkpoint files from storage devices of local PFS node to ones of remote storage nodes within the same replication group.

V. Implementation of BBOS

A. In-memory Key-value Store

We utilize Redis [29], an open source in-memory key-value store to help the processing of the engines. BBOS stores the location of the checkpoint files and important information needed for data management in the key-value store. One of the key-value pairs is used for providing the location of the files. After the checkpoint is completed, *I/O engine* saves the file path for each file name in key-value pair. At the moment, the names of the files are stored in sorted key-value list to identify cold data depends on version number and failure rates. Based on the key-value list, $Demoters$ demote the oldest checkpoint files first, and if there are not any old version checkpoint files, then start to demote the file with the highest failure rates. We also store $DWPP$ and $DWSF$ to decide the demotion threshold and throttle the speed of checkpoint and demotion. Since BBOS does not use page cache for checkpoint and restart, in-memory store is used to utilize unused memory and to facilitate the engine execution.

B. Stable Checkpoint and Demotion Performance

To provide stable checkpoint/restart and demotion performance, data management approach is optimized with new techniques. Checkpoint and demotion speeds are regulated as mentioned in III-A. However, it is difficult to accurately throttle write and read speed for them. Because the number of I/O requests per second from each application varies, the speed of checkpoint and demotion is different even if we send the same number of read requests per second for demotion. The system may not be able to provide stable checkpoint performance due to the inability to demote as much data as it should. For this reason, we use blkio [30] of the cgroup to throttle the speed of checkpoint and restart precisely. In addition, we utilize $send\_file()$ system call [31] to maintain stable demotion performance. For demotion, data must be read from BB and written to PFS. That causes context switching and data copying overhead between user and kernel level, resulting in low and unstable demotion performance. Since $send\_file()$ system call supports zero-copy, we can eliminate the demotion overhead. Furthermore, checkpoint/restart performance may be degraded due to garbage collection. To avoid the garbage collection overhead, we periodically request $TRIM$ after deleting the files. Also, $TRIM$ throughput is managed with blkio in order to minimize performance degradation.

VI. Evaluation

A. Experimental Environment

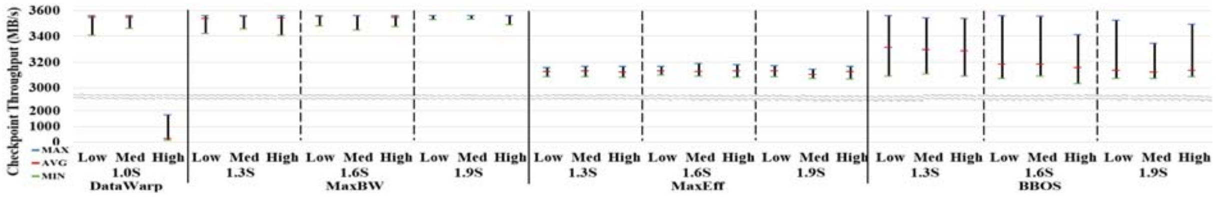
We evaluate our HPC storage management approach with eight compute nodes and a single storage node for BB and PFS. Four of the compute nodes consist of Intel Xeon Phi CPU 7290 processor with 72 physical cores and others are of Intel Xeon Phi CPU 7250 with 68 physical cores. The storage node consists of dual 12-core Intel Xeon Silver CPU 4115 and 32GB memory. For BB, we use four 800GB FADU NVMe SSDs provided by a semiconductor start-up company [32], with the sequential write and read performance up to 920MB/s and 3200MB/s. For PFS, four 4TB Samsung 860 EVO SATA SSDs with are installed. The compute nodes and the storage node are connected with a 100GbE Mellanox SN2100 switch.

We use Gluster file system (GlusterFS) [33] version 5.6 for both BB and PFS and configuration is tuned for high performance. Also, BBOS is implemented by modifying GlusterFS and adding some developed modules. In addition, each variable is determined as following: $Bwmax$ as 3.56GB/s, $Bwmin$ as 3GB/s, $Brmax$ as 1.6GB/s, $Brmin$ as 0.08GB/s and $period$ as 3800 seconds. For experiments, we execute large sequential writes to simulate checkpoint by applying a microbenchmark FIO [34] and the failure rates are decided based on the number of nodes each application uses. According to this paper [35], failure rates and Mean Time Between Failures ($MTBF$) have an inverse relationship. So, to express failure rates of applications simply for experiments, we utilize Mean Time Between Failures ($MTBF$).

To validate our system, we compare BB utilization and checkpoint/restart performance with Datawarp, one of the current HPC schedulers which use the dedicated BB allocation, and two policies of Harmonia [11] which is the only scheduler that uses the BB over-subscription method. Since Harmonia is not an open-source, we make an emulation 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, aims to optimize the BB system efficiency by maximizing the BB utilization. Because the policy wants to maintain high capacity of available BB, it always demotes data at full speed ($Brmax$) even when the checkpoint is performed concurrently. On the other hand, MaxBW from Harmonia aims to provide maximum checkpoint bandwidth to applications. With this policy, the checkpoint and the demotion are not performed at the same time. To describe the demotion threshold of two policies as shown in the figure 3, the demotion threshold of MaxEff is $0S$ of $DWSF$ and one of MaxBW is $1S$ of $DWSF$.

B. Burst Buffer Utilization

To compare BB utilization with each approach, we assume that each application requests 80GB checkpoint once a period. Then we count how many applications can finish the checkpoint within the period, which indicates the maximum $DWPP$ each scheduler can provide. Datawarp shows 0$\sim$100% of BB utilization since it allocates BB capacity as much as the users desire with a dedicated allocation method. In the best scenario, if all users demand BB allocation as much as they need, the total BB capacity is used up within the period, resulting in 100% of BB utilization. In most cases, however, BB utilization is low due to overabundant BB capacity request and the checkpoint/restart characteristics. On the other hand, since Harmonia and BBOS use an over-subscription BB allocation method, they can accommodate more applications within the period than Datawarp. MaxBW needs to ensure


(a) Checkpoint Throughput

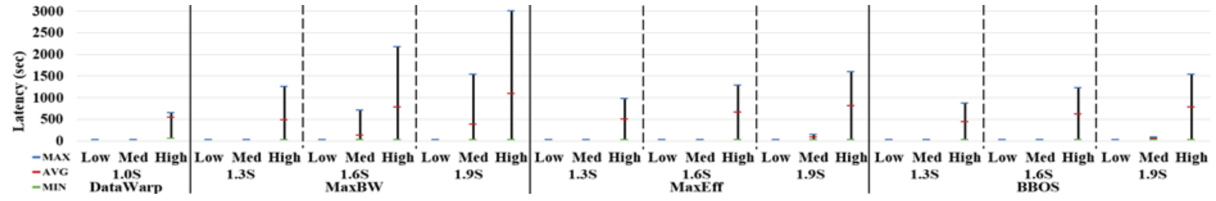

(b) I/O Latency

Fig. 5: Checkpoint throughput and latency depending on $DWPP$

maximum checkpoint throughput of the applications, so the demotion cannot be performed together with the checkpoint. As a result, it shows up to 190% of BB utilization. MaxEff shows 210% of BB utilization because it always performs the demotion at maximum demotion throughput, $Brmax$. To provide maximum $DWPP$, BBOS is performed like MaxEff, which also achieves BB utilization up to 210%.

C. Checkpoint Performance

To validate BBOS, we execute experiments in various situations with different scenarios for I/O job congestion and $DWPP$. Since the maximum $DWPP$ of Datawarp equals the total capacity of BB, we only evaluate Datawarp with $DWPP$ at $1S$ and others with $DWPP$ at $1.3S$, $1.6S$, and $1.9S$. We make different I/O job congestion patterns on following three scenarios: 1) Low: Time interval of each I/O job is equal and evenly distributed throughout the period. 2) Med: Time interval of each I/O job is halved of 1). 3) High: Time interval of each I/O job is shortened in tenth of 1). (e.g., If I/O jobs are requested every 50 seconds under the Low, I/O jobs arrive every 25 seconds and every 5 seconds under the Med and High.) And as in the section VI-B, all applications require 80GB checkpoint once per period.

Figure 5 shows the checkpoint throughput and I/O latency which is the time interval between the time of I/O request and the time of response under various $DWPP$ and I/O job congestion. I/O latency contains 1)the wait time until the end of the previous job to prevent concurrent execution of I/O jobs, 2)the stop time to make available BB space due to full BB and 3)the execution time of I/O job. In Datawarp, since all demotion is possible within enough BB idle time between the I/O jobs, it provides high checkpoint throughput under the Low I/O job congestion. On the contrary, in the High congestion, the checkpoint throughput of each application remains very low due to concurrent execution of the I/O jobs because the jobs arrive even when the previous jobs are not finished. As a result, Datawarp provides the lowest checkpoint throughput and similar average latency even with $DWPP$ at $1.0S$ compared to BBOS.

Unlike Datawarp, Harmonia and BBOS provide I/O scheduling to mitigate I/O interference across applications. Since MaxBW does not execute demotion and checkpoint at the same time, it always gives high checkpoint throughput. But some I/O jobs have to stop for available BB capacity before the execution. With the Low congestion, none of the applications wait to avoid I/O interference or to make space in BB, regardless of $DWPP$, because of sufficient idle time between the I/O jobs. With the Med, as $DWPP$ exceeds $1.6S$, some applications begin to have high latency. The larger $DWPP$, the smaller the idle time within a period, so the latency of $1.9S$ is larger than that of $1.6S$. With the High congestion, applications have to wait to prevent I/O interference and to stop for making space in BB, since there is not enough idle time between the jobs, which results in the highest latency. In addition, MaxBW has extreme performance variance across applications because the maximum latency is too high compared to the average. Figure 6 shows the time excluding execution time in latency of first to 45th I/O jobs with Med I/O congestion at $1.9S$. With MaxBW, no I/O job waits or stops until #36 I/O job, but I/O jobs arriving after #36 have to stop for enough space in BB before execution. Unfortunately, stop time of all the previous I/O jobs is accumulated. Therefore, the later arrived I/O jobs have a longer wait time, resulting in severe performance fluctuation. Unlike MaxBW, BBOS does not stop for making BB capacity because BBOS demotes data in advance so that BB is not full. So the execution time of I/O jobs gets longer, which makes the next I/O jobs to wait. As a result, BBOS shows the wait time gradually increasing from the beginning, but the wait time of I/O jobs does not explode. In conclusion, MaxBW provides higher performance than BBOS when there is no wait time, such as Low and Med of $1.3S$ and Low of $1.6S$ and $1.9S$. But, if there is not enough BB idle time per period or idle time between I/O jobs, MaxBW shows the

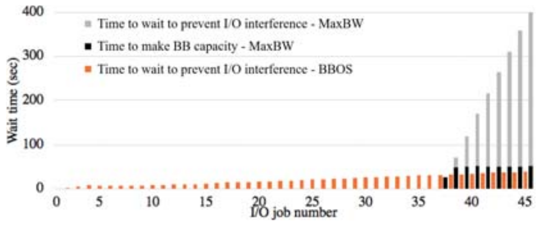


Fig. 6: Wait time of I/O jobs with MaxBW

higher latency and higher performance variance than BBOS, because BBOS prepare for situations where I/O jobs come in crowds by adjusting checkpoint performance.

On the other hand, since MaxEff and BBOS perform demotion in advance for BB not to overflow, they do not stop I/O jobs to make BB capacity before I/O execution. MaxEff shows the lowest checkpoint throughput because this method always demotes data at the highest demotion speed. In this way, they maintain relatively large space in BB, but provides lower checkpoint latency compared to MaxBW. BBOS adjusts checkpoint throughput between $Bwmax$ and $Bwmin$ depending on $DWPP$. The smaller $DWPP$, the higher checkpoint throughput we achieve by avoiding unnecessary concurrent execution of checkpoint and demotion unlike MaxEff. In the absence of wait time for making BB capacity, only the checkpoint throughput determines the latency, so BBOS shows lower latency than MaxEff (But, the latencies are too small to be seen in the figure 5(b)). When $DWPP$ is large, MaxEff shows higher latency than BBOS, even though they perform demotion more aggressively than we do. This is because MaxEff always demotes data at full demotion speed and it takes longer to process the checkpoint, which makes the next I/O jobs to wait longer. In our experiments, the difference in latency of MaxEff and BBOS seems small (about tens of secs) because the difference between $Bwmax$ and $Bwmin$ is not large. If the difference gets greater, BBOS can expect lower I/O latency than MaxEff.

Consequently, BBOS is the novel approach that takes advantage of and complements the shortcomings of MaxBW and MAXEff. By adjusting checkpoint and demotion speed depending on $DWPP$ and I/O job congestion, BBOS always provides relatively high checkpoint throughput and low latency than other approaches.

1) Direct checkpoint on PFS: When the maximum demotion throughput ($Brmax$) is greater than minimum checkpoint throughput ($Bwmin$), we can reduce unnecessary demotion overhead by bypassing BB. We conduct experiments with three different $DWPP$: $1.3S$, $1.6S$, and $1.9S$. Each application requests a 80GB checkpoint with one hour as checkpoint period. $MTBF$ of all I/O applications are set randomly from 0 to 100 minutes, respectively. As shown in Figure 7, the method decreases the amount of demoted data up to 38% by applying the bypassing BB method. Since checkpoints can be written directly on PFS after $DWSF$ becomes larger than BB capacity, the larger the $DWPP$, the longer the time that this method can be applied. Hence, as $DWPP$ gets higher,

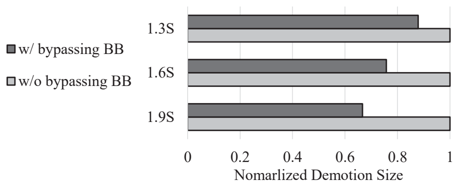


Fig. 7: Direct checkpoint on PFS by bypassing BB

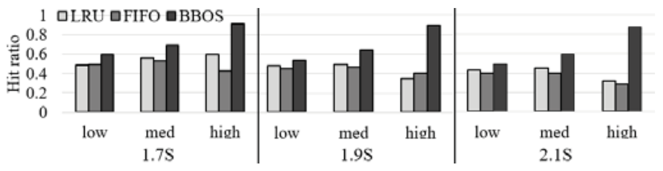


Fig. 8: Hit ratio of restart requests on BB

more checkpoint files that are cold data can be written directly on PFS. As a result, since less demotion is operated with checkpoints concurrently, more applications can experience higher checkpoint throughput.

D. Restart Performance

In this section, we measure the hit ratio of restart on BB in order to compare restart performance. We compare LRU and FIFO algorithms used in most HPC data management with BBOS. Since the ratio of the remaining amount to BB of the total data is different according to $DWPP$, we use three different $DWPP$ for the experiments as follows: $1.7S$, $1.9S$, and $2.1S$. Besides, we use three different variances of $MTBF$ sets. We choose $MTBF$ randomly between the range as following: 0 to 20 minutes(low), 0 to 50 minutes (med), and 0 to 100 minutes (high). We choose the applications which need restart based on the expected $MTBF$. For the sake of simplicity, all the checkpoint periods are fixed to be equal and the checkpoint size is set to 80GB. As shown in Figure 8, BBOS shows the highest hit ratio on BB, which is up to 3.4 times higher in comparison. In all of three methods, the hit ratio increases as $DWPP$ decreases since the checkpoint files have a higher chance of staying on BB. In the case of LRU and FIFO algorithms, however, cold data is chosen based only on the order of written time. As so, the hit ratio for each experiment is largely different and unrelated to the variance of $MTBF$. In contrast, BBOS shows increased hit ratio as the variance gets higher. With low variance, the effectiveness of our system is relatively lower than other scenarios since failure rates of the applications are similar due to low variance of $MTBF$. On the other hand, in case of high variance, the checkpoint files are distributed well on BB and PFS according to the failure rates. As a result, BBOS provides up to 3.1 times higher hit ratio of restart on BB compared to others.

1) Version-aware Data Placement: In order to keep the hit ratio high on BB, BBOS uses the version-aware data placement method by identifying outdated checkpoint files as cold data with the highest priority. To demonstrate the effectiveness of the method, a few assumptions are made for

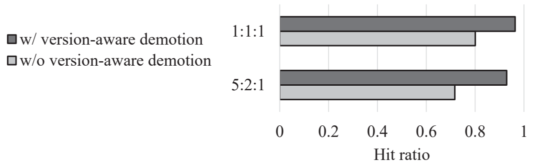


Fig. 9: Version-aware data placement

the following experiment. We choose three checkpoint periods for HPC applications as following: 60 minutes, 30 minutes, and 20 minutes. Each user requests 80GB checkpoint and $MTBF$ is selected randomly from 0 to 100. We assume that the applications maintain three or more versions of checkpoint. Thus, the applications with 60-minute period have one checkpoint version within a period, two versions for a 30-minute period and three versions for a 20-minute period. Lastly, we arrange the portions of the three periods as 1:1:1 and 5:2:1 with $DWPP$ of $1.9S$. Figure 9 shows the comparison of restart hit ratio on BB between the availability of version-aware method. In case of 1:1:1, all of the restart requests can be handled in BB with version-aware method in an ideal situation, because the BB capacity is larger than the total amount of new version checkpoints. However, if we have to make available capacity while there are no old version checkpoints, a single version checkpoint with high $MTBF$ can be selected as cold data. Therefore, on average, 96.4% of the restart requests are performed on BB in actual experiments with the version-aware method. On the contrary, without the version-aware method, selecting cold data is based on only $MTBF$. Fresh checkpoint files with high $MTBF$ exist on PFS and old version files with low $MTBF$ stay on BB. As a result, 80.1% of restart requests are given high restart performance from BB. In case of 5:2:1, as there is large number of applications with low checkpoint period, all new-version checkpoint files cannot be placed in BB. Thus, with the version-aware method, we handle 92.5% restart requests in BB which is slightly less than 1:1:1. Without the method, 71.7% restart requests are performed in BB, which is also lower than 1:1:1. Consequentially, the version-aware method increases the restart requests up to 29.5% which are performed in BB.

VII. Related Work

Many studies related to multi-tiered HPC storage system include BB which consists of expensive resources. Since the emergence of BB, researchers have actively focused on improving checkpoint performance in various ways. To reduce checkpoint overhead on PFS, [1], [25] have developed the multi-level checkpointing mechanism considering the different degree of reliability and the checkpoint cost of each tier in the HPC storage system. [36] has tried to transfer data asynchronously for checkpoint. [28] has observed that BB performance is excessively reduced due to garbage collection when multiple HPC users simultaneously use BB. To mitigate performance reduction, they have assigned isolated blocks to each user using multi-stream SSD. However, these papers do not consider I/O interference across applications which is one of the most important considerations for HPC system. To mitigate the I/O interference, with reference to existing I/O schedulers for PFS [27], some researches [37]–[39] have provided I/O scheduling techniques for BB. [38] has dynamically coordinated I/O jobs based on past I/O behavior of the application and system characteristics. [2] has developed an I/O scheduling technique by reshaping I/O traffic from BB to PFS.

Recently, researchers have started to take an interest in BB utilization as well as checkpoint performance. Some data management solutions [40] have been proposed for HPC multi-tiered storage system to capitalize on the benefits of BB. [15] has suggested a goal-driven data management that automatically manages data as required by applications. The users do not have to move data manually but need to understand application workflow to command data movement. [7], [41] have claimed that HPC applications can have some frequently accessed data. Through I/O profiling, hot data is identified and placed on BB. [42] has regulated I/O traffic by write access pattern of the applications. They have detected randomness in the write traffic and only random writes are stored in BB and sequential writes are propagated to PFS directly. But, checkpoint, occupying most of I/O traffic of the HPC system, does not have hotness and the random access pattern, since entire data of checkpoint is requested sequentially for recovery.

Since the mentioned papers still use the dedicated allocation method, BB cannot be fully utilized. To solve this problem, [26] uses BB over-subscription method. It has proposed five I/O scheduling policies with different aims, but these policies have several limitations because the aims are too narrowly focused. Also it does not care about demotion policy, such as a demotion threshold and demotion speed, and data placement method between BB and PFS for the HPC storage system with BB over-subscription allocation. Therefore, this leads to checkpoint performance fluctuation and low checkpoint/restart performance.

VIII. Conclusion

We proposed BBOS for the new all-flash HPC storage system. Specifically, we over-subscribed BB by allocating BB only during I/O phases, not an entire lifetime for higher BB utilization. In order to mitigate performance reduction caused by over-subscription, we provided the I/O scheduler and the data management module. The I/O scheduler resolved I/O interference across the HPC applications by coordinating the I/O jobs. For data management in the new HPC storage system, we analyzed and utilized the characteristics of checkpoint/restart. Based on the characteristics, we transferred data from BB to PFS transparently by adjusting the thresholds and the speed of the demotion according to $DWPP$. We also identified the cold data by considering different versions and failure rates.As a result, we improved BB utilization at least 2.2 times compared to the dedicated BB allocation method. Also, we guaranteed higher checkpoint throughput without sudden performance reduction and handled 96.4% of restart requests in BB and provided up to 3.1x higher restart performance than others.

Future Work: Since HPC applications have a consistent checkpoint period, we can predict precisely when the next checkpoint request of the application will arrive. This expectation enhances checkpoint throughput without extravagant demotion beforehand. In addition, we predict failure rates of applications only with the number of nodes they use. The exact causes of the failure can be used to determine the failure rates in the future.

Acknowledgment

This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231, and also used resources of the National Energy Research Scientific Computing Center. It is also supported by National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIP)(NRF-2016M3C4A7952587), by the Ministry of Science and ICT(2015M3C4A7065646), and by BK21 Plus(21A20151113068) for Pioneers in Innovative Computing (Dept. of Computer Science and Engineering, SNU).

References

[1] K. Sato *et al.*, "A user-level infiniband-based file system and checkpoint strategy for burst buffers," *Proceedings - 14th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2014*.

[2] T. Wang *et al.*, "TRIO: Burst buffer based I/O orchestration," *Proceedings - IEEE International Conference on Cluster Computing*, 2015.

[3] T. Xu *et al.*, "Explorations of Data Swapping on Burst Buffer," in *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*.

[4] "The ASC Sequoia Draft Statement of Work - https://asc.llnl.gov/sequoia/rfp/02 SequoiaSOW V06.doc."

[5] N. Liu *et al.*, "On the role of burst buffers in leadership-class storage systems," in *IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2012.

[6] O. Yildiz *et al.*, "Eley: On the Effectiveness of Burst Buffers for Big Data Processing in HPC Systems," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*.

[7] T. Xu *et al.*, "Explorations of Data Swapping on Burst Buffer," *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS 2019*.

[8] G. K. Lockwood *et al.*, "Storage 2020: A Vision for the Future of HPC Storage - https://escholarship.org/uc/item/744479dp," Tech. Rep.

[9] John Bent and others, "Serving Data to the Lunatic Fringe: The Evolution of HPC Storage — 2016 USENIX - https://www.usenix.org/publications/login/summer2016/bent."

[10] Z. Yang *et al.*, "AutoTiering: Automatic data placement manager in multi-tier all-flash datacenter," in *IEEE 36th International Performance Computing and Communications Conference, IPCCC 2017*.

[11] X. Meng *et al.*, "HFA: A Hint Frequency-based approach to enhance the I/O performance of multi-level cache storage systems," *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS 2014*.

[12] "Cori Burst Buffer," in *https://www.nersc.gov/users/computational-systems/cori/burst-buffer/*.

[13] "DATAWARP - https://www.cray.com/products/storage/datawarp."

[14] "Slurm Workload Manager - https://slurm.schedmd.com/publications.html."

[15] W. Shin *et al.*, "Data Jockey: Automatic Data Management for HPC Multi-tiered Storage Systems."

[16] R. A. Ashraf *et al.*, "Shrink or Substitute: Handling Process Failures in HPC Systems using In-situ Recovery."

[17] J. W. Young *et al.*, "A first order approximation to the optimum checkpoint interval," *Communications of the ACM*, 1974.

[18] J. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Generation Computer Systems*, 2006.

[19] L. Guoming *et al.*, "When is Multi-version Checkpointing Needed?" in *Proceedings of the 3rd Workshop on Fault-tolerance for HPC at extreme scale*.

[20] A. Saurabh *et al.*, "Understanding and Exploiting Spatial Properties of System Failures on Extreme-Scale HPC Systems," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*.

[21] N. Nichamon *et al.*, "Reliability-Aware Approach: An Incremental Checkpoint/Restart Model in HPC Environments," in *2008 IEEE international Symposium, Cluster Computing and the Grid (CCGRID)*.

[22] A. Saurabh *et al.*, "Adaptive Incremental Checkpointing for Massively Parallel Systems," in *Proceedings of the 18th annual international conference on Supercomputing*.

[23] J. Hui *et al.*, "Optimizing hpc fault-tolerant environment: an analytical approach," in *2010 39th International Conference on Parallel Processing*.

[24] P. F *et al.*, "system-level fault-tolerance in large-scale parallel machines with buffered coscheduling," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings*.

[25] A. Moody *et al.*, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2010*.

[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," *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, vol. 2018-Septe, pp. 290–301, 2018.

[27] M. Dorier *et al.*, "CALCioM: Mitigating I/O interference in HPC systems through cross-application coordination," *Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS 2014*.

[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 *2017 IEEE International Conference on Cluster Computing (CLUSTER)*.

[29] "Redis-https://redis.io/."

[30] "blkio-https://www.kernel.org/doc/documentation/cgroup-v1/blkio-controller.txt."

[31] "sendfile - http://man7.org/linux/man-pages/man2/sendfile.2.html."

[32] "FADU, THE SSD EXPERT - http://www.fadu.io."

[33] "Gluster - https://www.gluster.org/."

[34] "Fio - https://github.com/axboe/fio."

[35] D. Tiwari *et al.*, "Lazy Checkpointing: Exploiting Temporal Locality in Failures to Mitigate Checkpointing Overheads on Extreme-Scale Systems," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*.

[36] K. Sato *et al.*, "Design and modeling of a non-blocking checkpointing system," in *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*.

[37] S. e. a. Thapaliya, "Managing I/O Interference in a Shared Burst Buffer System," *Proceedings of the International Conference on Parallel Processing, 2016*.

[38] A. Gainaru *et al.*, "Scheduling the I/O of HPC Applications under Congestion," *IEEE 29th International Parallel and Distributed Processing Symposium, IPDPS 2015*.

[39] A. Kougkas *et al.*, "Leveraging burst buffer coordination to prevent I/O interference," in *Proceedings of the 2016 IEEE 12th International Conference on e-Science*.

[40] A. Kougkas, "Hermes: A multi-tiered distributed I/O Buffering System for HDF5," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing - HPDC '18*.

[41] G. Zhang *et al.*, "Automated lookahead data migration in SSD-enabled multi-tiered storage systems," in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*.

[42] X. Shi *et al.*, "SSDUP: a traffic-aware ssd burst buffer for HPC systems," in *Proceedings of the International Conference on Supercomputing - ICS '17*.