Managing I/O Interference in a Shared Burst Buffer System

Sagar Thapaliya and Purushotham Bangalore
University of Alabama at Birmingham
Birmingham, AL, USA
Email: {sagar, puri}@uab.edu

Jay Lofstead
Sandia National Laboratories
Albuquerque, NM, USA
Email: gflofst@sandia.gov

Kathryn Mohror and Adam Moody
Lawrence Livermore National Laboratory
Livermore, CA, USA
Email: {kathryn, moody20}@llnl.gov

***Abstract*—In this work, we investigate the problem of inter-application interference in a shared Burst Buffer (BB) system. A BB is a new storage technology for HPC architectures that acts as an intermediate layer between performance-hungry HPC applications and the slow parallel file system. While the BB is meant to alleviate the problem of slow I/O in HPC systems, it is itself prone to performance degradation under interference. We observe that the magnitude of interference effects can reach a level that matters to the HPC system and the jobs that run on it. We investigate I/O scheduling techniques as a mechanism to mitigate BB I/O interference. With our results, we show that scheduling techniques tuned to BBs can control interference and significant performance benefits can be achieved.**

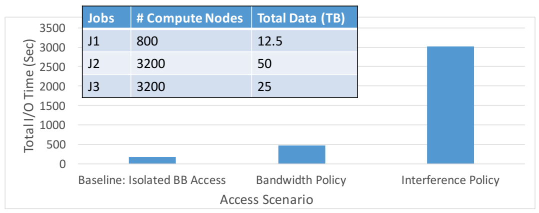
***Keywords*-Burst Buffer, Parallel I/O, Data Storage, Non Volatile Memory, Resource Management, Scheduling, I/O Interference**

I. Introduction

Today, high performance computing (HPC) I/O systems face many challenges. As applications scale, they demand specialized hardware and software to meet the performance needs of their increasingly intensive and concurrent I/O requests. Currently, the parallel file system (PFS) and I/O middleware provide such support [1], [2], but the PFS struggles to meet the performance demand. First, it is costly to scale up a PFS because the bandwidth to cost ratio is relatively low in disk-based storage systems. Second, the PFS can suffer from inter-application interference as HPC clusters are normally shared across multiple jobs that can access the PFS concurrently. The concurrent accesses can interfere with each other resulting in I/O performance degradation due to contention [3], [4].

To reduce contention, a new storage technology called a burst buffer (BB) has been introduced [5]. A BB is a layer of fast storage devices, such as non-volatile memory (NVM), between HPC applications and a relatively slow PFS. Vendors provide APIs for using BBs as a fast write or read staging area for the PFS. Recent research has shown that a BB can act as a cost effective solution to achieving a cost/performance balance [5]. In an HPC system, the BB will provide high performance, but is expected to be installed with a limited capacity, *e.g.*, the Trinity system at Los Alamos National Laboratory (LANL) has a usable BB capacity of 1.75× system memory [6]. The PFS is still used for capacity and data is asynchronously moved between the PFS and intermediate BBs reducing the effective I/O overhead of applications [5], [7].

HPC systems equipped with BBs will be available in the near future, *e.g.*, Trinity at LANL [6] and Cori at the National Energy Research Scientific Computing Center (NERSC) [8]. Vendors are creating software products to support and manage BB systems. For example, Cray provides DataWarp software



Jobs	# Compute Nodes	Total Data (TB)
J1	800	12.5
J2	3200	50
J3	3200	25

Fig. 1: Simulation of 3 concurrent jobs writing to a pool of 100 BB nodes on Trinity, under different BB allocation policies.

[7] and the SLURM scheduler supports BBs [9]. BBs are expected to strongly impact machines from the current new generation through exascale systems. As machines are deployed with BBs, uses beyond checkpoint/restart, such as in-transit data processing and data staging between workflow components [10] are expected to develop. We expect BBs to support these workloads alleviating the I/O problem that HPC systems currently face.

Today, little attention is paid to managing a BB system as a shared resource across multiple applications. Most of the work on BBs has explored its potential usefulness [5], [11] and the design of software systems to enable access from individual user applications [11], [12]. There is some work for managing it in a multi-application environment. For example, BB allocation policies based on limited write lifetimes of SSDs [13] and allocation policies based on capacity [7], [9]. Wang et al. [14] address part of the contention problem in the PFS when draining data from a BB. However, there are more sources of contention in the BB itself that need to be addressed, such as performance variability in SSDs during simultaneous access from multiple applications [15].

As motivation, in Figure 1, we show results for a simulation experiment to explore BB allocation policies based on bandwidth maximization and interference reduction using policies similar to those in DataWarp [7] and assuming the BB architecture of Trinity. These two allocation policies resulted in increases in total aggregated write time for three concurrently running jobs by up to 6.3× and 2.6× respectively compared to the baseline case. The impact for individual jobs is significant too. For example, in this experiment, the 800-compute node job (J1) suffered write time increases as much as 33× and 7× under these policies. In this simulation, we only capture a slowdown resulting from sharing BB nodes' I/O bandwidth across concurrent applications. These are significant performance penalties. The penalties can go even higher in real systems if additional contention occurs on shared resources such as SSDs and network.

There are also potential issues with an alternative BB architecture [8] where BB storage is placed local to compute nodes. For this architecture, jobs may have the advantage of reduced interference if they access the local BB. However, sharing may still be required, *e.g.*, to access shared data between workflow components or to access more BB space than what is available locally. During such sharing of BB resources on compute nodes, interference can even extend to other resources and operations such as memory access and network communications [16], [17].

Thus, issues remain with shared BB systems for which existing solutions may not provide effective support. Specifically, inter-application I/O interference is a dynamic issue occurring at runtime with the arrival of concurrent I/O requests from different jobs. We need a more robust mechanism to manage interference in a shared BB system. In this paper, we focus on using scheduling as a robust technique for managing interference in a shared BB system. We argue that scheduling I/O requests at runtime can be beneficial for controlling interference in a shared BB system and thus should be made an important component of BB management software. Such scheduling can be very effective at a system-wide access level where all BB access requests or I/O traffic goes into a global I/O queue and scheduling is applied to this queue. We believe scheduler specialization to fit the BB system and its workload may be more effective than a simple adaptation of scheduling.

A. Contribution

This paper demonstrates that interference effects will still be a problem for BBs and that BB scheduling is an effective management approach to address that I/O interference. We further show the importance of adapting the scheduler to the BB usage model and workloads.

To validate these points, we first quantify the impact of BB interference on HPC jobs. We then demonstrate the impact of different I/O scheduling techniques. These include (1) adoption of scheduling techniques from PFS I/O traffic, which controls high level I/O traffic with coarse grained access control (serialization) and (2) specialization of this technique to adapt to the BB system by adaptive sharing aware serialization. We evaluate these techniques though simulation and empirical experiments. For the evaluation, we use micro-benchmarks and realistic HPC I/O workloads generated by using I/O characteristics of supercomputer jobs from published papers.

We find that these scheduling techniques can effectively manage interference in a shared BB system. The coarse grained serialization can be effective when all the jobs are allocated BB resources across all the BB nodes in the system, *e.g.*, 62% reduction in total I/O time for a workload. Similarly, sharing aware serialization can be effective when individual jobs are allocated BB resources across subsets of the BB nodes, *e.g.*, 36% reduction in total I/O time for another workload.

II. Background

This section provides background on different implementations of BB systems, I/O workloads we expect to benefit from BBs, and sources of contention when using SSDs. We also motivate the need for explicit scheduling of BB resources.

A. BB System

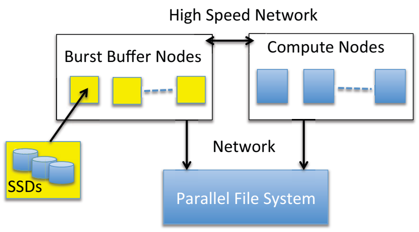


Fig. 2: BB architecture considered in this study.

In this work, we consider a shared BB system architecture where BBs are located in dedicated BB nodes and are shared by compute nodes (Figure 2). The BB uses an NVM-based storage system, such as flash based SSDs, to provide higher I/O bandwidth than disk. The BB resources are on separate nodes located within the same network space as the compute nodes. This allows compute nodes to operate without the performance degradation of external processes reducing network performance by accessing the local SSD. Instead, BB accesses are largely isolated by the network switches.

Under this model, application processes can access the BB system using dedicated software libraries directly accessing the BB or hidden underneath existing I/O middleware. Early work in this direction are presented by Sato et al. [11], Wang et al. [12], and Lofstead et al. [18]. Recently, DataWarp from Cray [7] and BB support from SLURM [9] were introduced.

There are alternative models for BB architecture such as spreading it across all of the compute nodes as a storage device or even on the memory bus on each compute node. In this paper, we focus on BB as dedicated secondary storage.

B. I/O Workloads for BB

The BB can act as a fast storage system to support various use cases for HPC applications including checkpoint/restart (C/R), pre-staging of input data, and for staging data between workflow components. C/R is a fault tolerance technique [19] where applications periodically dump state into storage system. In case of failure, the application will read the saved C/R data and restart computation using that saved state. C/R constitutes a large fraction of I/O time in HPC workloads [19].

In addition to C/R, BBs can act as a staging area between compute nodes and the PFS to support I/O operations. Pre-staging of input data to BBs can help applications incur less I/O time for reading input data compared to reading from the slower PFS, which is prone to issues such as inter-application interference [4]. Staging data in BBs can support operations such as in-transit data processing [20], which may include data visualization or analysis. Similarly, it can facilitate efficient sharing of transient output data between multiple workflow components by providing fast intermediate storage for shared data. The BB can also be used as a hidden cache for PFS access. In that case, data will be temporarily staged in the BB while it moves in either direction between the PFS and compute nodes.

All of the above use cases involve data transfers between 1) applications and BB, and/or 2) BB and the PFS. There can also be traffic across BBs, e.g., during in-transit data processing or data reorganization. All of these traffic patterns can cause interference. Without robust management, interference can result in contention and I/O performance degradation.

C. Interference in SSD

When applications use SSDs, they can avoid direct interaction with the magnetic disk-based PFS where contention and performance variability is an issue [4] and the seek and rotational latency of disk magnify the interference effects [21]. Instead, when using the SSDs, applications can avoid these two sources of latency resulting in much faster I/O performance. Since BBs use SSDs instead of magnetic disks, the community is expecting reduced I/O contention and performance variability resulting from inter-application interference.

However, SSD performance also suffers with interference under shared access. Researchers have shown that I/O interference can occur between both read and write I/O traffic in SSDs [22]. This occurs due to contention for service of concurrent I/O operations and due to the overhead of garbage collection. For NAND-based flash chips, data must be erased before any new data can be added to a block with existing data. This operation can interfere with I/O traffic since both will need to access storage locations in SSDs.

D. Need for BB Scheduler

HPC systems contain multiple schedulers at different layers. The most visible is the batch scheduler that manages execution of jobs on the cluster by selecting compute nodes and launching specific jobs on them. Another is the I/O scheduler for the PFS, which may run at storage severs to manage I/O requests at a given server. Multiple storage servers may even coordinate with each other to synchronize their scheduling of different jobs. In addition, we may also have a higher level global I/O scheduler that may schedule I/O steps of different jobs instead of managing I/O requests from individual I/O processes.

While such schedulers already exist in an HPC system, we argue that we need a scheduling mechanism that actively manages I/O traffic of BB system across workloads and I/O requests. Scheduling BB I/O traffic can be done by a dedicated BB scheduler or even be included inside a system-wide I/O scheduler after making it aware of BB I/O traffic.

Current BB management software packages, DataWarp and SLURM, focus on managing BB space capacity, but do not provide strong support for interference management. DataWarp provides mechanisms to control placement of space allocation on BB nodes when a job starts and also supports policy options that can be applied during placement including bandwidth and interference. Under the bandwidth policy, a job is allocated as many BB nodes as possible. Under the interference policy, a job is allocated as few BB nodes as possible. In both cases capacity requests are met. The goal of the bandwidth policy is to provide higher aggregate BB bandwidth whereas the interference policy aims to reduce inter-application interference due to sharing of BB resources.

We evaluated the DataWarp strategies in Section I (Figure 1). We assume that I/O of each job is scalable across all BB nodes available. Under the bandwidth policy, each job gets a BB allocation across all BB nodes, which is same as the baseline case, except in the baseline each job had isolated access to the allocated BBs. With that experiment, we found that the DataWarp strategies can have issues of performance reduction under both the bandwidth and interference policies. This indicates that we need a more robust mechanism to manage interference in a shared BB system.

III. Experimental Framework

In this section, we describe our experimental methodology for evaluating BB scheduling policies empirically and with simulation. We also show validation results for our simulator from empirical experiments.

A. Empirical Tool

For our empirical experiments, we developed a benchmark tool similar to IOR [23]. Our tool emulates the I/O behavior of concurrently executing MPI applications, provides them access to BB resources, and applies I/O scheduling to their BB accesses. This tool is sufficient to meet our requirement for the investigation in this paper — it generates situations of shared access to BB resources from multiple applications resulting in degraded I/O performance and controls the I/O requests according to scheduling policies. Such an approach appears elsewhere in the literature [24]. In a production environment, the BB scheduler can be implemented as a system component, *e.g.*, as part of the BB management software.

1) Tool Design: We implemented the tool as an MPI program written in C. We used MPI sub-communicators to partition ranks into the needed roles, namely one sub-communicator for each application and one for the scheduler. We used MPI messages to communicate between the applications and the scheduler.

2) BB File System: We use the user-level BB file system IBIO [11] and its C API for the applications to access the BB nodes. IBIO uses RDMA for data transfers between the compute and BB nodes. We have added a BB resource allocation layer on top of the IBIO library, which manages mapping of BB nodes to a given application at its start up and presents the individual BB servers as a unified BB system.

3) Interactions: Each application uses a single process, which we call the I/O coordinator, to talk to the scheduler. The I/O scheduler uses its main process to manage communications and runs a separate thread to perform scheduling.

When an application needs to access the BB, its I/O coordinator sends the scheduler a request for access permission. The I/O coordinator blocks while it waits for permission. The scheduler thread schedules the new request. The token management thread messages each waiting I/O coordinator in turn as the schedule dictates indicating it is their turn to access the BB. The application can then access the BB and then notifies the scheduler after its I/O step is complete. The scheduler is thus aware of the active BB access phase of each application. It can utilize this information during scheduling, *e.g.*, control the number of applications that access BB at a given time.

In our tool, explicit messages are exchanged between the applications and the I/O scheduler. However, in a production environment, we expect that such coordination could occur transparently to applications in I/O libraries or the file system. In addition, this approach would also enable a robust mechanism for tracking application BB access sessions. Should an application that is currently allocated a portion of the BB pool fail, the scheduler will be notified by the progression of the job script allowing recovery of the lost token and reallocation to the remaining scheduled applications.

B. Simulator

We design a deterministic discrete event simulator (DES) to model access to a BB system from parallel applications. We employ a simulator because it enables us to explore the performance of large-scale systems not yet available, e.g., Trinity. Here we describe how we model the different components and functionality of a BB system in the simulator.

1) I/O Request Model: We model I/O requests from HPC applications by using these parameters: number of I/O processes, data output per I/O process, and request arrival time. Such I/O requests represent I/O steps of applications such as writing a checkpoint data set. Each application also requests BB resources in the form of (a) a storage capacity, which can be equal to or more than its total output data size, and (b) a number of BB nodes across which the capacity should be allocated. To create a multi-application workload mix, we configure each application individually.

2) System Model: A BB system contains multiple BB nodes. The resource allocator serves resource requests of jobs according to requested capacity and BB node count. We use a round robin policy to select BB nodes for jobs.

3) Execution model: We run the simulation in multiple discrete steps. During each step, the system runs for a time epoch between two event points. The events represent changes such as arrival of a new I/O request, the start of execution of a new I/O request, or completion of an active request on one or more BB nodes. Each event can result in changes in the active applications that share the BB resources. With a change in sharing, change also occurs in the BB bandwidth available to each active application. We track bandwidth available to applications and length of time epoch for each step.

At the beginning of each step, we calculate I/O bandwidth allocated to the running applications. We present the model for computing the I/O bandwidth distribution in Section III-B4. We also track the next system event and use that to compute the simulation time epoch for that step. For the next system event, we use the earliest event across all the BB nodes. Using a common system event helps synchronize the progress of simulation across all the system components.

4) BB Bandwidth Distribution Model: Given a BB node and a set of jobs concurrently accessing it, the bandwidth that each job gets depends on the I/O attributes of all the jobs involved. In this paper, we follow a linear bandwidth division model where at a given time, the I/O bandwidth of a BB node is evenly distributed across all the I/O processes (across all the concurrent jobs) accessing it. We decided to use this model based on our observation of I/O performance from the Catalyst cluster. Using this model, the I/O bandwidth for each active I/O process accessing a BB node B is computed as

$$IOP\_BW_B = \frac{\text{peak bandwidth of B}}{\sum_{\forall \text{ job j with active I/O on B}} \text{\# I/O procs. of j}}$$

Then the I/O bandwidth an application A gets on this node is:

$$bandwidth_A = (\text{\# of I/O procs. of A}) \times IOP\_BW_B$$

TABLE I: Compute and BB System Simulation Parameters

Total compute nodes	19000	Total BB nodes	576
Compute node memory	128 GB	BB Node Space	6 TB
Compute node processor	32 cores	BB node bandwidth	5 GB/s

5) Application I/O Cost Model: To compute the I/O cost for each application writing data to a a shared BB system, we assume an application is allocated BB resources across N BB nodes and is writing data d_1, d_2, $\ldots d_n$ concurrently to the N BB nodes. The total time for an application I/O step is the maximum of the time it takes to write data across all of its BB nodes (*i.e.*, $max(T_{d1}, T_{d2}, \ldots T_{dn})$). Under our DES simulation model, writing of data to each BB node happens in multiple time epochs, t_1, t_2, $\ldots t_j$, $\ldots t_m$ for writing d_i to BB node B_i. Here, the application can receive different BB bandwidths in each time epoch, bw_1, bw_2, bw_j,$\ldots bw_m$. The bandwidth for each epoch will determine how much data from d_i will be written to B_i. The total time taken to write data d_i is $IOT_{bi} = \sum t_j$. In the presence of scheduling, a job may incur wait time in the I/O queue, which can happen for any time epoch t_j. For a time epoch that represents wait time for an application, the corresponding bandwidth b_j will be zero, which means that no data will be transferred during that epoch.

C. Evaluation Metric

As a primary metric for evaluating I/O performance, we use the total time to complete an I/O step for individual applications as well as aggregate I/O time across the concurrent applications. Reduction in these values reflects reduction in wall-clock time for the applications.

D. Test-Bed

1) TestBed for Empirical Experiments: We run our experiments on the Catalyst cluster [25] at Lawrence Livermore Laboratory. Each of its compute nodes has an Intel Xeon E5-2695 v2 processor with 24 cores and an 800 GB Intel 910 local NVM. It has an InfiniBand QDR (QLogic) interconnect. We use a subset of Catalyst cluster nodes to represent a BB system and use another subset as compute nodes to run applications.

2) Parameters for Simulation Experiments: For the simulation experiments, we parameterize the simulator with values similar to those of the compute environment and BB system of Trinity. We present the system parameterization in Table I.

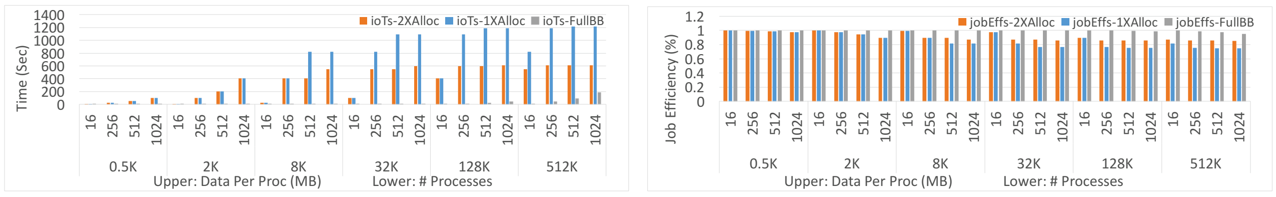
E. Simulator Validation

We are specifically interested in capturing inter-application interference during BB access. So, we will focus on capturing such interference during the validation.

To validate our model, we observe the I/O overhead for a job instance when it runs concurrently with other instances of the same job. We run a job under these situations: (1) run job alone, (2) run 2 instances together, and (3) run 4 instances together. We compare the simulation results with empirical results measured under similar configurations on our test bed, Catalyst. For both cases, the I/O overhead is:

$$\textit{I/O Overhead} = \frac{\textit{I/O Time}_{\textit{Current\_Config}}}{\textit{I/O Time}_{\textit{Alone}}}.$$

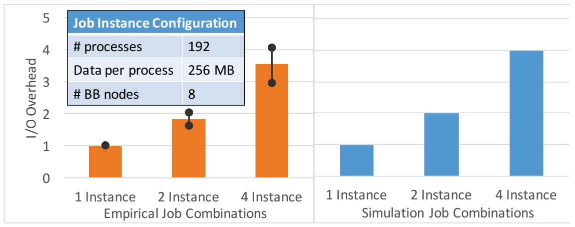
In Figure 4, we see that for both simulation and the empirical measurements, there is similar behavior for the increase in



(a) Job IO times.

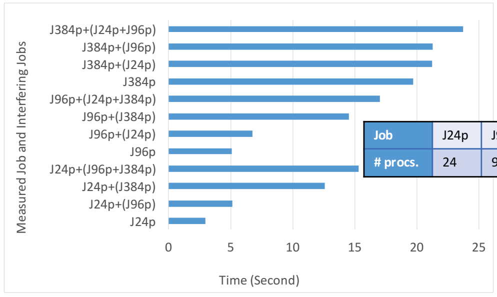
(b) Job efficiency under periodic I/O of 1 hour interval.

Fig. 3: Simulation of effect of BB allocation strategies on jobs on Trinity.



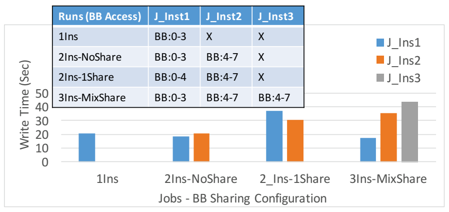
Job Instance Configuration	
# processes	192
Data per process	256 MB
# BB nodes	8

Fig. 4: Simulation and empirical results of I/O overhead for a job under interference between job instances.



Job	J24p	J96p	J384p
# procs.	24	96	384

Fig. 5: Effect of BB interference between three jobs running on Catalyst cluster.



Runs (BB Access)	J_Inst1	J_Inst2	J_Inst3
1Ins	BB:0-3	X	X
2Ins-NoShare	BB:0-3	BB:4-7	X
2Ins-1Share	BB:0-4	BB:4-7	X
3Ins-MixShare	BB:0-3	BB:4-7	BB:4-7

Fig. 6: Effect of BB interference between jobs under varying level of BB nodes sharing, on Catalyst cluster.

I/O overhead with increasing concurrency between job instances, *e.g.*, nearly $2\times$ increase in I/O overhead with $2\times$ increase in job instances. We ran five trials for the empirical measurements. Here we observe variation across different job instances. The variation is expected and arises from various factors such as contention between concurrent I/O traffic, non-constant SSD, and network performance.

The simulation performance behavior is sufficient for our experiments in this paper. It captures the case where SSD is well behaved and there is absence of measurable network overhead or interference. Here our hypothesis is that even when the BB system is well behaved, interference can occur between applications and that scheduling will be able to manage it. If we were to consider added contention and performance variability in SSDs and network, the effect of scheduling would be even higher as it can isolate I/O traffic to get rid of such interference between I/O streams of different applications.

IV. BB Performance Study

In this section, we further motivate our study of scheduling policies for BBs. First, we analyze the performance under different BB allocation policies over a range of job configurations. Then, we collect empirical results to observe the effect of inter-application interference on a BB hardware system. This also serves as further validation of our simulator of BB interference.

A. BB Allocation Policies

In this section, we analyze the effect of two allocation policies from Section II-D: the interference policy and the baseline, fully isolated policy. For the interference policy, we evaluate two cases: *1XAlloc:* select the minimum number of BB nodes that provides capacity to store a job's output; and *2XAlloc:* select BB nodes similar to *1XAlloc*, but provide double space capacity. This simulates the scenario where a job may reserve more space than it needs for a single I/O step. Next we have the baseline strategy, *FullBB:* select all BB nodes in the system for each job. In these experiments, *FullBB* is also equivalent to the bandwidth policy from Section II-D when there is no inter-job BB interference and each job can scale to all BB nodes. For this simulation, we use parameters for Trinity as described in Table I. We configure job instances using varying numbers of I/O processes and data output per process.

We can see the effect of allocation strategies in I/O time in Figure 3a. The figure shows that jobs complete their output phases significantly faster under *FullBB* compared to *2XAlloc* and *1XAlloc*. In addition, *2XAlloc* performs better compared to *1XAlloc*, *e.g.*, half the time for job with 8k processes and writing 512 MB data per process. Low I/O time is obtained when a job gets a BB allocation across a larger number of BB nodes, thus getting higher aggregated I/O bandwidth. This means that the number of BB nodes allocated varies significantly under these allocation strategies.

Figure 3b shows the efficiency for the jobs with periodic I/O phases of one hour. Periodic I/O behavior is common in HPC simulations, where the jobs write checkpoint and visualization data after a certain number of simulation steps. Here, efficiency is the measure of fraction of total time a job spends in science: *ComputeTime* / (*ComputeTime*+*IOTime*). We see that with *1XAlloc*, jobs can get low I/O efficiency, *e.g.*, 76% for a job with 32k I/O processes, and writing 512 MB per I/O process.

However, under *2XAlloc*, performance is higher in most of the cases. Here each job gets more than 85% efficiency. Similarly, under *FullBB*, we get more than 95% efficiency in all cases. The reduction in efficiency with the interference policy will be more significant — mainly under *1XAlloc* — for the jobs whose I/O steps repeat more frequently than one hour. Variation in I/O frequency across HPC jobs is a common scenario [5].

Summary for BB Allocation Policies: Three lessons are revealed. First, allocation strategies based only on capacity needs may not provide enough I/O bandwidth for jobs. Even though it has the advantage of localizing the allocation of a job to fewer BB nodes, the job may not get sufficient I/O bandwidth or performance. Second, if we look at *2XAlloc*, we see jobs get enough I/O performance under isolated execution. However, there are a limited number of BB nodes in the system, *e.g.*, 576 in the case of Trinity. This means that multiple jobs need to share BB nodes with each other and therefore interference can occur. Third, the allocation of a larger number of BB nodes (*e.g.*, *FullBB*) consistently performs better. This performance will decrease in the presence of interference, but even in presence of interference, it may still be higher that under capacity based allocation, as seen in Figure 1. We argue that, with proper management techniques, we can reduce the overhead of interference in a shared BB.

On future HPC systems, it is likely that BB allocations will be tailored to the needs of individual jobs. So, different jobs may get BB allocations striped across a varying number of BB nodes, where striping decisions may be based on various factors such as QoS needs and scalability of I/O operations. This can even change elastically at runtime to adjust the performance provided to a job. Deeper analysis of BB allocation strategies is a subject for future work. For the rest of this paper, we assume that each job will get allocations on a strict subset of BB nodes.

B. Interference in BB of Catalyst Cluster

Here, we conduct two empirical experiments. First, we use three different applications that share all BB nodes, and then we run multiple instances of one application and vary the degree of sharing of BB nodes.

For the first experiment shown in Figure 5, we use MPI jobs: J24p, J96p, and J384p, with 24, 96, and 384 processes and running on 1, 4, and 16 Catalyst nodes respectively. First we run each job alone (*none*), second we run pairs of jobs, and finally run all three jobs together (*all*). Under each configuration, the jobs concurrently write data to the BB.

In Figure 5, we observe that under concurrent access each job faces slowdown in its I/O time by varying factors under different sharing configuration. The worst slowdown is observed when all jobs are running concurrently (case all in Figure 5). The three applications experienced I/O slowdown by factors of 5.2×, 3.3×, and 1.2×, respectively. The aggregate I/O time for the three jobs increased by a factor of 2×. It is clear that the majority of the slowdown is because of sharing the SSD bandwidth between the jobs and there does not seem to be observable impact due to other factors such as contention or performance variability in SSDs and network.

In the second empirical experiment, we run three job instances, each configured with 192 processes. Each instance accesses multiple BB nodes and share a number of BB nodes with other jobs at varying levels. We present the BB allocation, sharing configuration, and results in Figure 6. We observe that interference can occur when two jobs fully or partially share the BB system (*e.g.*, both instances under *2_Ins-1Share* and instances *J_Inst2* and *J_Inst3* under *3Ins-MixShare*). But, when two jobs do not share any BB nodes (2Ins-NoShare), they did not interfere with each other, which means that network interference is not creating measurable I/O overhead for these jobs. We use these observations as a guiding factor for our sharing aware serialization algorithm, presented in Section V-B2.

Summary of BB Interference: The message from our empirical experiments on Catalyst is that BB interference can have a major impact in I/O performance with observed slowdowns as much as 5.2× in our experiments. The interference can occur under various allocation and sharing policies, *e.g.*, a bandwidth policy and also with the more realistic case of a varying number of BB nodes allocated across different jobs.

V. Proposed BB Scheduling

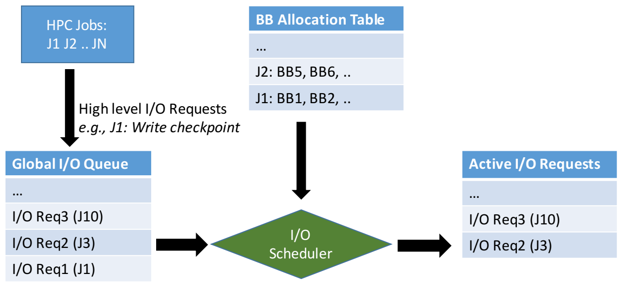


Fig. 7: BB interference management using I/O scheduling.

In this section, we propose a scheduling framework for a BB system (Figure 7). We assume a global I/O scheduler for the BB system similar to ones proposed for PFSs [4]. Applications submit access requests to the scheduler when then need to access the BB. For each application, its single process, acting as an I/O coordinator, will communicate with the scheduler to submit an I/O request. The scheduler stores the I/O requests in its global I/O queue, schedules for execution the requests present in its queue, and answers queries from applications for access token. The access token is used as a mechanism to control which application actively acesses the storage servers at a given time [4].

This global scheduler can use high-level information about the applications and system to improve its scheduling decisions. In this paper, we assume that it is aware of the BB nodes allocated to individual applications. BB allocation is done by the BB allocator software, *e.g.*, DataWarp and SLURM.

The scheduler may face performance challenges because it has to manage BB access traffic from all applications. Multiple optimizations can be applied to mitigate this challenge. In the design considered in this paper, only a single process (I/O coordinator) from each application talks to the scheduler keeping the load low. In addition, we can also implement the scheduler as distributed and hierarchical software, which will make this layer highly scalable. Such further optimizations are subject for future work.

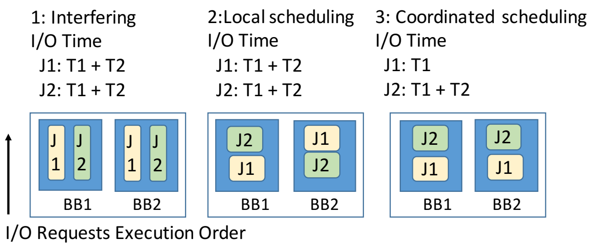


Fig. 8: Illustration- interference during BB access by two jobs, and effect of local and coordinated serialization.

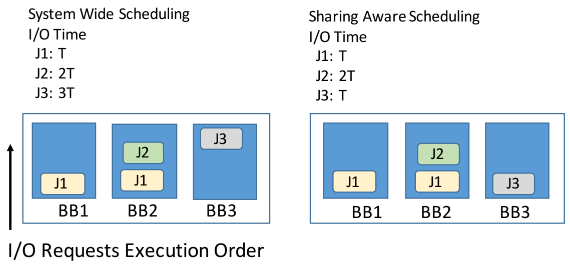


Fig. 9: Illustration- system wide and sharing aware serialization.

Next we discuss the BB access traffic control techniques, and scheduling policies we use in this paper.

A. Traffic Control and Coordination

Inter-application interference between HPC applications can be controlled effectively by using coarse grained access control [4], [24]. This means controlling high level accesses, such as an I/O step to write a complete checkpoint dataset, instead of controlling low level accesses, such as I/O requests from individual I/O processes or writing of intermediate data chunks. We utilize such coarse grain traffic control to manage BB accesses from applications.

While scheduling accesses to BB system, multiple BB nodes should be coordinated with each other to make synchronized progress for an application's I/O step (Figure 8). This will prevent I/O stragglers, *i.e.*, the few, slow I/O processes that can slowdown the whole I/O step, despite fast I/O completion by a majority of the processes. Under our coarse-grained traffic control, coordinated scheduling will be done implicitly because at any given time, all BB nodes are serving the same application.

B. Scheduling Algorithms

Now we present scheduling algorithms we use to manage BB access traffic.

1) Full Serialization Algorithm: In this paper, we first use coarse-grained serialization [4] as a technique to schedule coarse grained BB accesses. Under this scheme, each BB node serves one application at a time and allows it to complete writing all of its output data before switching to another application.

Under this algorithm, upon arrival of an application I/O step, its I/O request is stored in a global I/O queue. Then during the scheduling step of the scheduler, a single I/O request is selected from the queue for execution. Selection is based on the scheduling policy used in the system. In this paper, we use: (a) No Scheduling (NS), (b) First come first serve (FCFS) and (c) Shortest I/O job first (SJF).

In a production environment, we may relax this approach a little by allowing BB access to more than one application at a time and then control the number of concurrent applications and the fraction of total I/O bandwidth each concurrent application receives [21], [26]. In addition, scheduling is in itself a rich area. We can borrow ideas from that domain to effectively handle different situations. For example, we can apply knowledge of wait time in the I/O queue to avoid job starvation or use job's execution profile to improve its overall performance [27]. However, detailed and exhaustive analysis of alternative scheduling policies for BBs is out of scope in this paper.

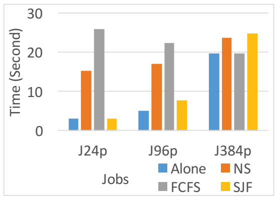
Algorithm 1: BB I/O scheduling with sharing awareness

```
   Data: PendingIOQueue, ActiveIOQueue
   Result: Updated PendingIOQueue, ActiveIOQueue
1  IOQueueCopy = PendingIOQueue ;
2  while IOQueueCopy is not empty do
3      sort PendingIOQueue using current schedl policy;
4      IOReqCand = PendingIOQueue[0] ;
5      if IOReqCand does not conflict with any request in
       SelectedIOQueue then
6          add IOReqCand to ActiveIOQueue;
7      end
8      remove IOReqCand from IOQueueCopy;
9  end
10 remove $IOReq \in ActiveIOQueue$ from
   $PendingIOQueue$;
```

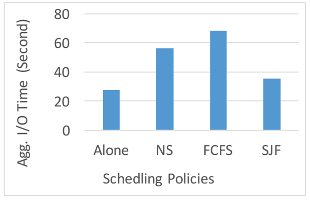
2) Sharing Aware Serialization Algorithm: We present our sharing aware serialization algorithm in Algorithm 1. This algorithm is a refinement of the full serialization algorithm. Here, we leverage the fact that different applications access different numbers of BB nodes such that an application may access only a subset of the BB nodes in the system. An application *A* will not face interference from other applications if the latter only access BB nodes other than the ones that *A* is accessing. Allowing access to such unused BB nodes can increase utilization and BB nodes, and as a result, increase system throughput (Figure 9). Here, we are making an assumption that the system has sufficient network bandwidth and interference in the network access is not significant.

VI. Empirical Evaluation

During our empirical evaluation of scheduling techniques, we use the jobs configurations J24p, J96p, and J384p, as in Section IV-B, and 8 BB nodes setup on the Catalyst cluster. We apply a system-wide full serialization algorithm, which is a good fit for these jobs since they access all BB nodes. We run the experiment once for each of the scheduling policies: NS, FCFS, and SJF. We ran five trials for each experiment and report average values.



(a) Measured data write times.



(b) Aggregate I/O time across 3 jobs.

Fig. 10: Effect of BB I/O scheduling policies for three concurrent MPI jobs running on the Catalyst cluster.

We present our result in Figure 10. Our result demonstrates that there is no beneficial performance improvement under FCFS for this experiment. Here, J384p runs faster, similar to the case of running alone, but the performance penalty for J24p and J96p are higher. Their I/O time increases by factors of 8.7× and 4.4× respectively compared to the case of running alone. This is because J384p arrives first and it writes more data compared to J24p and J96p taking longer to complete its I/O step. This means that J24p and J96p wait longer, which is evident in its I/O completion time.

SJF performs best for this workload set. Here, the I/O time for the jobs are 1.05×, 1.5× and 1.26× the time of running alone. This is a significant improvement from the interfering cases (NS), which had the I/O increase factors of 5.15×, 3.33×, and 1.2× respectively for the three jobs. The benefit is also clearly reflected in the aggregate I/O time for the jobs (Figure 10b). Under SJF, the aggregate time increased to 1.28× that under isolated runs whereas under NS, it was 2.02×. The major finding from this experiment is that with proper scheduling, the interference of a shared BB system can be reduced significantly.

VII. Simulation based Evaluation

In this section, we run simulation experiments to evaluate the effect of scheduling on BB I/O traffic on large scale HPC clusters. First, we consider the case when all the jobs in a given workload are allocated capacity across all of the BB nodes. This is a special situation that may not necessarily be common in reality. To capture a more realistic case, we conduct a second experiment where each job has its BB space allocation placed across a subset of BB nodes.

A. Simulation Configuration

We use the system configuration in Table I. We design I/O workloads that are representative of HPC I/O workloads. We present the configuration of the individual jobs in Table II and the concurrent workload mixes of these jobs we used in our experiments (Table III). Jobs in each workload mix share the BB system. The choices of workload parameters were guided by I/O characteristics of HPC jobs from supercomputers, such as described for Intrepid at Argonne National Laboratory [5]. We set the number of BB nodes in the BB system to 128 and limit compute node count to under 6000 for all the workloads.

Here, during each simulation, we take a workload set and run the I/O requests from all of its jobs. Since the jobs are not related to each other, the arrival order of their I/O requests are also not related. However, the arrival order can affect performance under

TABLE II: Jobs Configurations for Simulation Workloads

Job	# procs	# compute nodes(nCN)	data/proc	# BB Nodes	nCN per BB	data write per BB
J1	4096	128	1024 MB	128	1	32 GB
J2	16384	512	512 MB	128	4	64 GB
J3	65536	2048	128 MB	128	16	64 GB
J4	4096	128	1024 MB	16	8	512 GB
J5	4096	128	2048 MB	16	8	128 GB
J6	16384	512	512 MB	32	16	256 GB
J7	65536	2048	512 MB	64	32	512 GB

TABLE III: Workloads Configuration for Simulation.

Name	Mix w/ full stripe	Name	Mix w/ partial stripe
W1	1J1 + 1J2 + 1J3	W7	1J4 + 1J5 + 1J6 + 2J7
W2	2J1 + 2J2 + 2J3	W8	10J4 + 1J5 + 1J6 + 1J7
W3	0J1 + 2J2 + 4J3	W9	1J4 + 10J5 + 1J6 + 1J7
W4	2J1 + 4J2 + 0J3	W10	1J4 + 1J5 + 5J6 + 1J7
W5	10J1 + 2J2 +1J3	W11	0J4 + 10J5 + 0J6 + 2J7
W6	2J1 + 10J2 + 0J3	W12	10J4 + 10J5 + 2J6 + 1J7

an I/O scheduling policy, *e.g.*, FCFS, where requests are given priority based on their arrival order. To counter these effects, we randomize the I/O request arrival order across the jobs and repeat 10 trials. We report average performance across all the trials. We measure the aggregate I/O time across all jobs as the measure of performance.

B. Job Mix with Full Striping

In this experiment, we simulate execution of each workload W1 through W6 one at a time. Each job in the workloads has its BB allocation fully striped across all the 128 BB nodes. We use system wide I/O serialization and apply scheduling policies: NS, FCFS and SJF. We present our results in Figure 11.

We observe that for all workloads, aggregate I/O time is better when scheduling is applied. Even a simple policy such as FCFS beats NS. SJF performs best, *e.g.*, reduces aggregate I/O time for workload W1 by 62% compared to NS.

This tells us that scheduling can be an effective mechanism to reduce effect of interference for jobs when their BB allocation is striped across all the BB nodes and scheduling in terms of system wide coarse grained serialization of I/O steps of jobs is a good fit.

C. Job Mix with Partial Striping

For this experiment, we use workloads W7 through W12. Here, jobs in each workload are allocated a subset of the BB nodes with a higher number than that given by the interference policy and lower than that by our baseline policy from Section II-D. We use a round robin policy to assign BB nodes to the jobs. This represents a more generic and realistic BB resource

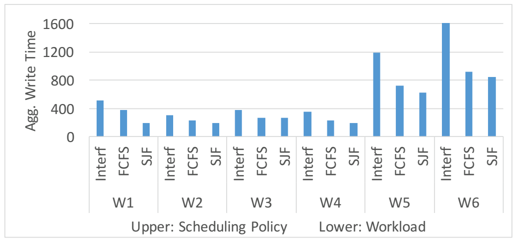


Fig. 11: Aggregate I/O time with full striping.

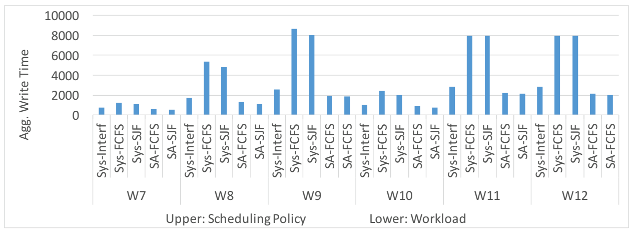


Fig. 12: Aggregate I/O time with partial striping.

allocation strategy where the number of BB nodes for each job may be decided based on various factors such as BB bandwidth requested by individual jobs, their I/O scalability, or load balancing by the resource manager. A detailed study of such an allocation model is future work.

We present our results in Figure 12. Unlike the case of full striping, we observe that aggregate I/O increases (worsens) for all scheduling policies under system wide serialization (Sys-*). Performance improves only in the case of sharing aware serialization (SA-*). We observe improved performance (reduced aggregate I/O time) for all workloads under both SA-SJF and SA-FCFS. Moreover, SA-SJF worked better with the best result of 36% improvement for W8.

Here, each job is accessing only a subset of BB nodes in the system, not all. With system-wide serialization, when a job is accessing the BB system, it is keeping idle the BB nodes that are not assigned to it. When we add sharing awareness, we serialize jobs only if they share BB nodes with each other. Jobs that do not share any BB nodes with each other run concurrently. That way we improve utilization of the BB system, and as a result, improve throughput of the BB system.

From this experiment, we observe that when allocations of jobs are striped across subsets of BB nodes in the system, sharing aware serialization is effective in scheduling the BB I/O traffic. Our conclusion is that we need to adapt scheduling algorithms to fit to the resource usage scenario of a BB system.

VIII. Related Work

BB is a new technology and therefore existing work on this topic is fairly sparse. In an early effort, Liu et. al [5] showed that the BB can act as an effective cache while writing data to the PFS. Bent et al. [28] showed that the BB can help reduce I/O jitter in HPC applications. Various efforts study software systems design for BB access [2], [11], [12]. Fang et al. [13] showed the necessity of managing the allocation of the limited life cycle of SSDs of BBs to different applications. More recent work includes BB management products DataWarp [7] from Cray and BB support from SLURM [9]. These are more relevant to us in terms of goals. These works provide mechanisms to manage resources of a shared BB system. SLURM focuses more on capacity management. On the other hand, DataWarp provides a placement based mechanism to reduce inter-application interference. It provides an optimization mode where a job gets BB capacity across fewer BB nodes, and as a result, shares BB nodes with fewer other jobs. In this work, we have shown the need for more robust mechanisms than these and investigated scheduling as one such mechanism.

HPC I/O scheduling has been studied in the context of parallel and distributed file systems and techniques have been applied at various levels of the system. In one example, Wachs, et al. [21] perform I/O time slice management across sharing jobs to manage Quality of Service (QoS) across multiple jobs. Similarly, Song, et al. [29] applied coordination across PFS servers to synchronize their time slice and serve one application at each time slice. This technique helps reduce interference in PFS and also achieve QoS in HPC I/O [26]. Inter-application interference can also be managed by using a high-level scheduler treating HPC I/O traffic in the form of coarse grained access, *e.g.*, write data during a checkpoint step. These accesses may be coordinated by using a system wide scheduler [4], [30] or by direct communication between applications [24]. A system wide I/O scheduler can utilize extra information from the HPC jobs and effectively mange I/O traffic, *e.g.*, use execution profile of applications to provide system wide optimization and global optimization to the applicaitons [27]. These techniques are relevant for a BB system as well. We have applied some of these techniques in this paper, *e.g.*, BB server coordination and coarse grained scheduling.

Management of shared resources is a rich topic and we can find relevant work in other domains as well, *e.g.*, managing access to a shared memory system [31], resource allocation in grid computing [32], and load balancing in parallel and distributed file systems [3], [33]. We may be able to borrow ideas from these areas to better solve interference and resource management problem in the BB system.

IX. Conclusion

In this paper, we investigated the problem of inter-application interference in the context of burst buffer (BB) systems. We identified that, even though BB provides much higher I/O bandwidth than PFS, and low contention storage devices, *e.g.*, SSDs, applications still suffer from interference during shared accesses. We demonstrated that we can use I/O scheduling as an effective mechanism to reduce the effect of interference of HPC jobs. We also presented a scheduling technique that was adapted to the usage model of BB, and found that it could improve I/O performance for BB systems.

In the future, we first plan to perform a more detailed investigation of scheduling BB traffic, with the goal of developing scheduling techniques specifically for BB. In addition, we also plan to investigate resources allocation techniques to effectively manage bandwidth and capacity of a shared BB system.

X. Acknowledgements

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-685595). It was also supported by the National Science Foundation under Grant No. 12292820.

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

[1] "Lustre: A Scalable, High Performance File System," http://www.lustre.org, [Online; accessed 9-Dec-2013].

[2] J. Lofstead and R. Ross, "Insights for Exascale IO APIs from Building a Petascale IO API," in *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2013, pp. 1–12.

[3] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf, "Managing Variability in the IO Performance of Petascale Storage Systems," in *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, 2010, pp. 1–12.

[4] S. Thapaliya, P. Bangalore, J. F. Lofstead, K. Mohror, and A. Moody, "IO-Cop: Managing Concurrent Accesses to Shared Parallel File System," in *ICPP Workshops*. IEEE Computer Society, 2014, pp. 52–60.

[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 *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, 2012, pp. 1–11.

[6] "Trinity-Overview," http://www.lanl.gov/projects/trinity/_assets/docs/trinity-overview-for-web.pdf, [Online; accessed 7-Sept-2015].

[7] "Cray DataWarp User Guide," http://docs.cray.com/books/S-2558-5204/S-2558-5204.pdf, [Online; accessed 1-Feb-2016].

[8] "Cori-Overview," https://www.nersc.gov/users/computational-systems/cori/, [Online; accessed 7-Sept-2015].

[9] "Slurm Burst Buffer Guide," http://slurm.schedmd.com/burst_buffer.html, [Online; accessed 6-Feb-2016].

[10] "Trinity-Burst Buffer Use Cases," https://www.nersc.gov/assets/Trinity--NERSC-8-RFP/Documents/trinity-NERSC8-use-case-v1.2a.pdf, [Online; accessed 7-Sept-2015].

[11] K. Sato, K. Mohror, A. Moody, T. Gamblin, B. R. de Supinski, N. Maruyama, and S. Matsuoka, "A User-Level InfiniBand-Based File System and Checkpoint Strategy for Burst Buffers," in *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, Chicago, IL, USA, May 26-29, 2014*, 2014, pp. 21–30.

[12] T. Wang, S. Oral, Y. Wang, B. W. Settlemyer, S. Atchley, and W. Yu, "BurstMem: A High-Performance Burst Buffer System for Scientific Applications," in *2014 IEEE International Conference on Big Data, Big Data 2014, Washington, DC, USA, October 27-30, 2014*, J. Lin, J. Pei, X. Hu, W. Chang, R. Nambiar, C. Aggarwal, N. Cercone, V. Honavar, J. Huan, B. Mobasher, and S. Pyne, Eds. IEEE, 2014, pp. 71–79.

[13] A. Fang and A. A. Chien, "How Much SSD Is Useful for Resilience in Supercomputers," in *Proceedings of the 5th Workshop on Fault Tolerance for HPC at eXtreme Scale, FTXS 2015, Portland, Oregon, USA, June 15, 2015*, N. DeBardeleben, F. Cappello, and R. L. Clay, Eds. ACM, 2015, pp. 47–54.

[14] T. Wang, S. Oral, M. Pritchard, B. Wang, and W. Yu, "TRIO: Burst Buffer Based IO Orchestration," in *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, 2015, pp. 194–203.

[15] S. Park and K. Shen, "FIOS: A Fair, Efficient Flash I/O Scheduler," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, ser. FAST'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 13–13.

[16] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng, "DataStager: Scalable Data Staging Services for Petascale Applications," in *Proceedings of the 18th ACM international symposium on High performance distributed computing*, ser. HPDC '09. New York, NY, USA: ACM, 2009, pp. 39–48.

[17] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs, "There Goes the Neighborhood: Performance Degradation Due to Nearby Jobs," in *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2013, pp. 1–12.

[18] J. F. Lofstead, I. Jimenez, and C. Maltzahn, "Consistency and Fault Tolerance Considerations for the Next Iteration of the DOE Fast Forward Storage and IO Project," in *ICPP Workshops*. IEEE Computer Society, 2014, pp. 61–69.

[19] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. 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*, Nov 2010, pp. 1–11.

[20] F. Zheng, H. Abbasi, C. Docan, J. Lofstead, Q. Liu, S. Klasky, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf, "PreDatA - Preparatory Data Analytics on Peta-Scale Machines," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, April 2010, pp. 1–12.

[21] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger, "Argon: Performance Insulation for Shared Storage Servers," in *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, ser. FAST '07. Berkeley, CA, USA: USENIX Association, 2007, pp. 5–5.

[22] S. Lee, T. Kim, K. Kim, and J. Kim, "Lifetime Management of Flash-Based SSDs using Recovery-Aware Dynamic Throttling," in *Proceedings of the 10th USENIX conference on File and Storage Technologies, FAST 2012, San Jose, CA, USA, February 14-17, 2012*, W. J. Bolosky and J. Flinn, Eds. USENIX Association, 2012, p. 26.

[23] H. Shan, K. Antypas, and J. Shalf, "Characterizing and Predicting the I/O Performance of HPC Applications Using a Parameterized Synthetic Benchmark," in *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2008, pp. 1–12.

[24] M. Dorier, G. Antoniu, R. B. Ross, D. Kimpe, and S. Ibrahim, "CALCioM: Mitigating I/O Interference in HPC Systems through Cross-Application Coordination," in *28th IEEE International Parallel & Distributed Processing Symposium*, Phoenix, AZ, 2014.

[25] "Catalyst Supercomputer Specs." https://computing.llnl.gov/?set=resources&page=OCF_resources, [Online; accessed 7-Sept-2015].

[26] X. Zhang, K. Davis, and S. Jiang, "QoS Support for End Users of I/O-Intensive Applications Using Shared Storage Systems," in *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2011, pp. 1–12.

[27] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Y. Robert, and M. Snir, "Scheduling the I/O of HPC Applications under Congestion," INRIA, Rapport de recherche RR-8519, Apr. 2014.

[28] J. Bent, S. Faibish, J. Ahrens, G. Grider, J. Patchett, P. Tzelnic, and J. Woodring, "Jitter-Free Co-Processing on a Prototype Exascale Storage Stack," in *MSST*. IEEE Computer Society, 2012, pp. 1–5.

[29] H. Song, Y. Yin, X. H. Sun, R. Thakur, and S. Lang, "Server-Side I/O Coordination for Parallel File Systems," in *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2011, pp. 1–11.

[30] S. Thapaliya, A. Moody, K. Mohror, and P. Bangalore, "Poster: Inter-application Coordination for Reducing I/O Interference," in *SC '13*, LLNL-POST-641538.

[31] M. Xie, D. Tong, K. Huang, and X. Cheng, "Improving System Throughput and Fairness Simultaneously in Shared Memory CMP Systems via Dynamic Bank Partitioning," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, Feb 2014, pp. 344–355.

[32] M. B. Qureshi, M. M. Dehnavi, N. Min-Allah, M. S. Qureshi, H. Hussain, I. Rentifis, N. Tziritas, T. Loukopoulos, S. U. Khan, C.-Z. Xu, and A. Y. Zomaya, "Survey on Grid Resource Allocation Mechanisms," *J. Grid Comput*, vol. 12, no. 2, pp. 399–441, 2014.

[33] P. Scheuermann, G. Weikum, and P. Zabback, "Data Partitioning and Load Balancing in Parallel Disk Systems," *The VLDB Journal*, vol. 7, no. 1, pp. 48–66, Feb. 1998.