Ph.D. DISSERTATION

Efficient I/O Management Schemes for All-Flash HPC Storage Systems

플래시 기반의 고성능 컴퓨팅 스토리지 시스템을 위한
효율적인 입출력 관리 기법

August 2020

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Hanul Sung

Ph.D. DISSERTATION

Efficient I/O Management Schemes for All-Flash HPC Storage Systems

플래시 기반의 고성능 컴퓨팅 스토리지 시스템을 위한 효율적인 입출력 관리 기법

August 2020

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Hanul Sung

Efficient I/O Management Schemes for All-Flash HPC Storage Systems

플래시 기반의 고성능 컴퓨팅 스토리지 시스템을 위한 효율적인 입출력 관리 기법

지도교수 엄현상

이 논문을 공학박사 학위논문으로 제출함

2020 년 7 월

서울대학교 대학원

전기·컴퓨터 공학부

성한울

성한울의 공학박사 학위논문을 인준함

2020 년 6 월

Abstract

Most I/O traffic in high performance computing (HPC) storage systems is dominated by checkpoints and the restarts of HPC applications. For such a bursty I/O, new all-flash HPC storage systems with an integrated burst buffer (BB) and parallel file system (PFS) have been proposed. However, most of the distributed file systems (DFS) used to configure the storage systems provide a single connection between a compute node and a server node, which hinders users from utilizing the high I/O bandwidth provided by an all-flash server node. To provide multiple connections, DFSs must be modified to increase the number of sockets, which is an extremely difficult and time-consuming task owing to their complicated structures. Users can increase the number of daemons in the DFSs to forcibly increase the number of connections without a DFS modification. Because each daemon has a mount point for its connection, there are multiple mount points in the compute nodes, resulting in significant effort required for users to distribute file I/O requests to multiple mount points. In addition, to avoid access to a PFS composed of low-speed storage devices, such as hard disks, dedicated BB allocation is preferred despite its severe underutilization. However, a BB allocation method may be inappropriate because all-flash HPC storage systems speed up access to the PFS.

To handle such problems, we propose an efficient user-transparent I/O management scheme for all-flash HPC storage systems. The first scheme, I/O transfer management, provides multiple connections between a compute node and a server node without additional effort from DFS developers and users. To do so, we modified a mount procedure and I/O processing procedures in a virtual file system (VFS). In the second scheme, data management between BB and PFS, a BB over-subscription allocation method is adopted to improve the BB utilization. Unfortunately, the allocation method aggravates the I/O interference and demotion overhead from the BB to the PFS, resulting in a degraded checkpoint and restart performance. To minimize this degradation, we developed an I/O scheduler and a new data management based on the checkpoint and restart characteristics.

To prove the effectiveness of our proposed schemes, we evaluated our I/O transfer and data management schemes between the BB and PFS. The I/O transfer management scheme improves the write and read I/O throughputs for the checkpoint and restart by up to 6- and 3-times, that of a DFS using the original kernel, respectively. Based on the data management scheme, we found that the BB utilization is improved by at least 2.2-fold, and a stabler and higher checkpoint performance is guaranteed. In addition, we achieved up to a 96.4% hit ratio of the restart requests on the BB and up to a 3.1-times higher restart performance than that of other existing methods.

Keywords: Operating System, High Performance Computing, Distributed File System, Parallel File System, File I/O Management, Data Management, Network Scalability, Virtual File System, Burst Buffer, Checkpoint, Restart, Demotion

Student Number: 2013-20801

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As computational capability has increased to over one petaflop, a large number of system components have been used in high-performance computing (HPC) systems, thereby causing an increase in the overall system failures [1–3]. As a failsafe, HPC applications aggressively utilize a checkpoint/restart, which is the most commonly applied fault tolerance mechanism. This has resulted in checkpoints dominating up to 75%∼80% of I/O traffic of HPC systems [2, 4] and significantly generating a bursty I/O. A bursty I/O is difficult to handle for a parallel file system (PFS), which has been a foundation to the storage tier of HPC systems consisting of hard disk devices (HDDs) and low speed network adapters. As the capabilities of storage and network devices advance, a burst buffer (BB), which is composed of high-end flash SSDs (e.g., 3D XPoint SSD and NVMe SSD) [5,6] and high-speed network (i.e., 100 GbE network) adapters, has been introduced as a new storage tier between the compute nodes and PFS. Moreover, there have been recent attempts to merge a BB with a PFS [7,8]. As the cost-per-bit of flash continues to decrease, it is becoming possible to replace

HDDs with low-end TLC SSDs [9]. The National Energy Research Scientific Computing Center (NERSC), which is a primary scientific computing facility announced that the next supercomputer with an all-flash storage system, called PERLMUTTER [10], will be delivered in 2020. It is therefore expected that a new all-flash HPC storage system with an integrated BB and PFS will be widely adopted in the near future. We believe that this new storage system will include high-end SSDs (e.g., 3D XPoint SSD and NVMe SSD) for the BB (performance) and low-end SSDs (TLC SATA SSD) for the PFS (capacity) placed on the same node.

Although some researchers have studied ways to utilize an all-flash storage system, none have focused on connections between a compute node and a server node despite I/O transmission scalability being one of the most critical issues in HPC storage systems. Unfortunately, existing distributed file systems (DFSs) used to configure the BB storage tier have adopted a single connection between the compute and BB server nodes. For this reason, even with expensive high-speed storage devices and network adapters on the BB node to handle a bursty I/O, users do not experience the high I/O bandwidth provided by the all-flash storage server node. Figure 1.1 shows write I/O bandwidths with and without widely used DFSs, such as GlusterFS, NFS, and Lustre. We used a single compute node and a server node and executed 16 threads to submit synchronous I/O requests. In addition, we adopted Ext4 as a local file system for the DFSs. Here, *LocalFS* achieves a rate of 2.6 GB/s using only Ext4 on the server node without DFSs. This is the ideal I/O bandwidth providable by a single server node. However, with GlusterFS, we can only achieve a rate of 600 MB/s, although 2.6 GB/s is theoretically achievable. Whereas GlusterFS is a user file system applying FUSE, NFS4 is implemented in a kernel and shows a low I/O bandwidth of 1.2 GB/s. Lustre is also developed in a kernel, but

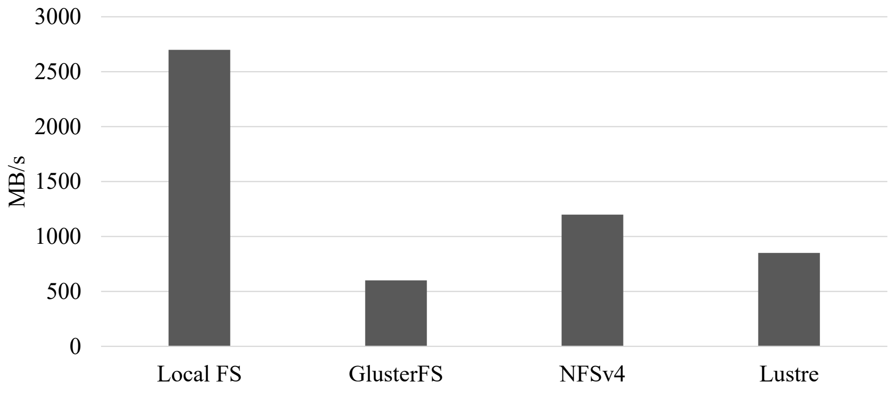


Figure 1.1: I/O throughputs of PFSs

provides 850 MB/s, which is 3-times lower than that of *LocalFS*. Such DFSs do not fully utilize the I/O bandwidth provided by the BB server node owing to the use of a single connection.

To solve the performance bottleneck caused by a single connection, all types of DFSs need to be modified to increase the number of connections between the compute and server nodes. However, because most DFSs have complicated structures, significant effort is required to scale the number of connections, and the DFSs must therefore continue using single connections. Therefore, most users are provided with a high I/O bandwidth by allocating a BB composed of multiple server nodes. However, this method causes high performance fluctuations owing to the sharing of server nodes with other users.

Because the client daemon establishes a new connection with the server daemon through the mount command upon startup, some users create multiple client daemons of the DFSs using a mount command to increase the number of connections. We created several client daemons with GlusterFS to increase the number of connections (between the compute and server nodes), and the write I/O throughputs for a checkpoint are increased by up to 2000 MB/s, as shown

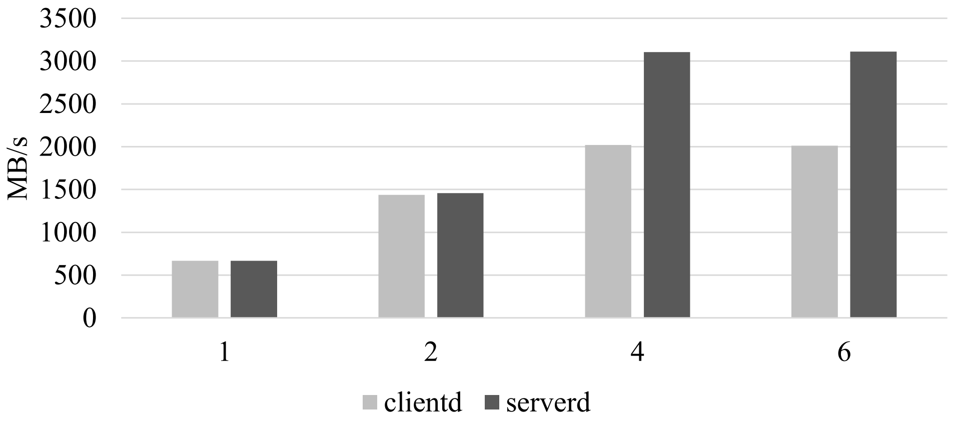


Figure 1.2: I/O throughputs of GlusterFS depending on the number of connections

in *clientd* of Figure 1.2.

Some other users have increased the number of connections by creating multiple server daemons in the same node. After a new server daemon is created, a new connection is established between the client and server nodes through the mount command. As shown in *serverd* of Figure 1.2, this method not only improves the write I/O throughput for a checkpoint as the number of connections increases, it also performs better than *clientd* because both methods have the same number of connections, despite *clientd* having only one server daemon and *serverd* having multiple server daemons. However, these methods require user effort to distribute I/O requests to multiple mount points because the number of mount points is same as the number of connections. If users have multiple files to write, they must manually choose and set one mount point for each file among multiple mount points to fully utilize multiple connections. In addition, if the number of files is smaller than the number of connections, the users will have greater difficulty distributing I/O requests of a file to multiple connections.

In previous HPC storage systems where a PFS layer consisting of low-speed HDD storage devices and low-speed network devices is located on different server nodes from those of the BB storage layer, because of the substantial performance differences offered by the BB and PFS, HPC users may prefer a dedicated BB allocation for the complete lifetime of their HPC applications to avoid access to the PFS as much as possible. However, this allocation method causes a severe underutilization of an expensive BB for two reasons. First, some users eagerly request BB resources (e.g., up to six-times the normal rate [11]) for I/O error prevention, performance scalability, a scale-up of the network connections, and complicated data movement between the BB and PFS, even when they actually utilize only a small portion (e.g., 5% of the BB is used per hour according to the logs from NERSC CORI [12]). Second, because HPC applications use a BB for only the I/O phases, BB stays idle for the remaining time. Checkpoints, which dominate the I/O traffic of HPC systems, are rarely requested, for example, at a rate of once per hour or several minutes, and thus the BB resources are mostly wasted. The PFS in the new system consists of a high-speed network and low-end SSDs, which significantly reduces the overhead of the PFS access. It is therefore not worth using a dedicated BB allocation method owing to the penalty of a BB underutilization.

To address these problems in Figure 1.3, we propose an efficient user-transparent I/O management scheme for all-flash HPC storage systems. With this scheme, we achieve faster checkpoints and restarts, which occupy most of the I/O traffic in an HPC system while improving the utilization of all-flash HPC storage systems.

First, we developed a user-transparent I/O transfer management subsystem in a virtual file system (VFS) to reduce the efforts of DFS developers and users applying multiple connections to handle the problem shown in Figure 1.3 ①.

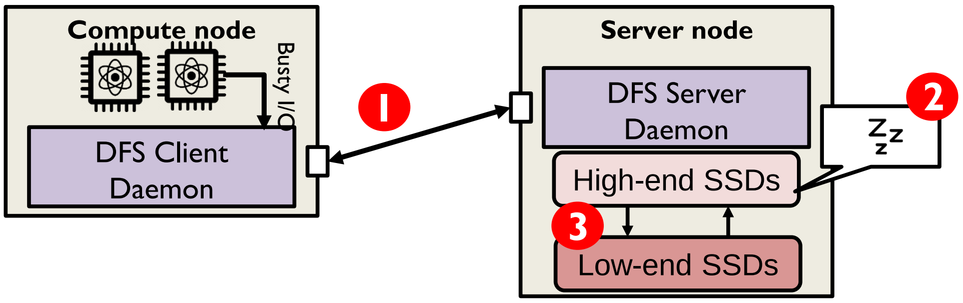


Figure 1.3: Problems of new all-flash HPC storage systems

To decrease the number of DFS modifications as much as possible, we increased the number of connections by making multiple client and server daemons for a DFS of a BB through a mount command. With this approach, each compute node has multiple mount points, resulting in excessive user effort in distributing I/O requests to multiple connections. Therefore, we implemented a new mount procedure and new I/O processing procedures. We changed the vertical mount hierarchy to a horizontal mount hierarchy for supporting multiple connections and exposed only one mount point among the multiple mount points to users. In addition, we added two scheduling policies for an I/O transfer, *TtoS* and *TtoM*. Through these policies, I/O requests are evenly distributed to multiple connections.

Second, we adopted a BB over-subscription allocation method instead of a dedicated BB allocation method for improving the BB utilization (for the problem shown in Figures 1.3 ② and ③. The method allocates the BB capacity only during the I/O phase, and not for the entire lifetime of HPC applications, resulting in a reduction in wasted BB resources. However, this method may affect the performance of the checkpoint/restart because a capacity larger than the total capacity of the BB is allocated to the HPC applications. To handle this problem, we transparently manage the data movement between the BB and PFS and schedule the I/O jobs based on characteristics of the checkpoint/restart. We analyzed the characteristics of a checkpoint/restart, adjusted the speed of the demotion from the BB to the PFS and determined the data placement between the BB and PFS based on the characteristics analyzed.

To demonstrate the effectiveness of our schemes, we evaluated our I/O transfer and data management schemes between the BB and PFS. For the I/O transfer management scheme, we used GlusterFS as a DFS for the BB and kernel v5.3.12. Our experimental results show that our scheme increases the write and read I/O throughputs for a checkpoint and restart by increasing the number of connections and provides the best I/O throughputs of GlusterFS in a given experimental environment. In addition, it improves the write and read performances by up to 6- and 3-times those of the DFSs with the original kernel, respectively. For the data management scheme, we compared our scheme to Datawarp, a representative of the current HPC schedulers that uses a dedicated BB allocation method, and Harmonia [13], which is the only approach available when considering a BB over-subscription. Our scheme improves the BB utilization considerably while providing a high checkpoint performance. In addition, it provides a high checkpoint capability and performs up to 96.4% of the restarts on a BB by utilizing the characteristics of a checkpoint/restart.

The contributions are summarized as follows:

- We discuss the issue of I/O bandwidth bottleneck caused by a single connection provided by major distributed file systems for a new all-flash HPC storage system that can be avoided.

- We increase I/O bandwidth from a single storage server node of DFSs via multiple connections by implementing an I/O subsystem in VFS without modifying DFSs and user applications so that users can utilize with I/O bandwidth of the server node transparently.

- We show that the I/O subsystem improves write and read I/O performances by up to 6x and 3x, respectively, compared with original kernel.

- We adopt the over-subscription BB allocation method to handle BB underutilization problem caused by the dedicated BB allocation method.

- We analyzed the checkpoint/restart characteristics of HPC applications. We observed 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 found the characteristics of HPC applications is highly related to low checkpoint/restart performance with existing data management approach.

- We propose a novel data management scheme between BB and PFS for all-flash HPC storage systems based on the characteristics of checkpoint/restart to provide high BB utilization as well as high checkpoint/restart performance. The scheme schedules I/O jobs, adjusts demotion threshold and speeds of checkpoint and demotion adaptively and manages data placement between BB and PFS.

- We implemented the scheme by adding some modules and modifying GlusterFS, one of the most popular distributed file system. It shows increased BB utilization and improved checkpoint/restart performances compared with prior works.

This dissertation is structured as follows:

- **Chapter 2** covers the background about Burst Buffer, virtual file system, network bandwidth and characteristics of checkpoint and restart for providing high I/O throughput and high utilization in a new all-flash HPC storage systems.
- **Chapter 3** demonstrates the motivation in which I/O transparent management and data management for all-flash HPC storage systems are proposed with problems and limitations of existing HPC storage systems.
- **Chapter 4** introduces I/O transfer management scheme for all-flash HPC storage systems. We first explain design and architecture of the scheme and describe the details of implementation. Next, we evaluate our scheme in an all-flash HPC storage system with various configurations.
- **Chapter 5** introduces user-transparent data management for all-flash HPC storage systems. We start with explaining the details of design and architecture of our scheme. Then, we describe the implementations of our scheme and evaluate the scheme on an all-flash HPC storage system with various experimental scenario.
- **Chapter 6** explains the several works related to our schemes.
- **Chapter 7** summarizes and concludes the dissertation. It also describes future direction.

Chapter 2

Background

2.1 Burst Buffer

HPC systems consist of thousands of compute nodes and storage nodes for scientific applications requiring complicated computation. Because of high cost of flash storage devices, PFSs have been configured with HDD-based storage nodes. A rapid growth in computing power has occurred a significant performance gap between compute nodes and storage nodes, making it difficult for HDD-based PFSs to handle massive amounts of data generated by compute nodes.

To handle this performance bottleneck caused by HDD-based PFSs, a new storage layer, called Burst Buffer composed of flash devices, has placed between compute nodes and PFSs. BB has mitigated the performance gap by handling bursty I/O, which PFSs were difficult to process, with high I/O bandwidth. There are two types of BB, node-local BB and remote shared BB, depending on where BB is placed. Each BB type has advantages and disadvantages, so HPC

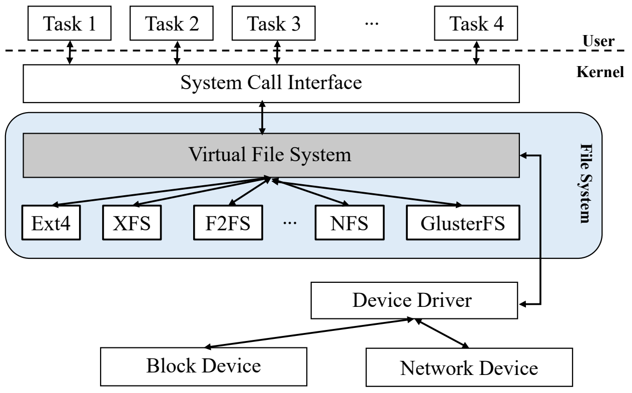


Figure 2.1: Virtual File System

providers select appropriate BB type according to purposes of their system.

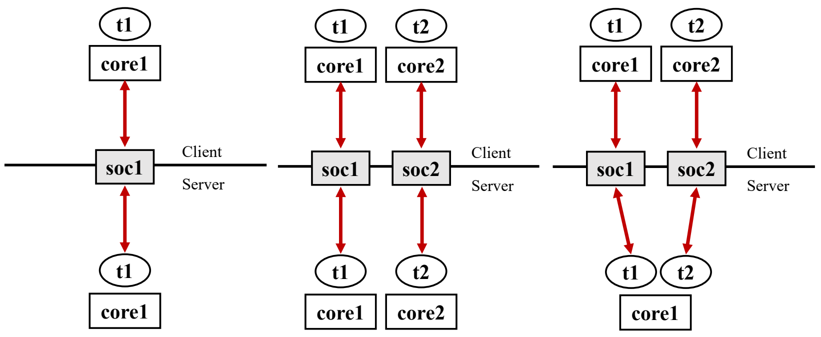
Node-local BB is used as local file system for each compute node by installing a flash storage devices on the compute node. For this reason, the overall performance of node-local BB increases linearly with the number of compute nodes, so it is excellent in performance scalability. In addition, since I/O requests are performed without network overhead for accessing remote nodes, node-local BB provides faster I/O processing. But, since each compute node uses a unique local file system, it causes additional efforts to share data between multiple compute nodes.

Unlike node-local BB, remote shared BB, which is a BB type used by most super computers, has unique physical nodes for BB. Multiple compute nodes use the same file system of BB, which makes they share data. Otherwise, performance isolation is not guaranteed in remote shared BB due to shared resource contention, because compute nodes share several hardware resources in BB nodes including flash devices.

2.2 Virtual File System

Virtual file system (VFS) is a common interface provided on file systems so that users can use multiple file systems without application modification. Users perform file operations using system call without worrying about the type of connected file system or data on local disk or network. As shown in Figure 2.1, VFS adopts common file model to hide specific file processing procedures of file systems and manages files. To do this, virtual file system has four objects, super block, inode, file and dentry.

Super block object stores information of mounted file systems, such as file system type, state, size, and inode. Virtual file system manages multiple file systems using the object. Inode object contains information related to a specific file, called metadata. Since VFS handles file operations with the inode, file systems have to built inode objects for their files. File object is required to manage information associated with files opened by tasks (applications). The object remains in memory only while each task accesses inode objects. In order for tasks to access files, the inode objects of the files must be linked to file objects associated with its tasks. To reduce overhead caused by process, Dentry object keeps the link between file and inode in cache.



(a) Single connection

(b) Two connections with isolated core for socket threads on a server node

(c) Two connections with shared core for socket threads on a server node

Figure 2.2: Network Bandwidth based on the ability of socket's threads

2.3 Network Bandwidth

PFSs create sockets for a connection with transport types, such as TCP, UDP, and RDMA. There are many factors, such as transmission unit size, that determine network bandwidth, but the ability of socket's threads to send and receive requests is overwhelming.

We measured maximum achievable network bandwidth between a client node and a server node with different the number of connections and cores for I/O transmission using a iPerf3 benchmark.

Figure 2.2a shows a single connection between the client node and the server node and each node has one socket thread for the connection. With *soc*1, we get 25Gbits/sec network bandwidth. For two connections in Figure 2.2b, there is one socket thread for each connection on each node. That is, each node has two threads for I/O transmission. As the number of connections and the number of socket threads doubled, the total CPU utilization for I/O transmission doubled and the network bandwidth was increased as almost two times high as

47Gbits/sec. And with two connections at Figure 2.2c, we make each client-side socket thread using one core, and the two server-side threads sharing one core. Despite having two connections, it show 28Gbits/sec network I/O bandwidth, which proves that the network bandwidth is greatly affected by how much the socket threads perform for I/O transmission, not by the number of connections. However, even if multiple socket threads execute I/O transmission for a single connection, it still cannot achieve high network bandwidth due to lock contention for the socket among the threads.

Therefore, in order to provide the aggregated I/O bandwidth of multiple flash devices from server to client, multiple connections must be created.

2.4 Mean Time Between Failures

The applications are recommended to use a technique called checkpoint for fault tolerance. They periodically pause their operations to save current data in Burst Buffer for the checkpoint. When in case of a failure, they can continue their operations from the latest checkpoint.

Mean Time Between Failures (MTBF) represents the average time period for each failure. The applications with more failures have low MTBF, and the ones with fewer failures have higher MTBF. According to this paper [14], the failure occurs frequently in proportion to the inversion of MTBF also on the real-world scenarios. Since the applications recover their data after the failure, the application with a higher failure rate has the higher chance of read request.

2.5 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 error coverage [15]. 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 [16]. Since failures of a single component are rare, the large number of the components unavoidably leads to frequent failures [17, 18]. 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 [19, 20].

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.

Chapter 3

Motivation

3.1 I/O Transfer Management for HPC Storage Systems

In this section, we describe the limitations of distributed file systems (DFSs) for utilizing a high I/O bandwidth of an all-flash BB.

3.1.1 Problems of Existing HPC Storage Systems

In the initial DFSs, low-speed storage devices, such as hard disks, resulted in a performance bottleneck. As storage technology developed, a software stack overhead of the DFSs began to occur. Despite a number of researchers having alleviated the software stack overhead, a performance bottleneck remains between client and server nodes of DFSs.

To determine the unresolved performance bottleneck, we analyze the I/O performance provided by the users of GlusterFS, a popular DFS. We use two nodes for a client and server of GlusterFS. These nodes are connected by a 100

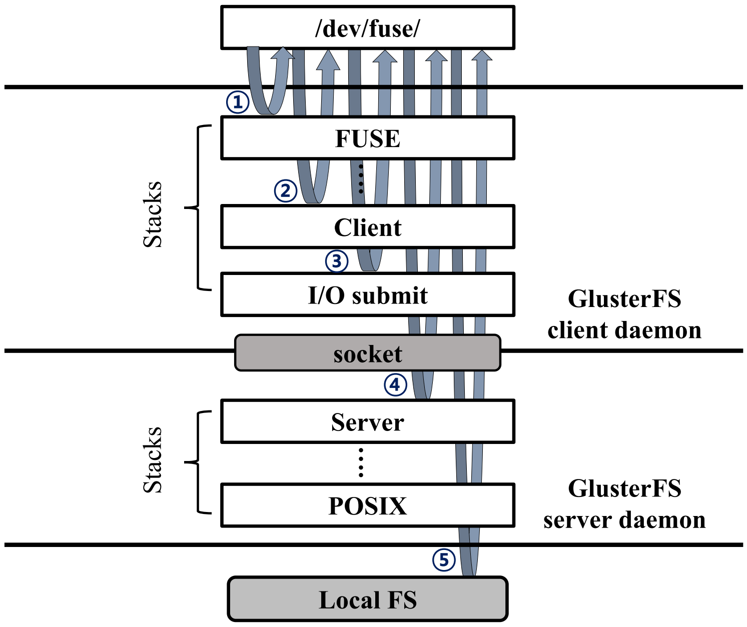


Figure 3.1: Loopback protocol in GlusterFS

GbE network switch, and the server node has five NVMe SSDs, which achieve rates of 3 GB/s and 16 GB/s for a sequential write and read, respectively. Figure 3.2 shows the sequential write I/O throughputs using a loop back method at various points in the GlusterFS, as shown in Figure 3.1. GlusterFS uses the Filesystem in Userspace (FUSE) interface as a user-level file system. First, we execute the loopback at the *FUSE* stack where GlusterFS receives I/O requests from the FUSE, as if the requests are successfully processed, as shown in ① of Figure 3.1. Second, we apply a loopback from a *Client* stack such as ②. Third, we apply the loopback at an *I/O submit* stack, which transfers I/O requests from the client node to the server node, such as in ③. Fourth, once the GlusterFS server receives an I/O request, such as in ④, we apply a loopback. Finally, we measure the performance normally without a loopback method, as in ⑤.

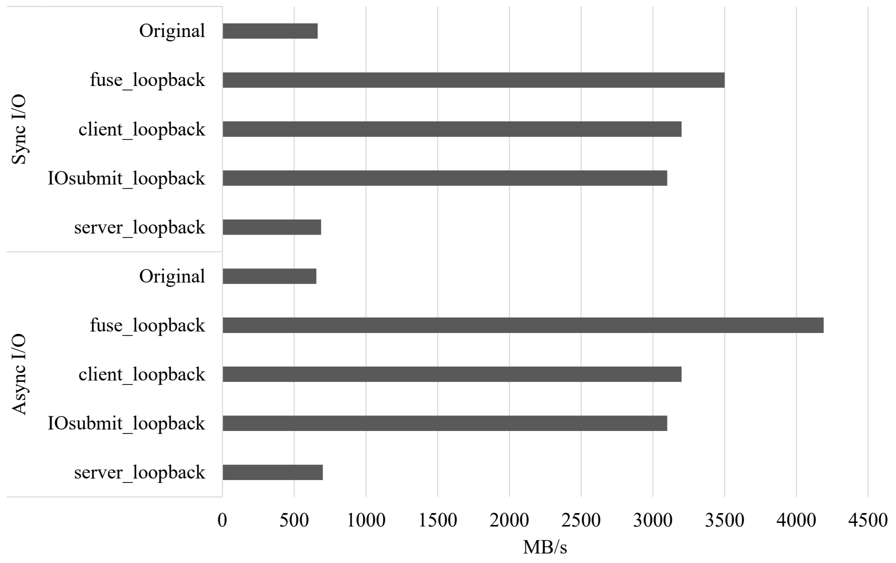
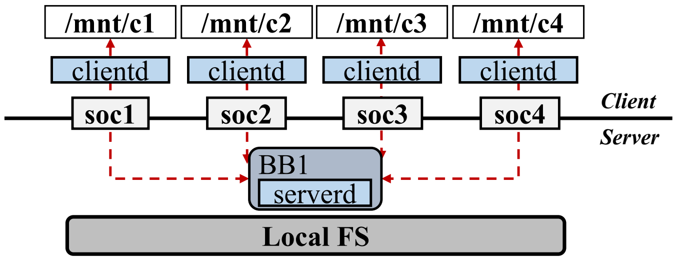
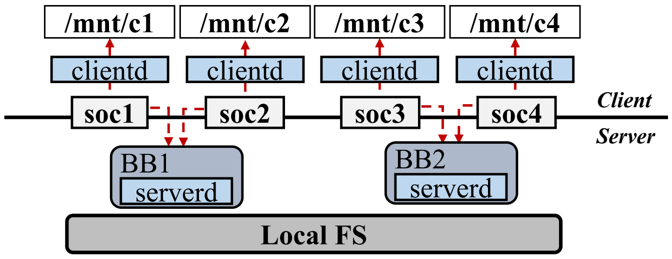


Figure 3.2: I/O throughput of GlusterFS analysis with a loopback method

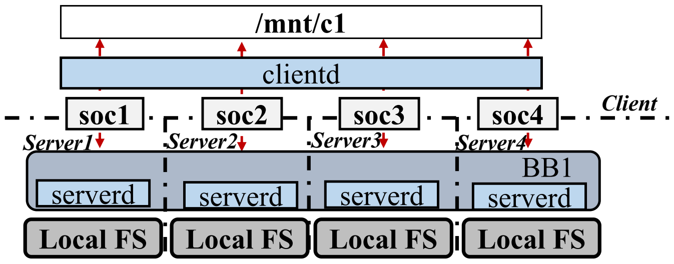
Using the loopback method on the GlusterFS client side, as the requests go through more stacks, the performance decreases in both sync and async I/O modes, although a bandwidth of over 3 GB/s is continuously provided. As shown in the result of *server_loopback*, when the requests transfer from the client node to the server node, write throughputs fall below 700 MB/s even if the data have not been written to the actual storage devices. In addition, the throughputs are extremely similar to that of a normally performed GlusterFS with no loopback method. In other words, the server node can provide more than a 3 GB/s write performance, whereas DFS users only utilize an I/O bandwidth of approximately 600 MB/s owing to the performance bottleneck on the DFS network layer with a single connection.



(a) Scale up connections by making multiple DFS client daemons



(b) Scale up connections by making multiple DFS server daemons



(c) Scale up connections by connecting multiple DFS server nodes

Figure 3.3: Existing solutions for utilizing multiple connections in DFSs

3.1.2 Limitations of Existing Approaches

Because of a performance bottleneck of the DFS network layer with a single connection, all-flash HPC storage systems composed of a DFS have a bounded I/O throughput. To handle this problem, multiple connections are required for a DFS to utilize the aggregated I/O bandwidth of the server node. However, because most DFSs are implemented into complicated structures, it is a difficult and time-consuming task to modify the systems to allow multiple connections between the client and server nodes.

Users can utilize the high I/O bandwidth by manually increasing the connections in three ways, as shown in Figure 3.3. First, they use multiple connections between the client and server nodes by creating client daemons through a mount command, such as shown in Figure 3.3a. For instance, if a user wants four connections between the client and server nodes, the user executes the following mount command, "*mount -t glusterfs BB1 /mnt/c(1∼4)*," four times. Because there are mount points connected to the same DFS in the same sever, the users must distribute I/O requests to them (c1, c2, c3, and c4) evenly to utilize the multiple connections. The easiest way is to use a different mount point for each file, although to do so the user must manually modify the applications. Moreover, if the number of files is not larger than the number of connections, some connections may not be utilized, and thus clients have to distribute I/O requests for the same file to the multiple connections in a complicated way.

Another way is to increase the number of connections by creating multiple DFSs and server daemons on the same server node. As shown in Figure 3.3b, a user makes two DFSs in the same server node and connects two client daemons to one server daemon through the mount commands, *"mount -t glusterfs BB1 /mnt/c(1,2)"* and *"mount -t glusterfs BB2 /mnt/c(3,4)"*. This allows the user to have four connections, but requires more complicated steps in addition to the effort required through the first approach. Because the two DFSs are different file systems, they manage files separately even if they use the same devices. Therefore, the same DFS must be used when the user executes I/O jobs multiple times on the same file. Otherwise, the same file exists in two DFSs and cannot guarantee data consistency.

Finally, as shown in Figure 3.3c, users can group multiple server nodes to make a single DFS, which is the most widely used approach. Each server node has a server daemon and connection to the client node. If the client node with a 100 GbE network adapter is provided with a 600 MB/s I/O bandwidth from a single server node, the client node must establish connections with approximately 10 server nodes. If the client uses too many server nodes, the user may experience a performance fluctuation owing to contention at the server node. As the number of server nodes used increases, the chance to share server nodes with other users increases, and the user obtains an unstable performance. We experimented using an IOR benchmark by applying five configurations and changing the number of OSTs and the stripe size on a Cori supercomputer in NERSC. As a result of applying each configuration five times, we found up to a 4-fold difference in performance despite the same configuration. Although it is proper for a single client node to use multiple server nodes for replication, as few server nodes as possible should be utilized for a stable performance. Despite this serious problem, this approach is used most often because many DFSs check the source IP and destination IP addresses and allow only a single connection between them. If we create multiple daemons on the same node as in the first and second approaches using an NFS4, two nodes are regarded to have already established a connection, and thus the NFS4 makes them share a single connection.

As mentioned above, significant efforts from all DFS implementers or users are needed to utilize a high I/O bandwidth of a single DFS server. Therefore, a new solution is needed to alleviate such efforts.

3.2 Data Management for HPC Storage Systems

3.2.1 Problems of Existing HPC Storage Systems

A BB is introduced for absorbing a bursty I/O in an HPC system. Most supercomputers, including Cori [12] from NERSC, allocate a BB by using a dedicated BB allocation method. The users specify the desired capacity for the applications, and the specified space is provided by an HPC scheduler [21, 22] for the entire lifetime of the application. However, this allocation method causes a severe underutilization of the BB, which is composed of expensive hardware resources, such as high-speed storage media and a high-speed network. In general, the users request more than the actual capacity necessary because the application jobs fail owing to an I/O error when the allocated capacity is insufficient. To avoid a failure, users are recommended by supercomputer providers to request a surplus BB capacity [11]. Users may also require a high capacity to not only avoid a failure but also achieve a higher performance. Because the scheduler determines the number of dedicated BB nodes in proportion to the requested capacity, the users request a larger capacity allowing them to experience a higher performance with more BB nodes and a higher parallelism. Along with the performance scalability, another reason for overabundant requests arises from a complicated data management in multi-tier HPC storage systems (i.e., local storage of a compute node, BB, and DFS). Because current supercomputers manage a BB and PFS separately, the users are challenged with a redundant and complicated management. For example, if users have a limited BB capacity for only one checkpoint, they should copy data manually from the BB to the PFS at every end of the I/O phase to make BB space for the next I/O phase. Furthermore, if the workflow of an application is complicated, manual data movement can be difficult for users to achieve [23].

BB underutilization is also caused by the characteristics of the checkpoint and restart. HPC applications apply a checkpoint with a fixed period [24–26], called a checkpoint period, by repeating the compute and I/O phases periodically. Unfortunately, as the checkpoint period ranges from tens of minutes to tens of hours, expensive BB resources remain idle for long compute phases. Moreover, each application requires a BB capacity larger than the actual checkpoint size to preserve the consistency of the checkpoints. At least twice as much BB capacity is needed for the checkpoints because the old checkpoint should be maintained until the new checkpoint is completely written in a safe manner. HPC users also store multiple versions of a checkpoint in a BB for data durability. Because only the latest version of a checkpoint is needed in case of a failure, the remaining older versions do not actually need to be stored in the BB.

These problems caused by a dedicated BB allocation method have motivated our HPC storage management approach based on an over-subscribing BB.

3.2.2 Limitations with Existing Approaches

Unlike a dedicated BB allocation method, a BB over-subscription method allocates more space to applications than the total capacity of the BB by allowing the applications to be used only during the I/O phase, and not within the whole lifetime. Thus, applications during the computation phase should yield a BB to other applications during the I/O phase through a demotion from the BB to the PFS. Therefore, a data management approach between the BB and PFS is needed. Many approaches have been proposed for a multi-tiered system [11, 13, 27, 28], including an approach between the cache, memory, and storage and an approach for a multi-tier storage of a distributed file system. However, for the following reasons, these approaches are unsuitable for a new

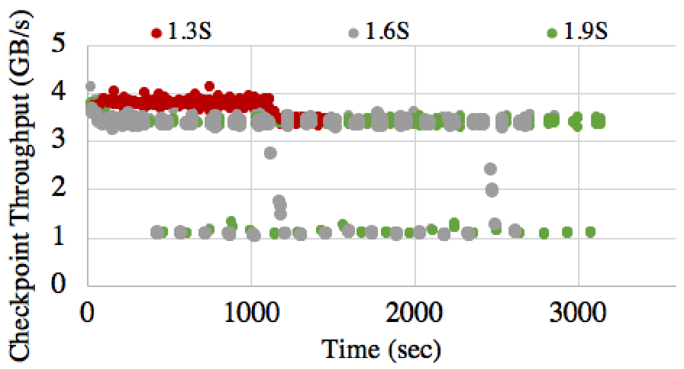


Figure 3.4: Checkpoint performance depends on DWPP

HPC storage system where the checkpoint dominates most of the I/O traffic.

For the first reason, the existing approaches use a static demotion threshold without considering the amount of data to be moved between storage tiers. With prior approaches, a demotion, i.e., the process of copying checkpoint files from the BB to the PFS, is only applied when the BB is idle before reaching the threshold. When the capacity of the BB used reaches the threshold, a demotion is concurrently applied with a checkpoint. If the number of users using the BB increases, the number of checkpoints also increases, resulting in an increased $DWPP$. The increased $DWPP$ causes a decrease in the BB idle time, which reduces the amount of the demotion without interrupting the checkpoint. In particular, because the speed of the data stored in the BB (write B/W of high-end SSDs) may overwhelm the speed of the demotion to the PFS (write B/W of low-end SSDs), the BB fills up when there is an insufficient BB idle time. With the BB filled, applications have to wait until the BB has the available capacity, leading to a significantly low checkpoint performance and a high latency. Figure 3.4 shows the checkpoint performance with different $DWPP$s after setting the same demotion threshold to 90% of the total BB capacity. Because S indicates the capacity of the BB, 1.3 S, 1.6 S, and 1.9 S writes are 1.3-, 1.6-, and

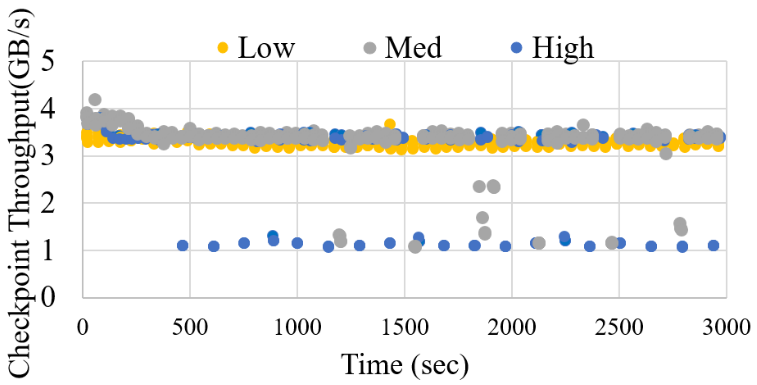


Figure 3.5: Checkpoint performance depends on I/O jobs arrival pattern (I/O job congestion)

1.9-times the size of the BB during a certain period, respectively. With 1.3 S, a slightly lower performance is demonstrated after 1000 s. However, with 1.6 S, the BB occasionally becomes full during the middle of an I/O job for the checkpoint, and thus the performance begins to decrease over time. For 1.9 S, almost half of the applications achieve a four-fold lower performance because they have to be stopped or apply a checkpoint with a demotion to increase the available BB capacity.

As another reason, the existing approaches do not consider the distribution of I/O jobs for a checkpoint. Specifically, even with the same $DWPP$, I/O jobs of applications for checkpoint arrive in groups or evenly within a certain period. When the I/O jobs arrive evenly, a BB idling time occurs between job arrivals. Thus, the files are demoted during a BB idling time, making sufficient space in the BB for the next I/O jobs. However, if the I/O jobs arrive in crowds, a lack of a BB idling time between I/O jobs causes little demotion, which leads to a capacity depletion of the BB. As shown in Figure 3.5, the checkpoint performance is highly related to the I/O job congestion under the same $DWPP$. There are three I/O job congestion patterns (low, medium, and high) with 1.9 S, which represent the rate of crowding of arriving I/O jobs. A low congestion always shows a higher performance because there is a sufficient idling time between I/O jobs. By contrast, when the I/O jobs arrive in crowds with medium and high congestion, it results in a low checkpoint performance.

Finally, the existing approaches identify hot and cold checkpoint files using basic algorithms based on the data locality, such as FIFO, LRU, and Hotness-aware. (Hot checkpoint files are left in the BB, whereas cold checkpoint files stay in the PFS.) However, as mentioned in section 2.5, because checkpoint files across applications do not have data locality, it is inappropriate to apply basic algorithms for selecting cold checkpoint files. HPC applications have their own checkpoint period and keep multiple versions of the checkpoints. With the FIFO algorithm, although old versions of the checkpoints for an application with a low checkpoint period are stored in the BB, the latest checkpoint with high checkpoint period can be chosen as the cold data. This lowers the efficiency of the BB and leads to a low restart performance. In addition, a checkpoint file is needed only in the case of a failure. Because a new checkpoint file is created during every checkpoint period, it is very unlikely that the same file will be reused multiple times. Therefore, an LRU or Hotness-aware algorithm leads to a low efficiency of the BB. Moreover, because there is no spatial locality between the checkpoint files, checkpoint files near a failed file do not need to be prefetched or left in the BB. The failure rates also need to be considered because HPC applications have their own failure rates. Without considering the failure rates, checkpoint files with a high failure rate might be chosen as cold data, and not as checkpoint files with a low failure rate.

Chapter 4

Mulconn: User-Transparent I/O Transfer Management for HPC Storage Systems

4.1 Design and Architecture

4.1.1 Overview

To enable users of new all-flash HPC storage systems to utilize a high I/O bandwidth from a single BB server node, we propose a new I/O transfer management scheme, called *Mulconn*, as shown in Figure 4.1. By developing *Mulconn* in a VFS, which is an abstract layer that all file systems depend on, we completely eliminate the effort required by users to distribute I/O requests to multiple connections, and reduce the effort required by developers to modify complicated DFS sources for multiple connections. To provide multiple connections, we modify a new mounting procedure and I/O procedures in a client-side VFS and utilize mount binding in a server-side VFS. However, in the case of DFSs that provide a single connection between nodes, as mentioned in section 3.1.2,

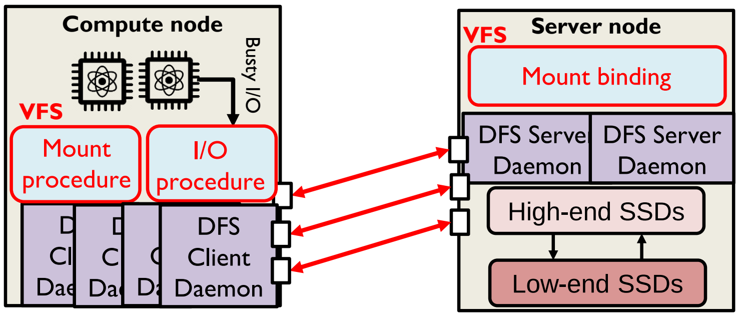
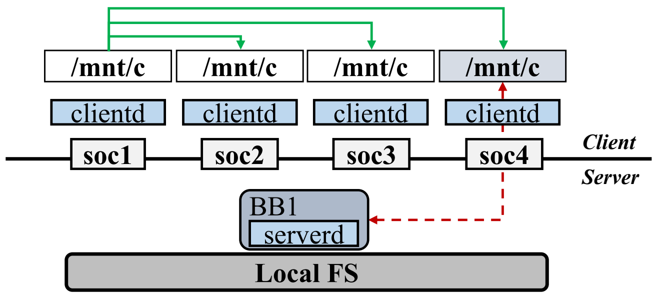


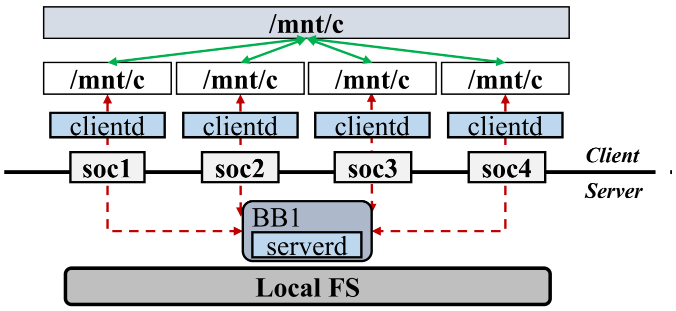
Figure 4.1: Overview of Mulconn

only a simple code modification is required to create a new socket even if a connection exists between two nodes when a new daemon is created.

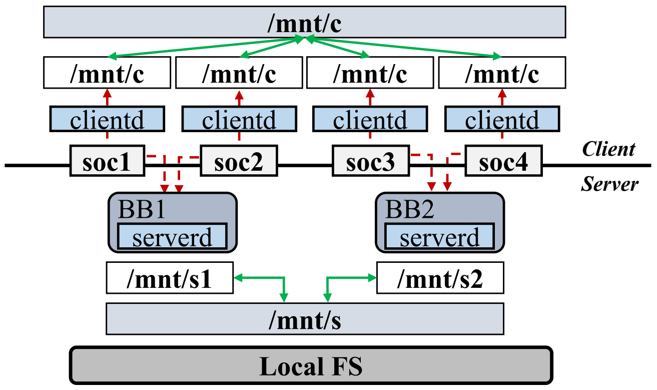
In this section, we present the design aspects of our *Mulconn*.



(a) Mount procedure with original VFS



(b) Mount procedure of *Mulconn* with multiple client daemons



(c) Mount procedure of *Mulconn* with multiple client and server daemons

Figure 4.2: Mount procedures with and without *Mulconn*

4.1.2 Scale Up Connections

A reliable and simple way to provide multiple connections while minimizing the modification of DFSs is making multiple client and server daemons by using the mount command. However, as mentioned earlier, significant effort is required by users to make the best of multiple connections. To handle this problem, we adhere to the method of creating connections through the mount command, but implement a new mount procedure in a VFS such that the users do not need to manage multiple mount points.

We make a new mount command, as shown below, allowing DFS users to easily achieve multiple connections. *"mount -mc n -t fstype -md BB_1,...,BB_n dir"* With an *mc* option, we receive the number of connections desired by the users and internally conduct a mount procedure several times to make the specified number of connections. In addition, we need to expose one mount point specified by the clients. However, users cannot use all connections with only a single mount point with the original VFS. If the mount command is requested multiple times on the same mount point, I/O requests to the mount point are processed through the recent BB owing to the vertical mount path hierarchy, as shown in Figure 4.2a. If a user executes the command "$mount\ -t\ glusterfs\ BB1\ /mnt/c$", four times, I/O requests to the $/mnt/c$ are transmitted only through the fourth connected socket despite the presence of four connections from the client node to the $BB1$.

For all connections using only one mount point, we change the vertical hierarchy of the mount path to a horizontal hierarchy. Figure 4.2b shows our system with a horizontal hierarchy. Unlike a traditional procedure, I/O requests to $/mnt/client$ are delivered using any socket.

As mentioned in section 3.1.2, users can increase the number of connections

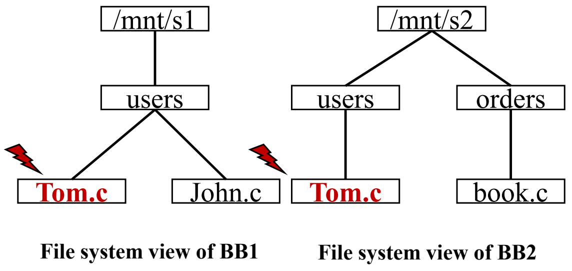


Figure 4.3: File system view of BBs with original VFS

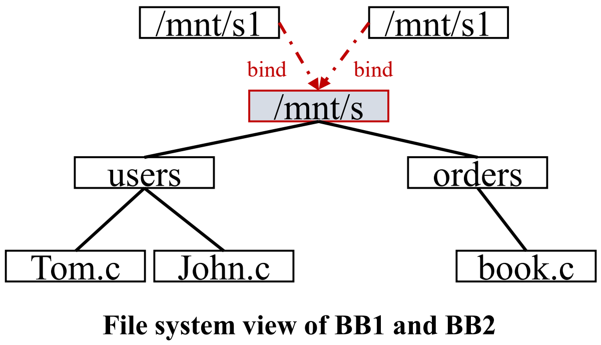


Figure 4.4: File system view of BBs with our *Mulconn*

by making DFSs (a BB in new HPC storage systems) on the same server node. However, the users must apply a file I/O by distinguishing which DFS each mount point is connected to. To do so, we add an *md* option in our mount command. Users specify the names of the DFS they use through this option. If "$mount\ -mc\ 4\ -t\ glusterfs\ -md\ BB1, BB2\ /mnt/c$" is executed by the user, we connect two of the four mount points to DFS1 and the other two to DFS2, as in Figure 4.2c. However, data conflicts may occur because two DFSs have different file system trees, as shown in Figure 4.3. When the user writes to a file called Tom.c, a mount point connected to BB1 is used and the file is stored in DFS1. However, when write requests are requests to the Tom.c file again, a mount point for BB2 is used, and thus Tom.c is also created in BB2. That is, the same file exists in multiple locations, resulting in a data conflict. For this reason, we bind the two mount points of BB1 and BB2 to the same directory of a local file system in the server node, and thus they have the same file system view, as indicated in Figure 4.4. With this method, regardless of which connection the user applies, the same file does not exist on multiple DFSs.

4.1.3 I/O Scheduling

To distribute I/O requests to multiple connections well (evenly), we propose two I/O scheduling polices. Figures 4.5 and 4.6 demonstrate our two polices, *TtoS* and *TtoM*.

TtoS

Figure 4.5 shows a thread-to-single connection (*TtoS*) policy, which is the simplest way to utilize multiple connections. The policy aims to allocate one connection among the multiple connections to each thread requesting I/O requests

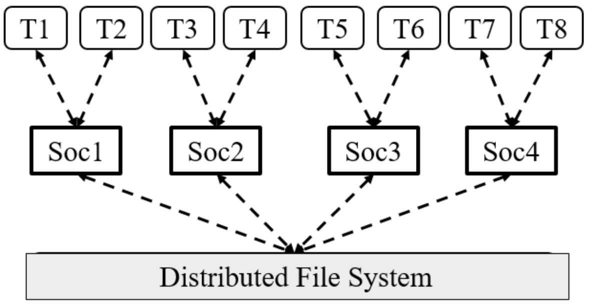


Figure 4.5: TtoS policy

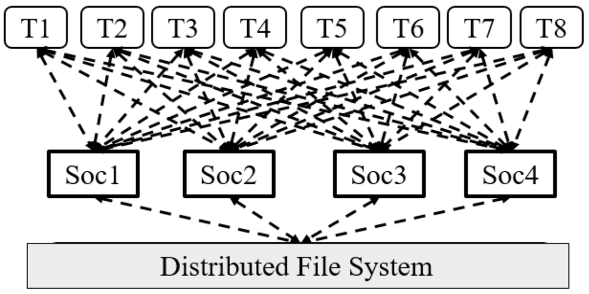


Figure 4.6: TtoM

from a DFS client node. Because each thread uses only one connection, the contention for the socket is relatively small. However, if the number of threads is smaller than the number of connections, there may be idle connections, and thus a high I/O bandwidth from the DFS server may not be utilized.

TtoM

To overcome the limitation of the *TtoS* policy, we propose a thread-to-multiple connections (*TtoM*) policy, which gives threads access to all sockets, as shown in Figure 4.6. To transfer I/O requests using all sockets, we open the same file over all connections. The files opened based on the number of connections are invisible to the users, and only one fd is provided, as if the user has only one file. When I/O requests of the file are issued, they are delivered in order. All connections are for the same DFS server node on the same client node, but the DFS server considers each connection to be a different client. Therefore, the DFS server asks a local file system (LFS) to open the same file the same number of times with the number of connections, resulting in a lock contention when I/O requests for the same file through different connections arrive at the LFS. In addition, because the sockets are accessed by all threads, the contention for the socket is much larger than that of $TtoS$.

4.1.4 Automatic Policy Decision

In addition to providing the two I/O scheduling policies, $TtoS$ and $TtoM$, $Mulconn$ also proposes an automatic policy decision for users who have difficulty choosing I/O policies. Based on the experimental results, we use $TtoS$ for write processing and choose $TtoM$ for read processing with an automatic policy decision.

Algorithm 4.1: File Open and File Close for *TtoS*

```
do_sys_open(path)
{
    int fd = alloc_fd();
    struct *file = do_filp_open(path);
    __fd_install(fd, file);
}
do_filp_open(path)
{
    struct file = alloc_empty_file();
    __follow_mount_rcu(file, path);
}
__follow_mount_rcu(file, path)
{
    int conn_num = thread_id % total_num_conn;
    __lookup_mount(path, file->mnt, conn_num);
}
__lookup_mount(path, file, conn_num) {
    file->mnt_path = file->mnt->mnt_array[soc_num];
}
__fd_install(fd, file)
    struct fdtable *fdt = thread->fdt;
    fdt->fd[fd] = file;
}
__close_fd(fd) {
    struct *file = thread->fdt->fd[fd];
    flip_close(file);
}
```

Algorithm 4.2: File Open and File Close for *TtoM*

```
do_sys_open(path)
{
    int fd = alloc_fd();
    for (int f_num = 0; f_num < conn_num; f_num++)
    {
        struct *file = do_filp_open(path,f_num);
        __fd_install(fd, file, f_num);
    }
}
do_filp_open(path, f_num)
{
    struct file = alloc_empty_file();
    __follow_mount_rcu(file, path, f_num);
}
__follow_mount_rcu(file, path, f_num)
{
    __lookup_mount(path, file->mnt, f_num);
}
__lookup_mount(path, file, f_num) {
    file->mnt_path = file->mnt->mnt_array[f_num];
}
__fd_install(fd, file, f_num)
    struct fdtable *fdt = thread->fdt;
    fdt->fd_array[f_num]->fd[fd] = file;
}
__close_fd(fd) {
    for (int f_num = 0; f_num < conn_num; f_num++)
    {
        struct *file = thread->fdt->fd_array[f_num]->fd[fd];
        flip_close(file);
    }
}
```

4.2 Implementation

In this section, we demonstrate the implementation of our system in a VFS using I/O processing.

4.2.1 File Open and Close

A file open request is applied first for file write/read requests. While processing the file open request, the connection that will handle the request is determined. With $follow\_mount\_rcu()$, we select one of the connections in the horizontal mount path hierarchy depending on the policy proposed and open a file through the connection.

Algorithm 4.1 shows the file open and file close procedures of $TtoS$. If a file open request is presented, an fd required for the I/O requests is assigned by the user through *alloc_fd()* (Line 3). Then, *do_filp_open()* is called to open the file of the *path* specified by the user (Line 4). In *do_filp_open()*, an empty file is first allocated and *__follow_mount_rcu()* is then called with the file to determine which mount path to use to send the requests to that file (Line 10). With the $TtoS$ policy, we choose the connection based on the thread ID (Line 14). Most applications that use DFSs run file I/O tasks simultaneously using multiple threads, and thus they have a series of thread IDs. Hence, the users make the best use of all connections. In addition, we call *__lookup_mount()* to specify the mount path to the file with the determined connection number (Line 15). With *__lookup_mount()*, we find the mount path that matches the number of connections in the horizontal mount hierarchy we created and insert the mount path to the file (Line 18). Next, *__fd_install()* is called with the prepared fd and file (Line 5). Because $TtoS$ uses a single connection per thread, an fd array of an fdtable stores the file according to the fd (Lines 21 and 22). If a file close request is required from the user, only one file is to be closed in the *TtoS* policy (Lines 24 and 25).

With the *TtoM* policy, we open the same file as many times as the number of connections through all sockets, as shown in Algorithm 4.2. Therefore, the file structure is allocated and *__fd_install()* is called based on the number of connections (Line 4 and 5). Unlike the traditional relationship between the fd and file, because the user actually has multiple files with one fd, we modify *struct fdtable*. In *fdtable struct*, there is an array that maps the opened file with the fd in the original kernel. We create an array for each connection, and map the files per connection to their own array, such as $thread \rightarrow fdt \rightarrow fd\_array[]$ in Lines 19 and 20. With multiple fd arrays, the users are available to access all connections with one fd.

In the case of a file close request, all open files are closed in all connections in the *TtoM* policy by finding the files from the fd_array of fdtable with only one fd (Lines 26–32).

Algorithm 4.3: File Write and File Read for *TtoS*

```
ksys_write(fd) or ksys_read(fd) or io_submit_one(fd)
{
    struct *file = __fget(fd);
    write (file, data); or read(file, data);
}
__fget(fd)
{
    struct *file = __fcheck_files(fd);
}
__fcheck_files(fd) {
    struct fdtable *fdt = thread->fdt;
    return fdt->fd[fd];
}
```

Algorithm 4.4: File Write and File Read for *TtoM*

```
ksys_write(fd) or ksys_read(fd) or io_submit_one(fd)
{
    struct *file = __fget(fd);
    write (file, data); or read(file, data);
}
__fget(fd)
{     struct *file = __fcheck_files(fd);
    file->access_cnt++;
    int conn_num = (file->access_cnt % total_num_conn;
    file = __fcheck_files_TtoM(fd, conn_num);
}
__fcheck_files(fd) {
    struct fdtable *fdt = thread->fdt;
    return fdt->fd[fd];
}
__fcheck_files_TtoM(fd, conn_num) {
    struct fdtable *fdt = thread->fdt->fd_array[conn_num];
    return fdt->fd[fd];
}
```

4.2.2 File Write and Read

After the file open operation, write and read I/O requests of the file arrive. Regardless of the synchronous and asynchronous I/O models applied, whenever write and read I/O requests are issued, the file created during the open operation is first brought in $\_\_fcheck\_files()$.

Algorithm 4.3 shows the file write and file read procedures with the *TtoS* policy. First, we execute a *__fget()* called $\_\_fcheck\_files()$ to find the file for the I/O requests (Lines 3-7). In $\_\_fcheck\_files()$, because only one file is mapped to one fd of the fd array, the procedure is the same as that of the original kernel.

With the *TtoM* policy 4.4, we first receive a file through a $\_\_fcheck\_files()$ as with the *TtoS* procedure. However, there are multiple fd arrays for one fd, and thus we select the fd array for each I/O request to use multiple connections evenly. To do so, we add a variable called *access_cnt* to $file\ struct$ and increase the value of *access_count* each time the I/O requests a file to arrive (Line 8). By calling $\_\_fcheck\_files\_TtoM()$, which we developed using the variable, fd arrays can be used in sequence (Lines 16-19). Therefore, *TtoM* creates I/O requests for a file that transfers through all connections.

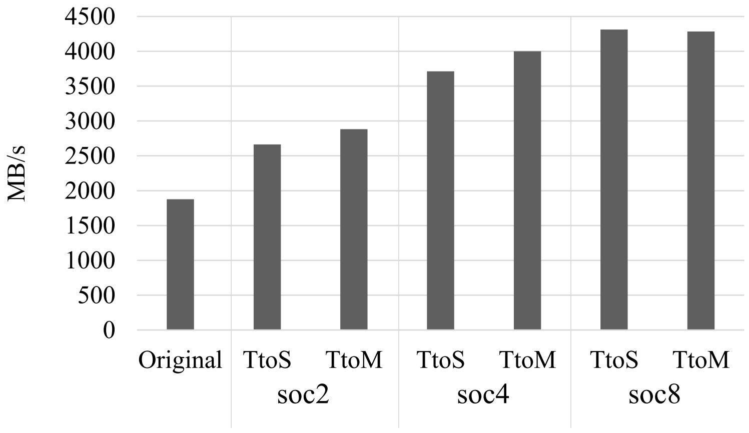
4.3 Evaluation

4.3.1 Experimental Environment

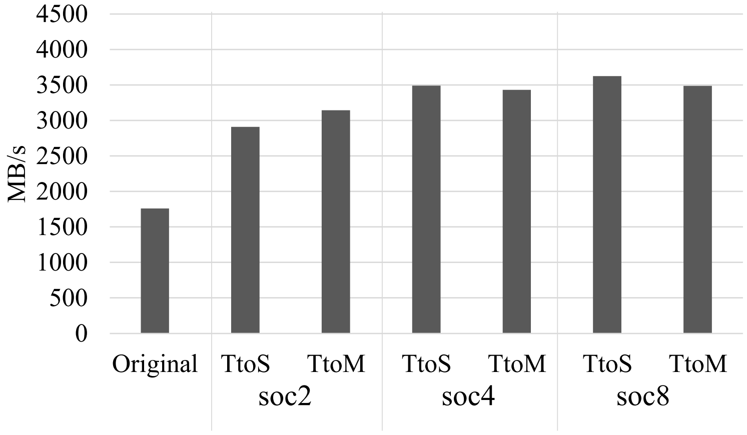
We evaluate our I/O subsystem, *Mulconn*, using a single client node and a sever node. To demonstrate the effectiveness of the I/O subsystem, we executed experiments under two different environments. In the first environment, we use two nodes for the client and server consisting of an Intel(R) Xeon(R) Silver 4216 processor and three 2-TB FADU NVMe SSDs provided by a semiconductor start-up company. Each SSD has an I/O bandwidth of up to 1700 and 3200 MB/s for a sequential write and read, respectively. In the second environment, we have two nodes with an AMD EPYC 7301 16-core processor for the client and server nodes. In addition, we adopt five Intel DC P4500 (1 TB) SSDs, each of which shows a sequential write and read bandwidth of up to 600 and 3200 MB/s, respectively. In both environments, the client and server nodes are connected with a 100 GbE Mellanox SN2100 Switch. We use GlusterFS version 6.5 for a parallel file system and tune the configurations of the GlusterFS for a high performance, but without making any modifications. Ext4 is used as the local file system for GlusterFS. In addition, we add the designed I/O subsystem in a VFS of kernel version 5.3.12. For the experiments, we use a micro benchmark FIO. To validate our system based on two policies, we compare sequential write and read I/O throughputs provided by the PFS of our system with those of the original kernel version 5.3.12.

4.3.2 I/O Throughputs Improvement

In this section, write and read I/O throughputs with synchronous and asynchronous I/O mode are measured by increasing the number of connections with multiple client daemons and server daemons through our system.

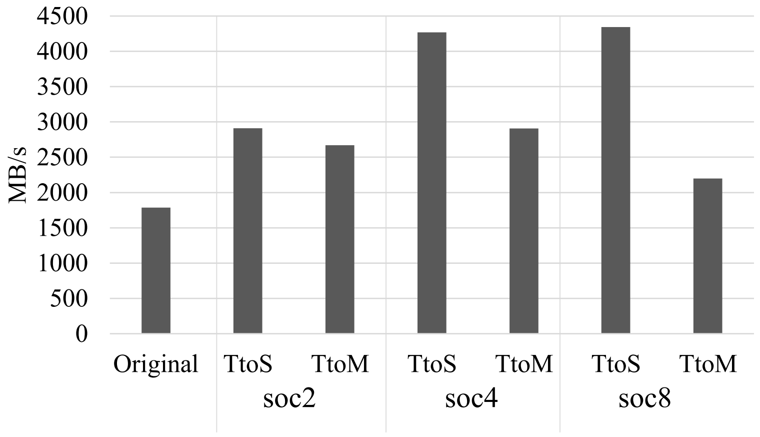


(a) Sync Write I/O throughputs

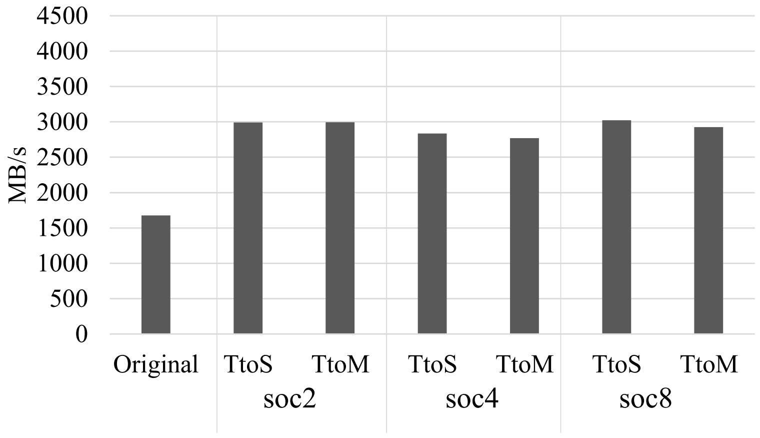


(b) Sync read I/O throughputs

Figure 4.7: I/O throughputs with synchronous I/O mode under various the number of connections



(a) Async write I/O throughputs



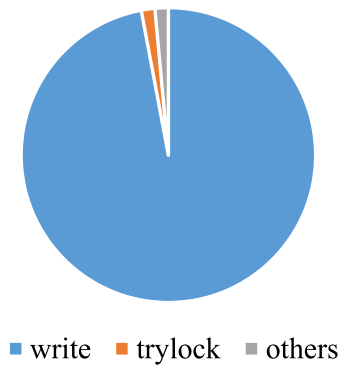
(b) Async read I/O throughputs

Figure 4.8: I/O throughputs with asynchronous I/O mode under various the number of connections

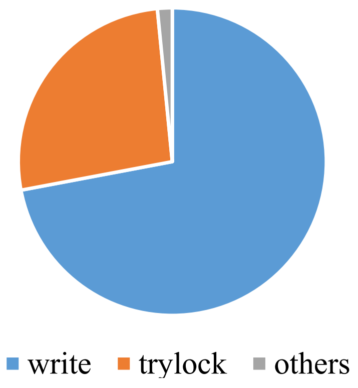
Scale Up Connections with Client Daemons

Figure 4.7 shows synchronous write and read I/O throughputs with GlusterFS under the first environment. Here, *Original* indicates the result with the original kernel, and *TtoS* and *TtoM* show the results with the optimized kernel including our I/O subsystem. For this experiment, we use 16 threads to execute I/O operations on each file. In the study, *Original* shows write and read I/O throughputs of approximately 1800 and 1700 MB/s, respectively, because there is only one connection between the client and server nodes. Unlike *Original*, *TtoS* and *TtoM* have two to eight connections by increasing the number of client daemons. Both *TtoS* and *TtoM* show a higher write I/O throughput as the number of connections increases and up to 4300 MB/s with eight connections. The read I/O throughputs also improve as the number of connections increases, although the performance improvement stops at approximately 3500 MB/s. As shown in the figure, *TtoS* and *TtoM* have a similar performance. This is because threads of *TtoM* cannot send I/O requests to multiple connections simultaneously but do send a single request to one of the connections and wait until the server receives it in synchronous mode.

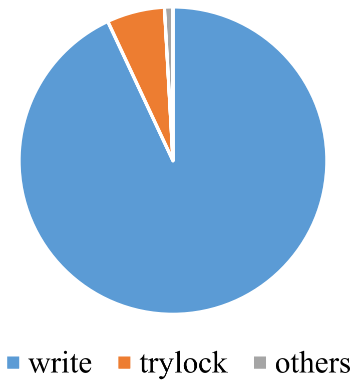
To achieve a higher I/O bandwidth, we utilize asynchronous I/O mode to submit I/O requests simultaneously. Despite using asynchronous I/O mode, *Original* shows a similar or lower write throughput compared with those from synchronous mode, as shown in Figure 4.8a. This is because only the maximum bandwidth obtained from a single connection is provided to the client, regardless of how many threads it uses to send requests asynchronously, and network contention occurs among multiple threads. In contrast to similar amounts of change in write throughput from *TtoS* and *TtoM* depending on the number of connections in synchronous I/O mode, they show different patterns of throughput change in asynchronous I/O mode. Here, *TtoS* improves the write throughput with the number of connections and shows higher throughputs than using synchronous I/O mode. However, in the case of *TtoM*, the write throughput does not increase after four connections, and the performance even decreases with eight connections. Although the same numbers of connections are formed, the two policies provide different write throughputs because of the different ways the I/O threads use the connections, as mentioned in section 4.1.3. In addition, *TtoS* allows each thread to use only one connection, and thus each file is opened only once in the PFS server and local file system. By contrast, *TtoM* allows each thread to use all connections, and therefore the file is opened as many times as the number of connections. There is actually one user thread that writes to the file, although because file open operations are performed through different PFS client daemons, each of which has a connection, multiple threads are considered to request an open operation on the same file in the PFS server. Thus, the PFS server daemon requests as many file operations as the number of connections to the local file system. For this reason, multiple threads access the same file in the local file system, Ext4, resulting in a lock contention.



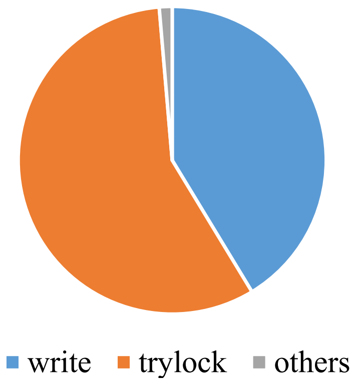
(a) TtoS2



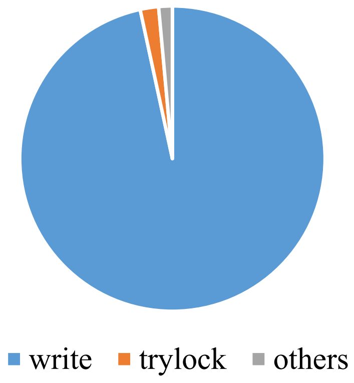
(b) TtoM2



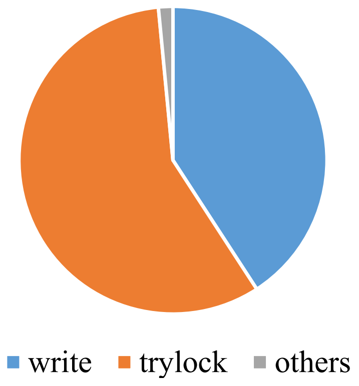
(c) TtoS4



(d) TtoM4



(e) TtoS8



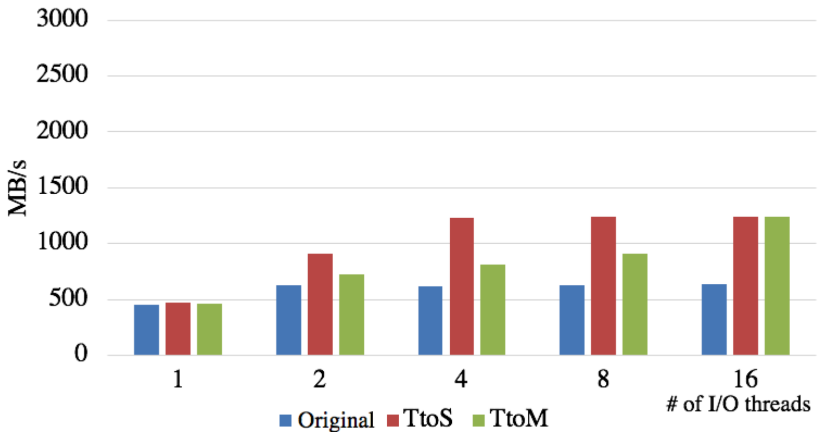
(f) TtoM8

Figure 4.9: Lock contention analysis for write operations with *TtoS* and *TtoM*

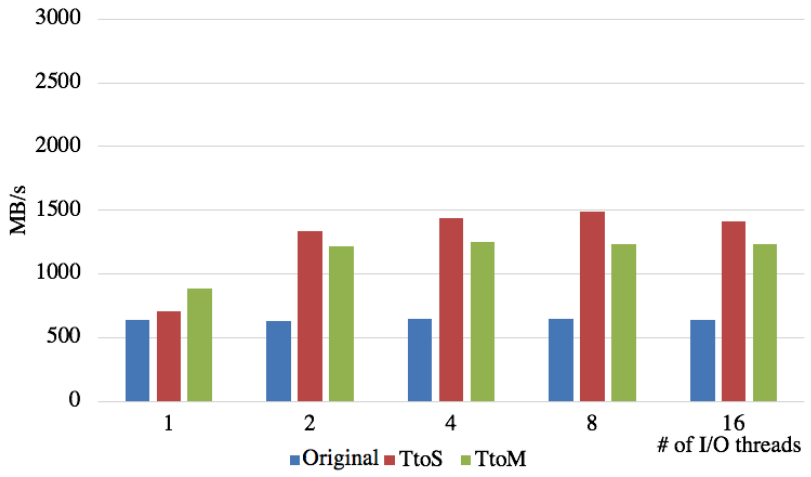
To analyze the lock contention in $TtoM$, we profile a function that performs a write operation in Ext4 ($ext4\_file\_write\_iter()$). The function is divided into three parts, and we measure the time spent in each part with two, four, and eight connections. One of the parts is $inode\_trylock()$ used to guarantee data consistency by preventing access from multiple threads. Another is $\_\_generic\_file\_write\_iter()$ used to write data to the storage devices. The final part includes the remaining functions not used in the other two parts. Figure 4.9 shows the ratio of the execution time in each part to the total execution time of $ext4\_file\_write\_iter()$. Regardless of the number of connections, $TtoS$ spends most of its time executing the function for an actual write to the devices. By contrast, in the case of $TtoM$, the execution time required to attempt to obtain the lock is greater than the time required to write, except for two connections. With these two connections, both $TtoS$ and $TtoM$ show almost similar write I/O throughputs because the time for a *trylock* is less than that for a *write*. When there are four and eight connections, $TtoS$ and $TtoM$ have a similar time to write, but $TtoM$ spends almost 11- and 50-times more time obtaining the lock than $TtoS$ for four and eight connections, respectively. In other words, the lock contention is severe in $TtoM$ because the multiple threads try to perform a write operation on the same file, and thus the write I/O throughput is lower than that of $TtoS$. As the number of connections increases, the number of threads accessing the same file increases, resulting in a more serious lock contention, which allows $TtoM$ to provide a lower I/O write throughput.

Fortunately, $TtoM$ shows unchanged read I/O throughputs after two connections in asynchronous I/O mode, as shown in Figure 4.8b. This is because there is no lock for the read operations in $ext4\_file\_read\_iter()$. However, both $TtoM$ and $TtoS$ show lower read I/O throughputs in asynchronous I/O mode than those in synchronous I/O mode. A decrease in performance may occur in the software overhead of GlusterFS or FUSE owing to the many requests being submitted concurrently. A decreased performance does not occur during an asynchronous write I/O execution because the write operation is processed more slowly on a device than a read operation. This problem is out of the scope of our system, however, and thus we do not address it herein.

Figure 4.10 and 4.11 shows write I/O throughputs with two and four connections under the second environment. In this environment, original kernel, which provides only single connection, shows around 500MB/s regardless of the number of I/O threads and the number of client daemons. On the other hand, Mulconn performs much better. As shown in the figure 4.10, TtoS shows higher write I/O throughputs than TtoM because there is no lock contention. By creating two cilent daemons, two connections are used, and up to 1500MB/s write I/O throughput is achieved. As with the figure 4.11, the number of daemons is increased from 2 to 4, and the write I/O throughputs doubled. We shows 6-times performance improvement over the Original kernel with just four connections.

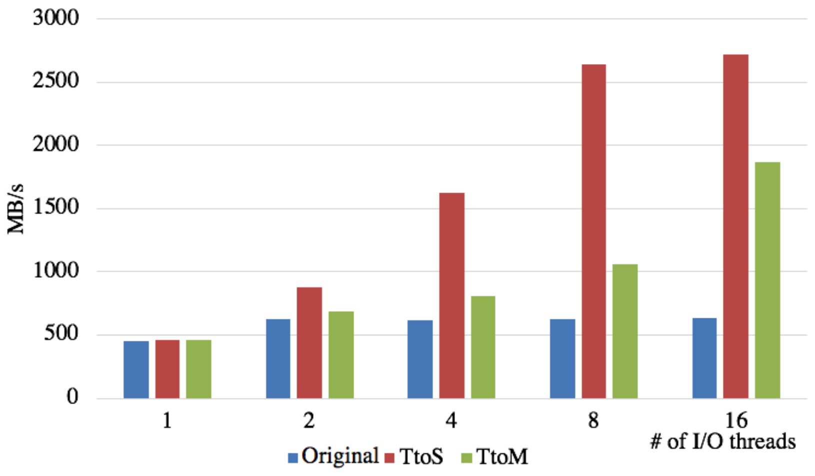


(a) Sync write I/O throughputs

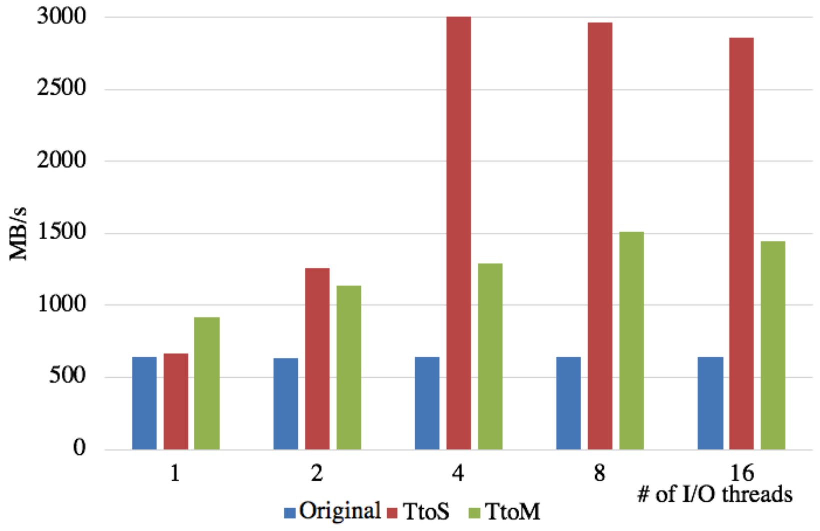


(b) Async write I/O throughputs

Figure 4.10: Write throughputs with two connections(client daemons) under various the number of I/O threads

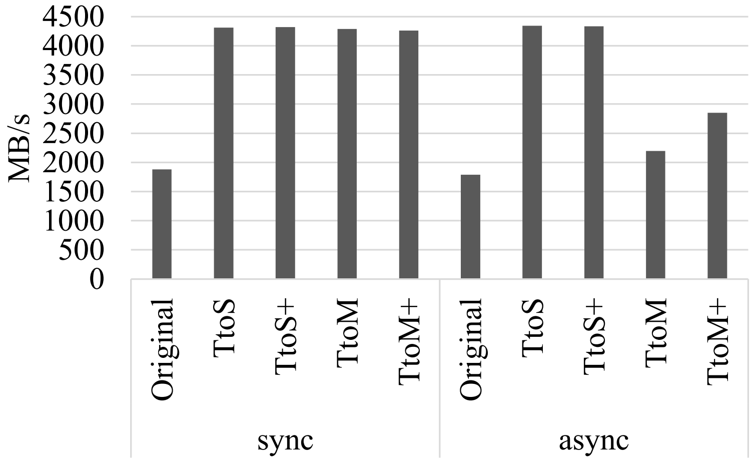


(a) Sync write I/O throughputs

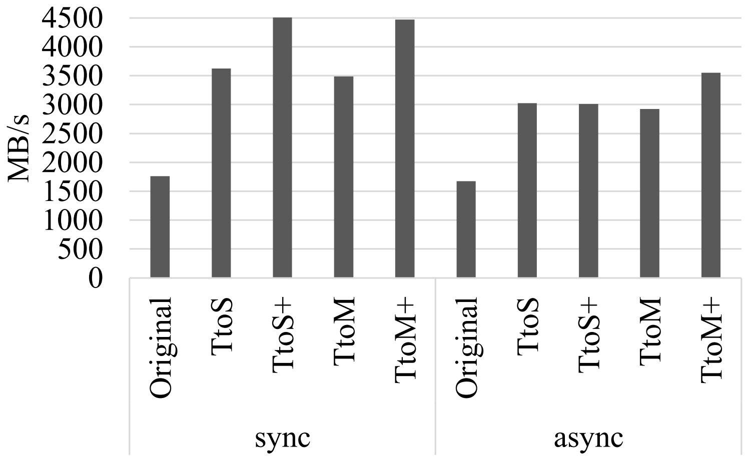


(b) Async write I/O throughputs

Figure 4.11: Write throughputs with four connections(client daemons) under various the number of I/O threads



(a) Write I/O throughputs with multiple server daemons



(b) Read I/O throughputs with multiple server daemons

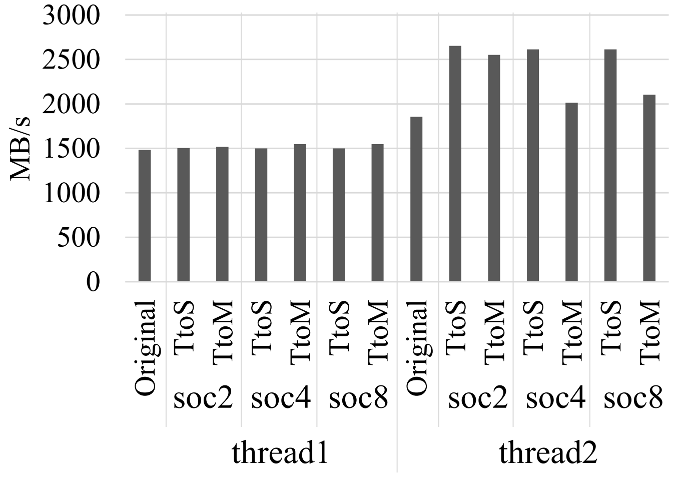
Figure 4.12: I/O throughputs with multiple connection by increasing the number of server daemons

Scale Up Connections with Server Daemons

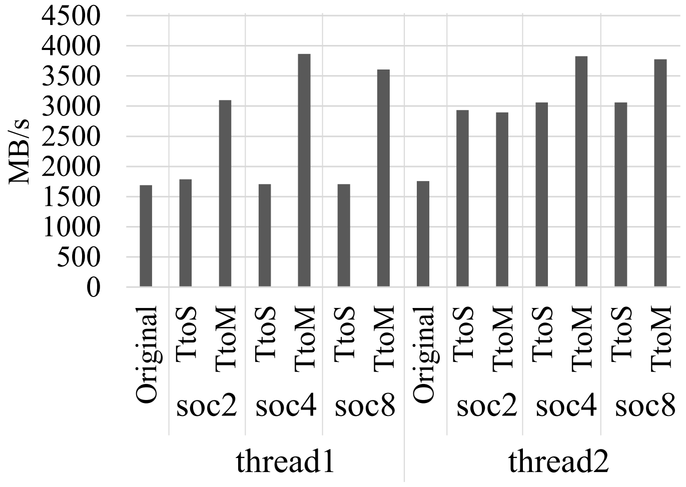
Despite increasing the number of connections to eight through the creation of client daemons, the write and read I/O throughputs are no longer improved with the *TtoS* and *TtoM* policies. Because of the increased number of connections, the single server daemon of GlutsterFS has difficulty processing the increased I/O requests, and thus the server daemon reaches a performance bottleneck. To handle this performance bottleneck, we add new connections by creating a new server daemon as mentioned in Section 4.1.2. We call *TtoS* and *TtoM* using connections made by creating server daemons as *TtoS+* and *TtoM+*.

We use eight connections between the client and server nodes and two server daemons, as shown in Figure 4.12. In *TtoS+* and *TtoM+*, we have four connections to the first server daemon and the other four connections to the second daemon. For executions of write operations, *TtoM+* only performs better than *TtoM* in asynchronous I/O mode. In other experimental write results, the scale-up connections with server daemons have no effect because there is an insufficient number of requests for a single server daemon to reach a performance bottleneck.

As shown in Figure 4.12b, *TtoS+* and *TtoM+* show higher read I/O throughputs than *TtoS* and *TtoM* with synchronous I/O mode. As mentioned before, read throughputs are low in asynchronous I/O mode owing to the numerous requests submitted at the same time. We expected that increasing the number of server daemons would improve the read throughputs; however, *TtoM+* shows only a 600 MB/s higher async read throughput than *TtoM*.



(a) Async Write I/O throughputs with fewer threads than the number of connections



(b) Async read I/O throughputs with fewer threads than the number of connections

Figure 4.13: I/O throughputs with fewer threads for I/O job than the number of connections

4.3.3 Comparison between *TtoS* and *TtoM*

In section 4.3.2, because we use a maximum of 8 connections and 16 threads for an I/O, all connections are utilized in both *TtoS* and *TtoM*. However, if the number of threads is smaller than the number of connections, idle connections will occur with *TtoS*. Because *TtoS* allows each thread to use only one connection, it provides a low throughput despite being able to achieve a higher throughput with more connections when the number of threads is less than the number of connections. By contrast, *TtoM* allows each thread to access all connections, and thus all connections are utilized regardless of the number of threads. Regarding the efficiency of the two policies, we have one or two threads for I/O jobs and two, four, and eight connections for the experiments shown in Figure 4.13. In synchronous I/O mode, the number of connections used by each thread does not affect the I/O throughput because the threads have to wait for a return after submitting an I/O request. Therefore, we only measure the I/O throughputs in asynchronous I/O mode.

Figure 4.13a shows the write I/O throughputs with one and two threads using two to eight connections. With one thread for the I/O, *Original*, *TtoS*, and *TtoM* show similar write I/O throughputs. The throughputs of *TtoS* and *Original* are the same because both utilize a single connection owing to the use of one thread. All connections are used by one thread under *TtoM*, but it has the same throughput as the others because of the overhead caused by a lock contention. In the experiments with two threads, *TtoS* improves the write throughput, whereas *TtoM* shows a lower throughput than *TtoS* owing to the increased lock contention.

There is no lock contention in the read operation, resulting in higher read throughputs of *TtoM* than those of *TtoS* in Figure 4.13b. When read I/O

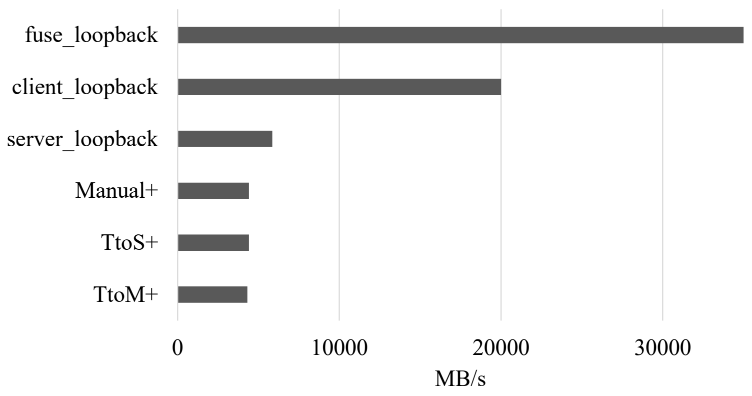


Figure 4.14: Comparision of I/O throughputs of our system and maximum available I/O thorughputs

requests are submitted with one thread, $TtoS$ uses only one connection, whereas $TtoM$ sends multiple requests to all connections simultaneously, showing up to three-times more read throughputs. When two threads are executed, $TtoS$ utilizes two connections, and thus the throughputs are improved compared to those with a single thread. However, as the number of connections increases to four to eight, $TtoM$ shows a higher throughput than $TtoS$.

When the number of threads for the I/O is smaller than the number of connections, $TtoM$ has little effect on the write I/O throughput but leads to a significantly improved write throughput.

4.3.4 Effectiveness of Our System

To verify how well our system utilizes the I/O bandwidth provided by a single server, we compare the I/O throughput of $TtoS+$ and $TtoM+$ with the throughputs of the method mentioned in section 3.1.2, which requires effort by DFS users. We refer to this method as $Manual+$ in this section. In addition, we conducted loopback experiments to emphasize the effectiveness of our system.

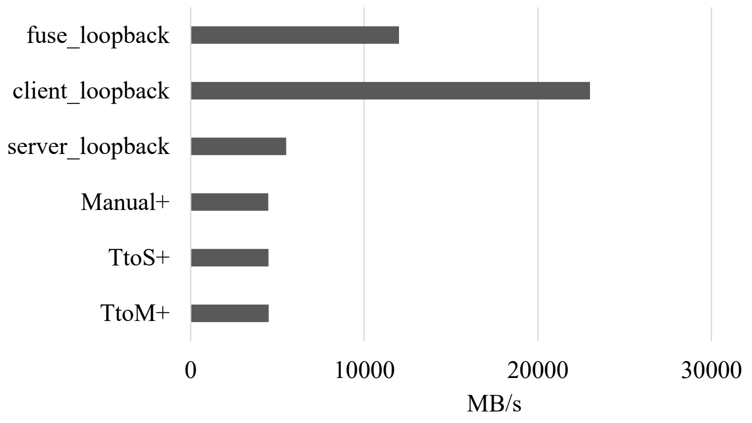


Figure 4.15: Comparision of I/O throughputs of our system and maximum available I/O thorughputs

For the experiments, we use eight connections with two server daemons, and thus $TtoS+$, $TtoM+$, and $Manual+$ have eight mount points for the server daemons. To allow $Manual+$ to easily distribute I/O requests to the eight connections, we allow each thread to access one connection such as $TtoS+$. Figures4.14 and 4.15 show write and read I/O throughputs with 16 threads in synchronous I/O mode. The write I/O throughputs are over 20 GB/s when write I/O requests go through the fuse and client stacks. However, as the I/O requests pass the server stack, which returns the requests as soon as the server daemon receives them, the write throughputs decrease to 5859 MB/s. We make connections and distribute I/O requests to eight mount points manually; $Manual+$ has a similar write throughput as our approach, which means there is little overhead in our system, although we allow the DFS users to utilize multiple connections without much effort. In addition, our $TtoS+$ and $TtoM+$ show a throughput close to that of $server_loopback$ even when writing to real devices.

For the read executions, a fuse or stack of GlusterFS for FUSE has difficulty processing many read requests simultaneously, and thus the write throughput is improved by making the I/O requests go through the client stack to keep them from getting crowded at the fuse and the stack for FUSE. Similar to the throughputs for a write I/O, the read throughputs for the *server_loopback* and *Manual+* decrease to 5500 and 4471 MB/s, respectively. Predictably, our *TtoS+* and *TtoM+* provide the same performance as *Manual+*.

Based on these results, we prove that our system helps the PFS users to obtain the maximum I/O throughput in the simplest way provided by this environment and GlusterFS.

4.4 Summary

Herein, we proposed BBOS for use in a new all-flash HPC storage system. Specifically, we over-subscribed the BB by only allocating it during I/O phases, and not during the entire lifetime for a higher BB utilization. To mitigate a performance reduction caused by an over-subscription, we provided the I/O scheduler and data management module. The I/O scheduler resolved the 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 a checkpoint/restart. Based on these characteristics, we transferred data from the BB to the PFS transparently by adjusting the thresholds and speed of the demotion according to the $DWPP$. We also identified cold data by considering different versions and failure rates.As a result, we improved the BB utilization by at least 2.2-times that of the dedicated BB allocation method. In addition, we guarantee a higher checkpoint throughput without a sudden performance reduction and handle 96.4% of restart requests in the BB, providing up to a 3.1-times higher restart performance than that of other approaches.

Chapter 5

BBOS: User-Transparent Data Management for HPC Storage Systems

5.1 Design and Architecture

5.1.1 Overview

Figure 5.1 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.

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 [29]. In addition, interleaved data from the multiple I/O jobs is saved in the same SSD block, which causes garbage collection

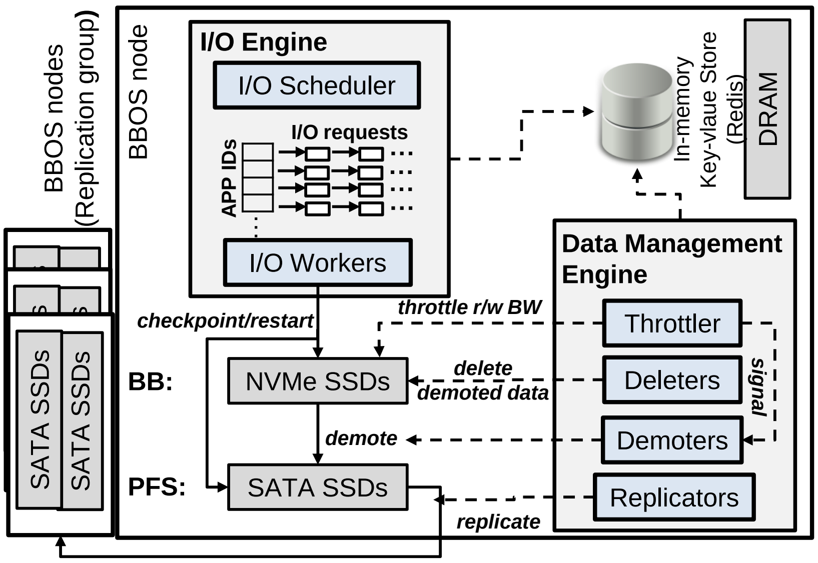


Figure 5.1: Architecture of BBOS

overhead [30]. 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 5.1. 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.

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.

5.1.2 Data Management Engine

As mentioned in section 3.2.2, because checkpoint/restart characteristics are not fully considered, some applications may suffer from a severe performance degradation. To address this problem, we first set the demotion threshold and adjust the speed of the checkpoint and demotion depending on the $DWPP$. In addition, we developed a data placement policy for the new HPC storage system to improve the BB efficiency and restart performance. With our data management approach, a checkpoint and demotion are managed for each specific period for expediency. We demote all data written during this period ($DWPP$) for easy management within the next period.

Adaptive Demotion Adjustment

To prevent the BB from overflowing, we determine a demotion threshold in consideration of the $DWPP$ and I/O job congestion. As shown in Figure 3.4, because the $DWPP$ affects the amount of data to be demoted during a period, a smaller demotion threshold is chosen for a larger $DWPP$. Even if the $DWPP$ is the same size, the BB may fill up depending on the I/O job congestion shown in Figure 3.5. Under a worst-case scenario, I/O jobs arrive without any idle time for the BB. To prepare for the worst case, we need to demote the number of data equal to the $DWPP$ minus the capacity of the BB (S), called C, along with a checkpoint execution. In addition, the speed of the checkpoint and demotion ($Bwmax$, $Bwmin$, $Brmax$, and $Brmin$) also affect the demotion threshold when demoting an amount of data equal to C with a checkpoint. The throughput of the checkpoint and demotion are influenced by the concurrent execution of the write and read operations. When a 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 the write and read bandwidth. As a result, the minimum demotion throughput ($Brmin$) is determined by the maximum checkpoint throughput ($Bwmax$), i.e., the maximum write throughput provided by the BB. The minimum checkpoint throughput ($Bwmin$) is determined by the maximum demotion throughput ($Brmax$), i.e., the maximum write throughput provided by the PFS, as shown in equation (5.1). (The m and b values may differ according to the various devices applied.) Thus, we adjust the speed of the checkpoint from $Bwmax$ to $Bwmin$, and the speed of demotion from $Brmin$ to $Brmax$ after the demotion threshold.

$$BW_w = m \times BW_r + b \qquad (m < 0) \tag{5.1}$$

To easily calculate the demotion threshold, we show our demotion management by categorizing the patterns of the demotion into three groups according to the $DWPP$, as shown in Figure 5.2. Here, S is the capacity of the BB, and

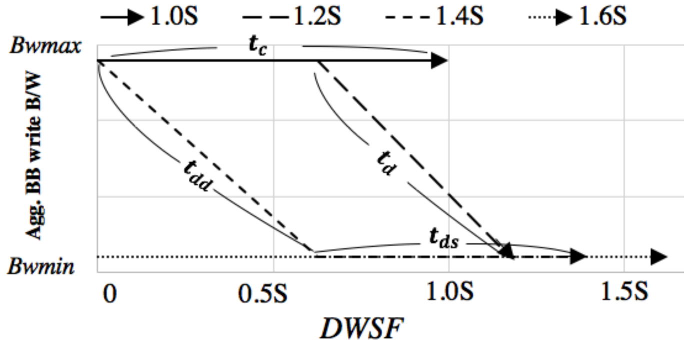


Figure 5.2: BBOS demotion management

$DWSF$ is the amount of data written thus far within the period. Within a single period, the time given to execute a checkpoint at $Bwmax$ without any demotion is t_c, and t_d is the time required to demote C while the checkpoint continues. Here, t_d is composed of t_{dd} and t_{ds}, the former being the time when the demotion throughput gradually changes from $Brmin$ to $Brmax$, and the latter being the time when the demotion throughput is $Brmax$ without changing.

1)Pattern 1: A demotion is only performed when the BB is idle
As shown in Figure 5.2, when $DWPP$ is 1.0 S, the BB does not overflow within this period because $DWPP$ has the same capacity as the BB. Thus, a checkpoint is possible with the $Bwmax$ without a concurrent execution of any demotion.

2)Pattern 2: A demotion is conducted with a checkpoint for a certain period of time
As with 1.2 S, shown in the figure, $DWPP$ has a larger capacity than the BB (S), resulting in a positive value of C for demotion with checkpoint execution. However, C is not as large, and thus a demotion needs to be conducted only for a certain time with a checkpoint. The threshold depends on C. The larger C is,

the earlier the ideal start time of the demotion. In the case of a 1.2-S $DWPP$, the threshold of $DWSF$ is 0.7 S. This means that a demotion is executed even if a checkpoint is applied when $DWSF$ reaches 0.7 S. The checkpoint throughput is adjusted between $Bwmax$ and $Bwmin$ for a demotion. With a change in the checkpoint throughput, the demotion throughput is also adjusted between $Brmin$ and $Brmax$.

3)Pattern 3: A demotion is always conducted with a checkpoint As $DWPP$ increases, C sufficiently increases to the point at which a demotion starts at the same time as the checkpoint. The demotion throughput increases from $Brmin$ to $Brmax$, and the larger C is, the more quickly the demotion throughput reaches $Brmax$. With 1.4 S, the threshold for a start of a demotion is zero, and the threshold for a demotion with $Brmax$ is 0.6S. If the demotion is applied using $Brmax$ from the beginning with a checkpoint such as 1.6S, we can handle the highest capacity. Therefore, the $DWPP$ from this scenario becomes the maximum period allowed.

With the following equation (5.2), the thresholds used to start a demotion and to demote with $Brmax$ are also determined according to C. Because it is mandatory to demote all data on the BB within a certain period for the sake of the next period, the period is determined as shown in equation (5.3).

$$
\begin{aligned}
& t_c > 0, \\
& \qquad \int_0^{t_c+t_{dd}} BW_w(t)\,dt \\
& \qquad = Bwmax \times t_c + \tfrac{Bwmax+Bwmin}{2} \times t_{dd} = DWPP \\
& \qquad \int_0^{t_c+t_{dd}} BW_r(t)\,dt \\
& \qquad = Brmin \times t_c + \tfrac{Brmax+Brmin}{2} \times t_{dd} = C \\
& t_c = 0, \\
& \qquad \int_0^{t_{dd}+t_{ds}} BW_w(t)\,dt \\
& \qquad = \tfrac{Bwmax+Bwmin}{2} \times t_{dd} + Bwmin \times t_{ds} = DWPP \\
& \qquad \int_0^{t_{dd}+t_{ds}} BW_r(t)\,dt \\
& \qquad = \tfrac{Brmax+Brmin}{2} \times t_{dd} + Brmax \times t_{ds} = C
\end{aligned}
\tag{5.2}
$$

$$
(period - (t_c + t_d)) \times Bdmax \geq S \tag{5.3}
$$

Data Placement Policy

To handle the limitations with an existing data placement, we developed a data placement policy based on the characteristics of a checkpoint/restart. In our data placement policy, a 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 a restart. For a high restart performance, we select cold files by considering the checkpoint version and failure rates. The old version checkpoint files do not need to be in the BB, and thus they have the highest priority to be cold files. If there are no old version checkpoint files in the BB, we identify the cold files based on the failure rates. In this paper, we determine the failure rates of the applications depending on the number of nodes used. However, as many prior studies have mentioned regarding the causes of failures, the failure rates can be determined using such causes.

Direct Checkpoint on PFS

We expect a change in HPC storage systems in the future, with *Brmax* reaching close to *Bwmin* because BB and PFS can be placed on the same node. Because cold data from the BB are destined to be in the PFS, such data do not need to be written on the BB first, wasting BB resources. For this reason, we optimize the data management approach by bypassing the BB. Because we know the failure rates of the incoming checkpoint, we can classify in advance whether the checkpoint is hot or cold by comparing the failure rates with those of other checkpoints on the BB. If the incoming checkpoint is determined to be cold, the checkpoint is directed to be written on the PFS. The optimized approach reduces the amount of demotion, diminishing the concurrent execution of the checkpoint and demotion. We can therefore provide a higher checkpoint performance.

5.2 Implementation

In this section, we describe the implementation details of the proposed approach.

5.2.1 In-memory Key-value Store

We utilize Redis [31], an open-source in-memory key-value store to help the processing of the engines. The BBOS stores the location of the checkpoint files and important information needed for data management in the key-value store. As shown in Table 5.1, nine key-value pairs are applied for the engines. One of the key-value pairs is used to provide the location of the files. After the checkpoint is completed, *I/O engine* saves the file path for each file name in the key-value pair. At that moment, the names of the files are stored in a sorted key-value list to identify cold data depending on the version number and failure rates. Based on the key-value list, *Demoters* demote the oldest checkpoint files first, and if there are no older versions of the checkpoint files, they start to demote the file with the highest failure rates. We also store $DWPP$ and $DWSF$ to determine the demotion threshold and throttle the speed of the checkpoint and demotion. In Sections 5.2.2 and 5.2.3, we describe the process flows of each engine using the key-value store in detail. Because the BBOS does not use a page cache for a checkpoint and restart, in-memory store is used to utilize unused memory and facilitate the engine execution.

5.2.2 I/O Engine

I/O engine schedules I/O jobs and finds an appropriate storage tier for the I/O jobs. Concurrently, it also collects the information for the demotion. The process flow of *I/O workers* is described as in algorithm 5.1.

	KEY	VALUE	Description
1	“V-order”	Sorted Set(MTBF, APPID)	Record victims and sort them by MTBF
2	“Clean”	List(APPID)	List up App ID which have demotion-finished files for cleanup when BB needs free space
3	APPID+“restart”	Sorted Set(MTBF,time)	Record the restart time and new MTBF for every read request for restart
4	AppID+deviceID	Sorted Set(ver, filename)	Record the version and the name of the checkpoint files of each HPC apps for each device
5	filename	String(location)	Record the file path for easy file access on the demotion or restart
6	APPID+“BB”	NULL	Record App ID if the checkpoint files of the app are in BB
7	“E-order”	List(APPID)	Record list of App IDs with more than two different checkpoint versions in BB
8	“DWSF”	String(DWSF)	Record DWSF to throttle the checkpoint/restart speed
9	“R-order”	filename	Record the file name for replicating files to remote storage nodes
10	“DWPP”	DWPP	Record the total amount of the checkpoint written to BB by the applications for a certain period

Table 5.1: Key-value pairs in BBOS

Algorithm 5.1: Pseudo-code for Checkpoint

```
1: if freespace != enough
2:     Signal to DELETER
3: if get('APPID+"BB"', temp) != NULL
4:     put('E-order', APPID)
5: put('APPID+NVMe#', 'filename')
6: put('filename', 'loc')
7: Executing writes for checkpointing..
8: put('"DWSF"', 'DWSF + current file size')
```

Demoted data can stay in the BB unless the capacity is insufficient for a new checkpoint. Thus, the engine first checks if there is sufficient space left before processing the checkpoint, and if not, it sends *Deleters*, a signal used to delete demotion-finished files (lines 1 and 2). In addition, the engine checks if there are any outdated checkpoint files of the application on the BB. This is because the older versions of the checkpoint files do not need to remain in the BB after the new checkpoint files are safely written. To check the files, the engine checks if there is a key of the application in pair #6, and if so, it records the app ID to pair #7, which collects victims with the highest priority (lines 3 and 4). The engine enlists the file names of each application on pair #4 for *Demoters* that perform the demotion on each high-end SSD in the BB (line 5). To provide the location of the files, the engine saves the file path for each file name in pair #5 (line 6). After completing the checkpoint, the engine updates pair #8 with the current file size for *Throttler* (lines 7 and 8). We need to record the *DWSF* because the capacity of the BB is always full owing to the fact that our system keeps the demotion-finished data in the BB until free space is actually necessary.

5.2.3 Data Management Engine

Four modules from this engine manage an efficient demotion process between the BB and the PFS.

Throttler

Throttler regulates the checkpoint and the restart speed while monitoring the *DWSF*. *Throttler* obtains the *DWSF* from pair #8 and compares the value with the threshold to start a demotion. After the threshold, the checkpoint/restart speed is adjusted, and the module throttles the speed as reconfigured.

Demoters

Demoters receive a signal from *Throttler* about a device in the BB that requires a demotion. After that, *Demoters* collect information from the in-memory store to execute the demotion described in Algorithm 5.2. First, *Demoters* check for every victim in pair #7 because the older version of the checkpoint files should be demoted first (line 1). If there is no victim, a victim is retrieved from pair #1, which is ordered by *MTBF* (lines 2 and 3). If the victim is from pair #1, the files of the victim are not deleted from the BB immediately after a demotion to preserve the restart performance (lines 7 and 8). However, it is necessary to record the app ID to pair #2 and erase these demoted files when the BB needs the available capacity (line 9). If the victim is from pair #7, it also means that the victim has older versions of the checkpoint files. Because the older version files do not need to remain in the BB, they are deleted (lines 10 and 11). Finally, *Demoters* update pair #5 (line 12) and put a filename in pair #9 for replication (line 13).

Algorithm 5.2: Pseudo-code for Demotion

```
1: pop("'E-order"', APPID)
2: if APPID != exists
3:    pop("'V-order"', {MTBF, APPID})
4:    flush ← TRUE
5: pop('APPID+NVMe#', {ver, files})
6: for each file of files
7:    if(flush == TRUE)
8:       demote file from NVMe# to SATA#
9:       put("'Clean"', APPID)
10:   else
11:      demote file of old version from NVMe# to SATA#
12:      delete file in NVMe#
13:   update('filename', 'loc')
14:   put("'R-order"', 'filename')
```

Deleters

Deleters erase the fully demoted files after receiving a signal from *I/O workers*. To do so, *Deleters* pop the app ID first, which is inserted in pair #2, and delete the files from the BB using pairs #4 and #5.

Replicators

Finally, *Replicators* replicate the checkpoint files from the local storage device to the remote devices within the same replication group. Each storage node has a mount point of the PFS, which consists of storage nodes in the same replication group except itself. A 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 #9 without interfering with the BB performance.

5.2.4 Stable Checkpoint and Demotion Performance

To provide a stable checkpoint/restart and demotion performance, a data management approach is optimized using new techniques. The checkpoint and demotion speeds are regulated as mentioned in 5.1.2. However, it is difficult to accurately throttle their write and read speeds. Because the number of I/O requests per second from each application varies, the speed of the checkpoint and demotion is different even if we send the same number of read requests per second for a demotion. The system may not be able to provide a stable checkpoint performance owing to the inability to demote as much data as it should. For this reason, we use blkio [32] of the cgroup to precisely throttle the speed of the checkpoint and restart. In addition, we utilize the $send\_file()$ system call [33] to maintain a stable demotion performance. For the demotion, the data must be read from the BB and written to the PFS. This causes a context switching and data copying overhead between the user and kernel level, resulting in a low and unstable demotion performance. Because the $send\_file()$ system call supports a zero-copy, we can eliminate the demotion overhead. Furthermore, the checkpoint/restart performance may be degraded owing to a garbage collection. To avoid a garbage collection overhead, we periodically request $TRIM$ after deleting the files. In addition, the $TRIM$ throughput is managed using blkio to minimize the performance degradation.

5.3 Evaluation

5.3.1 Experimental Environment

We evaluated our HPC storage management approach using eight compute nodes and a single storage node for the BB and PFS. Four of the compute nodes consist of an Intel Xeon Phi 7290 CPU processor with 72 physical cores and the others are of an Intel Xeon Phi 7250 CPU with 68 physical cores. The storage node consists of dual 12-core Intel Xeon Silver 4115 CPUs and 32 GB of memory. For the BB, we use four 800-GB FADU NVMe SSDs provided by a semiconductor start-up company [34], with a sequential write and read of up to 920 and 3200 MB/s, respectively. For the PFS, four 4-TB Samsung 860 EVO SATA SSDs are installed. The compute and storage nodes are connected with a 100 GbE Mellanox SN2100 switch.

We use the Gluster file system (GlusterFS) [35] version 5.6 for both the BB and PFS, the configuration of which is tuned for a high performance. In addition, the BBOS is implemented by modifying GlusterFS and adding some developed modules. In addition, each variable is determined as following: $Bwmax$ of 3.56 GB/s, $Bwmin$ of 3 GB/s, $Brmax$ of 1.6 GB/s, $Brmin$ of 0.08 GB/s, and *period* of 3800 s. For the experiments, we execute large sequential writes to simulate a checkpoint by applying a microbenchmark FIO [36], and the failure rates are determined based on the number of nodes used by each application. According to [14], the failure rates and mean time between failures ($MTBF$) have an inverse relationship. Thus, to express the failure rates of applications simply for experimental purposes, we utilize the $MTBF$.

To validate our system, we compare the BB utilization and checkpoint/restart performance with Datawarp, a current HPC scheduler that uses a dedicated BB allocation, and two policies of Harmonia [13], which is the only scheduler that

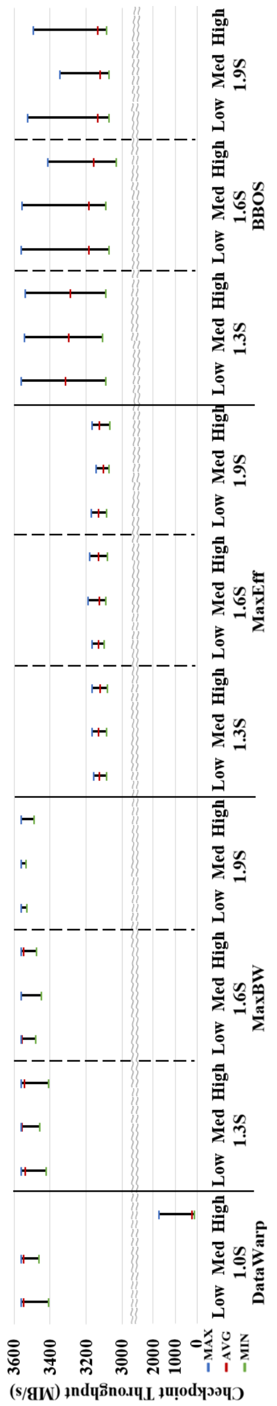


Figure 5.3: Checkpoint Throughput depending on $DWPP$

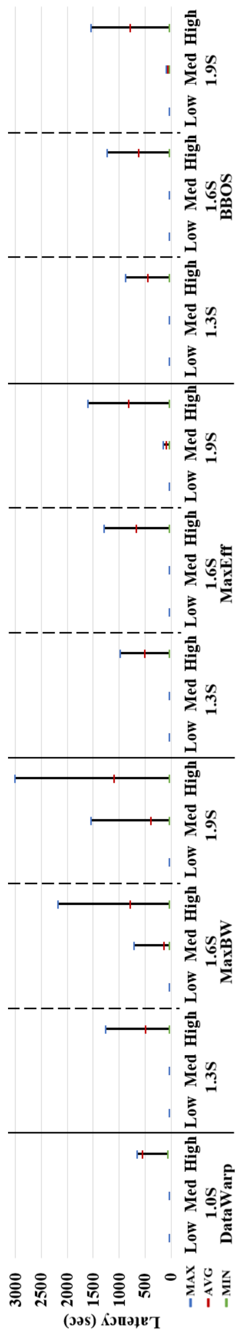


Figure 5.4: I/O latency depending on $DWPP$

uses a BB over-subscription method. Because Harmonia is not opensource, we created an emulation based on the paper. Datawarp does not apply I/O scheduling, whereas Harmonia schedules I/O jobs to prevent them from overlapping with each other. MaxEff, one of Harmonia's policies, aims to optimize the BB system efficiency by maximizing the BB utilization. Because the policy aims to maintain a high capacity of the available BB, it consistently demotes the data at full speed ($Brmax$) even when the checkpoint is concurrently applied. By contrast, MaxBW from Harmonia aims to provide the maximum checkpoint bandwidth to the applications. With this policy, the checkpoint and demotion are not conducted at the same time. To describe the demotion threshold of the two policies, as shown in Figure 5.2, the demotion thresholds of MaxEff and MaxBW are zero and 1 S for $DWSF$.

5.3.2 Burst Buffer Utilization

To compare the BB utilization with each approach, we assume that each application requests an 80-GB checkpoint once per period. We then count how many applications can finish the checkpoint within the period, which indicates the maximum $DWPP$ providable by each scheduler. Datawarp shows 0$\sim$100% of the BB utilization because it allocates as much BB capacity as the users desire with a dedicated allocation method. Under the best scenario, if all users demand as much BB allocation as they need, the total BB capacity is used up within this period, resulting in 100% BB utilization. In most cases, however, the BB utilization is low due to an overabundant BB capacity request and the checkpoint/restart characteristics. By contrast, because Harmonia and BBOS use an over-subscription BB allocation method, they can accommodate more applications within the period than Datawarp. MaxBW needs to ensure the maximum checkpoint throughput of the applications, and thus the demotion cannot be performed together with the checkpoint. As a result, a BB utilization of up to 190% is demonstrated. MaxEff shows a BB utilization of 210% because it always performs the demotion at the maximum demotion throughput, i.e., $Brmax$. To provide the maximum $DWPP$, a BBOS is applied similarly to MaxEff, which also achieves a BB utilization of up to 210%.

5.3.3 Checkpoint Performance

To validate the BBOS, we executed the experiments under various situations with different scenarios for the I/O job congestion and $DWPP$. Because the maximum $DWPP$ of Datawarp equals the total capacity of the BB, we only evaluate Datawarp with a $DWPP$ of 1 S, and the others with a $DWPP$ of 1.3, 1.6, and 1.9 S, respectively. We generated different I/O job congestion patterns based on the following three scenarios: 1) Low congestion: The time interval of each I/O job is equal and evenly distributed throughout the period. 2) Medium congestion: The time interval of each I/O job is half that of a low congestion. 3) High congestion: The time interval of each I/O job is shortened two one-tenth of a low congestion. (If I/O jobs are requested every 50 s under the low congestion pattern, I/O jobs arrive every 25 s, whereas they arrive every 5 s under the medium and high congestion patterns.) In addition, as described in section 5.3.2, all applications require an 80-GB checkpoint once per period.

Figure 5.3 shows the checkpoint throughput and I/O latency, which is the time interval between the time of an I/O request and the time of the response under various $DWPPs$ and an I/O job congestion. The I/O latency contains 1) the wait time until the end of the previous job to prevent a concurrent execution of I/O jobs, 2)the stop time to make BB space available owing to the full BB, and 3) the execution time of the I/O job. In Datawarp, because all demotions are possible within a sufficient BB idle time between I/O jobs, a high checkpoint throughput is provided under a low I/O job congestion. By contrast, under high congestion, the checkpoint throughput of each application remains extremely low owing to a 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 a similar average latency even with a $DWPP$ of 1.0 S when compared to the BBOS.

Unlike Datawarp, Harmonia and BBOS provide I/O scheduling to mitigate I/O interference across applications. Because MaxBW does not execute a demotion and checkpoint at the same time, it always provides a high checkpoint throughput. However, some I/O jobs have to stop before the execution based on the available BB capacity. With a low congestion, none of the applications wait to avoid I/O interference or make space in the BB, regardless of the $DWPP$, because of a sufficient idle time between I/O jobs. With medium congestion, when the $DWPP$ exceeds 1.6 S, some applications begin to achieve a high latency. The larger the $DWPP$ is, the shorter the idle time within a period, and thus the latency at 1.9 S is larger than that at 1.6 S. With high congestion, applications have to wait to prevent I/O interference and must stop to make space in the BB because there is insufficient idle time between jobs, which results in the highest latency. In addition, MaxBW has an extreme performance variance across applications because the maximum latency is too high compared to the average. Figure 5.5 shows the time-excluding execution time in terms of latency for the 1st to 45th I/O jobs with medium I/O congestion at 1.9 S. With MaxBW, no I/O job waits or stops until I/O job #36, but I/O jobs arriving after #36 have to stop for sufficient space in the BB prior to execution. Unfortunately, the stop time of all previous I/O jobs is accumulated. Therefore, the later arriving I/O jobs have a longer wait time, resulting in a severe performance fluctuation. Unlike MaxBW, the BBOS does not stop to increase the BB

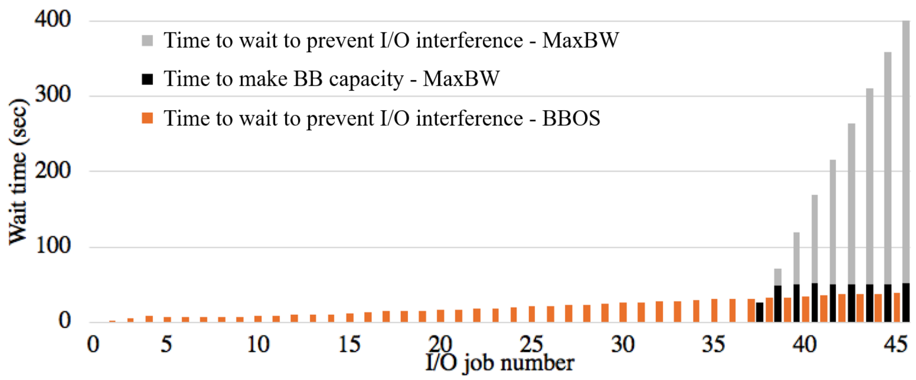


Figure 5.5: Wait time of I/O jobs with MaxBW

capacity because BBOS demotes the data in advance and thus the BB is not full. Therefore, the execution time of the I/O jobs increases, which forces the next I/O jobs to wait. As a result, the BBOS shows a gradual increase in the wait time initially, although the wait time of the I/O jobs does not significantly increase. In conclusion, MaxBW provides a higher performance than the BBOS when there is no wait time, such as Low and Med of $1.3S$ and Low of $1.6S$ and $1.9S$. However, if there is an insufficient BB idle time per period or idle time between I/O jobs, MaxBW shows the higher latency and a higher performance variance than the BBOS, because the BBOS prepares for situations in which I/O jobs come in groups by adjusting the checkpoint performance.

By contrast, because MaxEff and BBOS apply a demotion in advance to prevent a BB overflow, they do not stop the I/O jobs to create BB capacity prior to the I/O execution. MaxEff shows the lowest checkpoint throughput because this method always demotes data at the highest demotion speed. In this way, a relatively large space in the BB is maintained but provides a lower checkpoint latency compared to that of MaxBW. The BBOS adjusts the checkpoint throughput between $Bwmax$ and $Bwmin$ depending on the $DWPP$. The smaller $DWPP$ is, the higher the checkpoint throughput we achieve by avoiding unnecessary concurrent checkpoint and demotion executions unlike with MaxEff. In the absence of a wait time to create BB capacity, only the checkpoint throughput determines the latency, and thus the BBOS shows a lower latency than MaxEff (however, the latencies are too small to be seen in Figure 5.4). When the $DWPP$ is large, MaxEff shows a higher latency than the BBOS, even though it applies a demotion more aggressively than our approach. This is because MaxEff always demotes data at the full demotion speed and takes longer to process a checkpoint, which results in a longer wait period for the next I/O jobs. In our experiments, the difference in latency of MaxEff and BBOS seems to be small (within tens of seconds) because the difference between $Bwmax$ and $Bwmin$ is not large. If the difference increases, the BBOS can expect a lower I/O latency than MaxEff.

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

Direct checkpoint on PFS

When the maximum demotion throughput ($Brmax$) is greater than the minimum checkpoint throughput ($Bwmin$), we can reduce the unnecessary demotion overhead by bypassing the BB. We conducted experiments using three different $DWPPs$, i.e., 1.3, 1.6, and 1.9 S. Each application requests an 80-GB checkpoint with a checkpoint period of 1 h. The $MTBFs$ of all I/O applications are set randomly from 0 to 100 min. As shown in Figure 5.6, the method decreases the number of demoted data by up to 38% by bypassing the BB method. Because checkpoints can be written directly on the PFS after $DWSF$

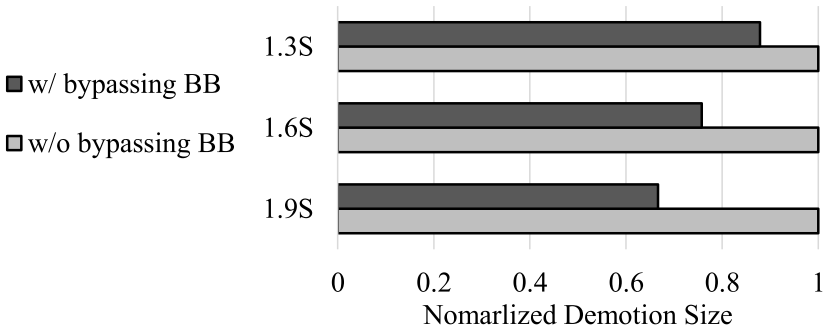


Figure 5.6: Direct checkpoint on PFS by bypassing BB

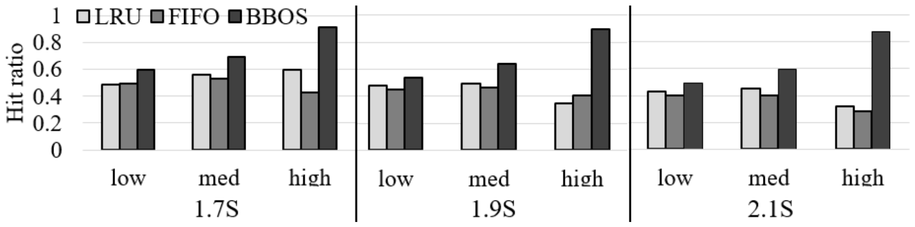


Figure 5.7: Hit ratio of restart requests on BB

becomes larger than the BB capacity, the larger the $DWPP$ is, the longer the period of time this method can be applied. Hence, as $DWPP$ increases, more checkpoint files of cold data can be written directly on the PFS. Consequently, because fewer demotions are applied concurrently with the checkpoints, more applications can experience a higher checkpoint throughput.

5.3.4 Restart Performance

In this section, we measure the hit ratio of the restart on the BB to compare the restart performance. We compare the LRU and FIFO algorithms used in most HPC data management schemes with the BBOS. Because the ratio of the remaining amount to BB of the total data differs according to the $DWPP$, we use three different $DWPPs$ for the experiments, i.e., 1.7, 1.9, and 2.1 S. In addition, we use three different variances of the $MTBF$ sets. We choose $MTBF$

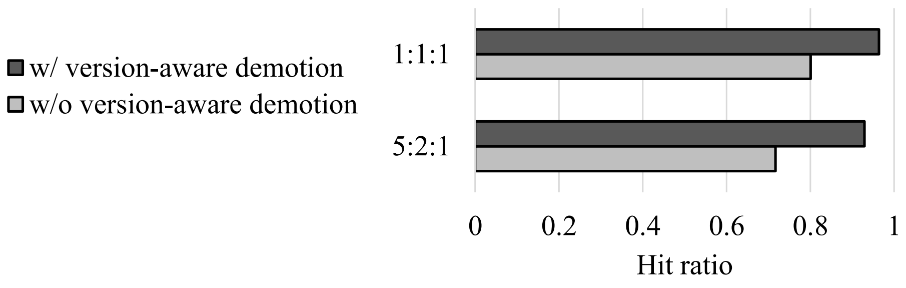


Figure 5.8: Version-aware data placement

randomly in the range of 0 to 20 min (low congestion), 0 to 50 min (medium congestion), and 0 to 100 min (high congestion). We choose the applications that need restart based on the expected $MTBF$. For the sake of simplicity, all checkpoints are fixed to equal periods, and the checkpoint size is set to 80 GB. As shown in Figure 5.7, the BBOS shows the highest hit ratio on the BB, which is up to 3.4-times higher in comparison. With all three methods, the hit ratio increases as $DWPP$ decreases because the checkpoint files have a higher chance of remaining on the BB. In the case of the LRU and FIFO algorithms, however, cold data are chosen based only on the order of the written time. Thus, the hit ratio for each experiment is largely different and unrelated to the variance of $MTBF$. By contrast, the BBOS shows an increased hit ratio as the variance increases. With a low variance, the effectiveness of our system is relatively lower than the other scenarios because failure rates of the applications are similar owing to a low variance of the $MTBF$. By contrast, in the case of a high variance, the checkpoint files are distributed well on the BB and PFS according to the failure rates. As a result, the BBOS provides up to a 3.1-times higher hit ratio of restarts on the BB compared to the others.

Version-aware Data Placement

To maintain a high hit ratio on the BB, the BBOS uses a version-aware data placement method by identifying outdated checkpoint files as cold data with the highest priority. To demonstrate the effectiveness of this method, a few assumptions are made for the following experiment. We choose three checkpoint periods for the HPC applications, i.e., 60, 30, and 20 min. Each user requests an 80-GB checkpoint and an $MTBF$ of 0 to 100 is randomly selected. We assume that the applications maintain three or more versions of the checkpoint. Thus, the applications with a 60-min period have one checkpoint version, and have two versions for a 30-min period and three versions for a 20-min period. Finally, we arrange the ratios of the three periods as 1:1:1 and 5:2:1 with a $DWPP$ of 1.9 S. Figure 5.8 shows a comparison of the restart hit ratio on the BB between the availabilities of the version-aware method. In the case of 1:1:1, all of the restart requests can be handled in the BB with the version-aware method in an ideal situation, because the BB capacity is larger than the total number of new version checkpoints. However, if we have to make available capacity when no older version checkpoints are available, a single version checkpoint with a high $MTBF$ can be selected as cold data. Therefore, 96.4% of the restart requests on average are applied on the BB in the actual experiments with the version-aware method. By contrast, without a version-aware method, the selection of cold data is based only on the $MTBF$. Fresh checkpoint files with a high $MTBF$ exist on the PFS, and older version files with a low $MTBF$ remain on the BB. As a result, 80.1% of the restart requests are given a high restart performance from the BB. In the case of 5:2:1, because there is a large number of applications with a low checkpoint period, none of the new-version checkpoint files can be placed in the BB. Thus, with the version-aware method, we handle 92.5% restart requests in the BB, which is slightly less than 1:1:1. Without this method, 71.7% of the restart requests are performed in the BB, which is also lower than 1:1:1. Consequentially, the version-aware method increases the restart requests up to 29.5%, which are applied in the BB.

5.4 Summary

Herein, we proposed BBOS for use in a new all-flash HPC storage system. Specifically, we over-subscribed the BB by only allocating it during I/O phases, and not during the entire lifetime for a higher BB utilization. To mitigate a performance reduction caused by an over-subscription, we provided the I/O scheduler and data management module. The I/O scheduler resolved the 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 a checkpoint/restart. Based on these characteristics, we transferred data from the BB to the PFS transparently by adjusting the thresholds and speed of the demotion according to the $DWPP$. We also identified cold data by considering different versions and failure rates.As a result, we improved the BB utilization by at least 2.2-times that of the dedicated BB allocation method. In addition, we guarantee a higher checkpoint throughput without a sudden performance reduction and handle 96.4% of restart requests in the BB, providing up to a 3.1-times higher restart performance than that of other approaches.

Chapter 6

Related Work

Many studies related to a multi-tiered HPC storage system include a BB, which is composed of expensive resources. Since the emergence of a BB, researchers have actively focused on improving the checkpoint performance in various ways. To reduce the checkpoint overhead on a PFS, in [1, 27], the authors have developed a multi-level checkpoint mechanism considering the different degrees of reliability and the checkpoint cost of each tier in an HPC storage system. In [37], the authors attempted to transfer data asynchronously for a checkpoint. In [30], the authors observed that the BB performance is excessively reduced owing to a garbage collection when multiple HPC users simultaneously use a BB. To mitigate the reduction in performance, they assigned isolated blocks to each user using multi-stream SSDs. However, these approaches do not consider the I/O interference across applications, which is one of the most important considerations for an HPC system. To mitigate the I/O interference, with reference to existing I/O schedulers for a PFS [29], some studies [38–40] have provided I/O scheduling techniques for a BB. In [39], the authors dynamically coordinated I/O jobs based on the past I/O behavior of the application and system characteristics. In addition, [2] developed an I/O scheduling technique by reshaping the I/O traffic from the BB to the PFS.

File servers [41–44] used for grid computing have increased the end-to-end connections because a single TCP/IP connection prevents high data rates from an advanced network technology [45–50]. However, as described above, there have been many studies on obtaining a high I/O bandwidth in all-flash BB servers, but studies on scaling up the connections in a distributed file system have received little attention.

Researchers have recently started to take an interest in BB utilization and the checkpoint performance. Some data management solutions [51] allowing multi-tiered HPC storage systems to capitalize on the benefits of a BB have been proposed. In [23], the authors suggested a goal-driven data management scheme that automatically manages data as required by the applications. Although the users do not have to move data manually, they must understand the application workflow to command the data movement. In [11,52], the authors claimed that HPC applications can achieve some frequently accessed data. Through I/O profiling, hot data are identified and placed in the BB. In [53], the authors regulate I/O traffic using a write access pattern of the applications. They detect randomness in the write traffic and only random writes are stored in the BB. In addition, sequential writes are propagated directly to the PFS. However, checkpoints, occupying most of the I/O traffic of an HPC system, do not have hotness or a random access pattern because all checkpoint data are requested sequentially for recovery.

Because the studies mentioned above still use a dedicated allocation method, the BB cannot be fully utilized. To solve this problem, in [28] the authors use a BB over-subscription method. In this case, five I/O scheduling policies are proposed with different aims; however, these policies have several limitations because the aims are too narrowly focused. In addition, a demotion policy, including the demotion threshold and demotion speed, is not considered, nor is a data placement method between the BB and PFS for an HPC storage system with an over-subscribed BB allocation. This therefore leads to a fluctuation in the checkpoint performance and a slow checkpoint/restart.

Chapter 7

Conclusion

All-flash HPC storage systems have recently been proposed. However, existing I/O transfer and data management schemes between the BB and PFS do not support new all-flash HPC storage systems, resulting in a low checkpoint and restart throughput and a low storage utilization. In this dissertation, we explored two problems for handling the bursty I/O of HPC applications and addressed them by proposing a new user-transparent I/O management using two types of schemes, i.e., I/O transfer schemes between the compute and server nodes and data management schemes between the BB and PFS.

In Chapter 3, we described the problems of existing I/O transfer approaches for HPC storage systems and the limitations of existing solutions in dealing with such problems. Owing to a single connection between a compute node and a server node of existing DFSs, a high I/O bandwidth of multiple flash storage devices cannot be utilized. Existing solutions for handling this problem require considerable efforts from DFS developers and users of HPC storage systems. Therefore, we proposed a user-transparent I/O transfer scheme to solve the problem caused by a single connection and to compensate for the limitations of existing solutions. We increased the number of connections without modifications of the DFSs and HPC applications by modifying a mount procedure and I/O processing procedures in a virtual file system. Experimental results prove that our scheme improves the write and read throughput for a checkpoint and restart by up to 6- and 3-times that of the existing I/O transfer method, respectively, using the original kernel.

In Chapter 4, we investigated the problems of the existing storage allocation method for HPC storage systems and the limitations of existing data management schemes. The dedicated BB allocation method is inefficient, causing a BB underutilization owing to a faster PFS access. For this reason, we utilized a BB over-subscription allocation method; however, this BB allocation method may degrade the performance of a checkpoint and restart. We therefore proposed a new data management scheme using a BB and PFS based on the characteristics of a checkpoint and restart. We managed the data movement from the BB to the PFS by adjusting the speed of the demotion and adaptively determine the candidates for demotion. Experimental results showed that our scheme improves the BB utilization by up to 2.2-times that of the most popular dedicated BB allocation scheme, Datawarp, and proved a higher and stable checkpoint performance. In addition, we achieved up to a 3.1-times higher restart performance than other data management schemes.

In a future study, using an I/O transfer scheme to increase the number of connections by creating server daemons, we need to bind the mount points of multiple BBs to the same directory of a local file system such that they have the same file system view. GlusterFS used in this dissertation applies metadata from the local file system, and thus multiple server daemons have the same metadata. However, in the case of DFSs that have their own metadata management system, each server daemon may have different metadata for the same file. Of course, the data consistency will be guaranteed in a local file system, although problems may arise in metadata-related I/O requests. For this reason, there is a root to revise our I/O subsystem, allowing the DFSs to extend the connections with the server daemons. With the data management scheme, because HPC applications have a consistent checkpoint period, we can accurately predict when the next checkpoint request of the application will arrive. This expectation enhances the checkpoint throughput without an extravagant demotion in advance. In addition, we can predict the failure rates of applications based only on the number of nodes used. In the future, the exact causes of a failure can be applied to determine the failure rates.

Bibliography

[1] K. Sato, K. Mohror, A. Moody, N. Maruyama, and S. Matsuoka, "A user-level infiniband-based file system and checkpoint strategy for burst buffers," in *Proceedings - 14th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2014*, pp. 21–30, IEEE Computer Society, 2014.

[2] T. Wang, S. Oral, Y. Wang, B. Settlemyer, S. Atchley, and W. Yu, "BurstMem: A high-performance burst buffer system for scientific applications," in *Proceedings - 2014 IEEE International Conference on Big Data, IEEE Big Data 2014*, pp. 71–79, Institute of Electrical and Electronics Engineers Inc., jan 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, 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 *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–11, IEEE, apr 2012.

[6] O. Yildiz, A. C. Zhou, and S. Ibrahim, "Eley: On the Effectiveness of Burst Buffers for Big Data Processing in HPC Systems," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 87–91, IEEE, sep 2017.

[7] G. K. Lockwood, D. Hazen, Q. Koziol, S. Canon, K. Antypas, J. Balewski, N. Balthaser, W. Bhimji, J. Botts, J. Broughton, T. L. Butler, G. F. Butler, R. Cheema, C. Daley, T. Declerck, L. Gerhardt, W. E. Hurlbert, K. A. Kallback-Rose, S. Leak, J. Lee, R. Lee, J. Liu, K. Lozinskiy, D. Paul, C. Snavely, J. Srinivasan, T. S. Gibbins, and N. J. Wright, "Storage 2020: A Vision for the Future of HPC Storage," tech. rep., 2017.

[8] John Bent, Brad Settlemyer and G. Grider, "Serving Data to the Lunatic Fringe: The Evolution of HPC Storage — USENIX," tech. rep.

[9] Z. Yang, M. Hoseinzadeh, A. Andrews, C. Mayers, D. Evans, R. T. Bolt, J. Bhimani, N. Mi, and S. Swanson, "AutoTiering: Automatic data placement manager in multi-tier all-flash datacenter," in *2017 IEEE 36th International Performance Computing and Communications Conference, IPCCC 2017*, vol. 2018-Janua, pp. 1–8, Institute of Electrical and Electronics Engineers Inc., feb 2018.

[10] "Nersc Perlmutter," in *https://www.nersc.gov/systems/perlmutter/*.

[11] T. Xu, K. Sato, and S. Matsuoka, "Explorations of Data Swapping on Burst Buffer," in *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*, vol. 2018-Decem, pp. 517–526, IEEE Computer Society, feb 2019.

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

[13] 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 *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*, vol. 2015-April, pp. 376–383, IEEE Computer Society, 2014.

[14] D. Tiwari, S. Gupta, and S. S. Vazhkudai, "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*, pp. 25–36, IEEE, jun 2014.

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

[16] 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.*

[17] 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).*

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

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

[20] F. Petrini, K. Davis, and J. C. Sancho, "System-level fault-tolerance in large-scale parallel machines with buffered coscheduling," in *Proceedings - International Parallel and Distributed Processing Symposium, IPDPS 2004 (Abstracts and CD-ROM)*, vol. 18, pp. 2903–2910, 2004.

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

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

[23] 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 *Proceedings - 2019 IEEE 33rd International Parallel and Distributed Processing Symposium, IPDPS 2019*, pp. 511–522, Institute of Electrical and Electronics Engineers Inc., may 2019.

[24] R. A. Ashraf, S. Hukerikar, and C. Engelmann, "Shrink or Substitute: Handling Process Failures in HPC Systems using In-situ Recovery," jan 2018.

[25] J. W. Young and J. W., "A first order approximation to the optimum checkpoint interval," *Communications of the ACM*, vol. 17, pp. 530–531, sep 1974.

[26] J. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Generation Computer Systems*, vol. 22, pp. 303–312, feb 2006.

[27] A. Moody, G. Bronevetsky, K. Mohror, and B. R. De Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2010*, 2010.

[28] A. Kougkas, H. Devarajan, and X.-H. Sun, "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*, (New York, New York, USA), pp. 219–230, ACM Press, 2018.

[29] M. Dorier, S. Ibrahim, G. Antoniu, and R. Ross, "Omnisc'IO: A Grammar-Based Approach to Spatial and Temporal I/O Patterns Prediction," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, vol. 2015-Janua, pp. 623–634, IEEE Computer Society, jan 2014.

[30] 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)*, pp. 245–255, IEEE, sep 2017.

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

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

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

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

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

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

[37] 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 *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 2012.

[38] S. Thapaliya, P. Bangalore, J. Lofstead, K. Mohror, A. Moody, and L. Livermore, "Managing I/O Interference in a Shared Burst Buffer System," in *2016 45th International Conference on Parallel Processing*, 2016.

[39] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Y. Robert, and M. Snir, "Scheduling the I/O of HPC Applications under Congestion," in *Proceedings - 2015 IEEE 29th International Parallel and Distributed Processing Symposium, IPDPS 2015*, pp. 1013–1022, Institute of Electrical and Electronics Engineers Inc., jul 2015.

[40] A. Kougkas, M. Dorier, R. Latham, R. Ross, and X. H. Sun, "Leveraging burst buffer coordination to prevent I/O interference," in *Proceedings of the 2016 IEEE 12th International Conference on e-Science, e-Science 2016*, pp. 371–380, Institute of Electrical and Electronics Engineers Inc., mar 2017.

[41] "GridFTP ."

[42] "bws/xdd: XDD - The eXtreme dd toolset."

[43] "Fast Data Transfer."

[44] L. Zhang, W. Wu, P. DeMar, and E. Pouyoul, "mdtmFTP and its evaluation on ESNET SDN testbed," *Future Generation Computer Systems*, vol. 79, pp. 199–204, feb 2018.

[45] F. García-Carballeira, J. Carretero, A. Calderón, J. D. García, and L. M. Sanchez, "A global and parallel file system for grids," *Future Generation Computer Systems*, vol. 23, pp. 116–122, jan 2007.

[46] D. Nadig, E. S. Jung, R. Kettimuthu, I. Fosterz, S. V. Nageswara Rao, and B. Ramamurthy, "Comparative Performance Evaluation of High-performance Data Transfer Tools," in *International Symposium on Advanced Networks and Telecommunication Systems, ANTS*, vol. 2018-December, IEEE Computer Society, jul 2018.

[47] Y. Liu, Z. Liu, R. Kettimuthu, N. Rao, Z. Chen, and I. Foster, "Data Transfer between Scientific Facilities – Bottleneck Analysis, Insights and Optimizations," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 122–131, IEEE, may 2019.

[48] S. B. Lim, G. Fox, A. Kaplan, S. Pallickara, and M. Pierce, "GridFTP and parallel TCP support in NaradaBrokering," in *International Conference on Algorithms and Architectures for Parallel Processing*, vol. 3719 LNCS, pp. 93–102, Springer, Berlin, Heidelberg, 2005.

[49] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster, "The Globus striped GridFTP framework and server," in *Proceedings of the ACM/IEEE 2005 Supercomputing Conference, SC'05*, vol. 2005, 2005.

[50] K. T. Murata, P. Pavarangkoon, K. Yamamoto, Y. Nagaya, K. Muranaga, T. Mizuhara, A. Takaki, O. Tatebe, E. Kimura, and T. Kurosawa, "Multiple streams of UDT and HpFP protocols for high-bandwidth remote storage system in long fat network," in *7th IEEE Annual Information Technol-*

ogy, Electronics and Mobile Communication Conference, IEEE IEMCON 2016, Institute of Electrical and Electronics Engineers Inc., nov 2016.

[51] 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.*

[52] 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 *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST2010*, IEEE Computer Society, 2010.

[53] X. Shi, M. Li, W. Liu, H. Jin, C. Yu, and Y. Chen, "SSDUP: A Traffic-Aware SSD Burst Buffer for HPC Systems," in *ICS 2017: International Conference on Supercomputing June 14-16, Chicago: Sponsored by ACM/SIGARCH*, vol. 10, 2017.

요약

고성능 컴퓨팅 스토리지 시스템의 입출력 대역폭의 대부분은 고성능 어플리케이션의 체크포인트와 재시작이 차지하고 있다. 이런 고성능 어플리케이션의 폭발적인 입출력을 원활하게 처리하게 위하여, 고급 플래시 저장 장치와 저급 플래시 저장 장치를 이용하여 버스트 버퍼와 PFS를 합친 새로운 플래시 기반의 고성능 컴퓨팅 스토리지 시스템이 제안되었다. 하지만 스토리지 시스템을 구성하기 위하여 사용되는 대부분의 분산 파일 시스템들은 노드간 하나의 네트워크 연결을 제공하고 있어 서버 노드에서 제공할 수 있는 높은 플래시들의 입출력 대역폭을 활용하지 못한다. 여러개의 네트워크 연결을 제공하기 위해서는 분산 파일 시스템이 수정되어야 하거나, 분산 파일 시스템의 클라이언트 데몬과 서버 데몬의 갯수를 증가시키는 방법이 사용되어야 한다. 하지만, 분산 파일 시스템은 매우 복잡한 구조로 구성되어 있기 때문에 많은 시간과 노력이 분산 파일 시스템 개발자들에게 요구된다. 데몬의 갯수를 증가시키는 방법은 각 네트워크 커넥션마다 새로운 마운트 포인트가 존재하기 때문에, 직접 파일 입출력 리퀘스트를 여러 마운트 포인트로 분산시켜야 하는 엄청난 노력이 사용자에게 요구된다. 서버 데몬의 개수를 증가시켜 네트워크 커넥션의 수를 증가시킬 경우엔, 서버 데몬이 서로 다른 파일 시스템 디렉토리 관점을 갖기 때문에 사용자가 직접 서로 다른 서버 데몬을 인식하고 데이터 충돌이 일어나지 않도록 주의해야 한다. 게다가, 기존에는 사용자들이 하드디스크와 같은 저속 저장 장치로 구성된 PFS로의 접근을 피하기 위하여, 버스트 버퍼의 효율성을 포기하면서도 전용 버스트 버퍼 할당 방식 (Dedicated BB allocation method)을 선호했다. 하지만 새로운 플래시 기반의 고성능 컴퓨팅 스토리지 시스템에서는 병렬 파일 시스템으로의 접근이 빠르기때문에, 해당 버스트 버퍼 할당 방식을 사용하는것은 적절치 않다.

이런 문제들을 해결하기 위하여, 본 논문에서 사용자에게 내부 처리과정이 노출

되지않는 새로운 플래시 기반의 고성능 스토리지 시스템을 위한 효율적인 데이터 기법들을 소개한다. 첫번째 기법인 입출력 전송 관리 기법은 분산 파일 시스템 개발자와 사용자들의 추가적인 노력없이 컴퓨트 노드와 서버 노드 사이에 여러개의 커넥션을 제공한다. 이를 위해, 가상 파일 시스템의 마운트 수행 과정과 입출력 처리 과정을 수정하였다. 두번째 기법인 데이터 관리 기법에서는 버스트 버퍼의 활용률을 향상 시키기 위하여 버스트 버퍼 초과 할당 기법 (BB over-subscription method)을 사용한다. 하지만, 해당 할당 방식은 사용자 간의 입출력 경합과 디모션 오버헤드를 발생하기때문에 낮은 체크포인트와 재시작 성능을 제공한다. 이를 방지하기 위하여, 체크포인트와 재시작의 특성을 기반으로 버스트 버퍼와 병렬 파일 시스템의 데이터를 관리한다.

본 논문에서는 제안한 방법들의 효과를 증명하기 위하여 실제 플래시 기반의 스토리지 시스템을 구축하고 제안한 방법들을 적용하여 성능을 평가했다. 실험을 통해 입출력 전송 관리 기법이 기존 기법보다 최대 6배 그리고 최대 2배 높은 쓰기 그리고 읽기 입출력 성능을 제공했다. 데이터 관리 기법은 기존 방법에 비해, 버스트 버퍼 활용률을 2.2배 향상 시켰다. 게다가 높고 안정적인 체크포인트 성능을 보였으며 최대 3.1배 높은 재시작 성능을 제공했다.

주요어: 운영체제, 고성능 컴퓨팅, 분산 파일 시스템, 병렬 파일 시스템, 파일 입출력 서브시스템, 데이터 관리, 네트워크 확장, 가상 파일 시스템, 버스트 버퍼, 체크포인트, 재시작, 디모션

학번: 2013-20801

감사의 글

2013년 겨울, 눈이 쌓인 언덕길을 따라 302동으로 첫 등교를 하던 때가 엊그제 같은데 벌써 2020년 여름이 되어 마지막 등교를 앞두게 되었습니다. 꽃 같던 20대를 지나 30대 초반까지 생활했던 연구실을 막상 떠나려니, 후련하기도 하고 아쉽기도 합니다. 눈물과 웃음이 함께 했던 몇 년 동안, 인복이 많아 수많은 분들에게 갚을 수 없는 사랑과 도움을 받았습니다.

우선 새로운 연구의 길로 인도해 주신 엄현상 지도교수님, 언제나 세심한 지도를 해주신 염헌영 교수님께 감사의 말씀을 올립니다. 입학부터 졸업까지 희로애락을 함께해준 명원 오빠, 지웅 오빠 그리고 나의 짐을 함께 나눠주었던 수진이, 지수, 지우, 화정 언니, 계신 오빠, 정용이, 동규 그리고 분산시스템연구실 동료들에게 미안함과 고마움이라는 상반된 감정을 동시에 전합니다.

하나뿐인 딸 스트레스 받을까 봐 묵묵히 뒤에서 아낌없이 지원해준 사랑하는 우리 아빠, 엄마, 누나 수발드느라 힘들었을 착한 내 동생 그리고 언제나 아껴주시는 어머님께 형언할 수 없는 감사함을 전합니다. 새벽에 집에 들어가고 주말에 학교에 나가 있어도 서운한 내색 하나 없이 언제나 내 편이 되어주던 소중한 내 남편 박경원 씨에게도 고마움과 함께 무한한 사랑을 전합니다.

어디서도 경험할 수 없는 값진 시간이였습니다. 이곳에서의 소중한 경험을 발판삼아 앞으로도 성장하는 삶을 이어가고자 합니다. 다시 한번, 올바른 길로 이끌어주신 모든 분들께 감사의 말씀 드립니다.