TRIO: Burst Buffer Based I/O Orchestration

Teng Wang† Sarp Oral‡ Michael Pritchard† Bin Wang† Weikuan Yu†
Auburn University† Oak Ridge National Laboratory‡
Auburn University, AL 36849 Oak Ridge, TN 37831
{tzw0019,mjp0009,bwang,wkyu}@auburn.edu {oralhs}@ornl.gov

***Abstract*—The growing computing power on leadership HPC systems is often accompanied by ever-escalating failure rates. Checkpointing is a common defensive mechanism used by scientific applications for failure recovery. However, directly writing the large and bursty checkpointing dataset to parallel filesystem can incur significant I/O contention on storage servers. Such contention in turn degrades the raw bandwidth utilization of storage servers and prolongs the average job I/O time of concurrent applications. Recently burst buffer has been proposed as an intermediate layer to absorb the bursty I/O traffic from compute nodes to storage backend. But an I/O orchestration mechanism is still desired to efficiently move checkpointing data from bursty buffers to storage backend. In this paper, we propose a burst buffer based I/O orchestration framework, named TRIO, to intercept and reshape the bursty writes for better sequential write traffic to storage severs. Meanwhile, TRIO coordinates the flushing orders among concurrent burst buffers to alleviate the contention on storage server bandwidth. Our experimental results reveal that TRIO can deliver 30.5% higher bandwidth and reduce the average job I/O time by 37% on average for data-intensive applications in various checkpointing scenarios.**

I. INTRODUCTION

More complex natural systems such as weather forecasting and earthquake prediction are being simulated on large-scale supercomputers with a colossal amount of hardware and software components. The unprecedented growth of system components results in an ever-escalating failure rate. According to a survey conducted on the 100,000-node BlueGene/L system at Lawrence Livermore National Laboratory (LLNL) in 2009, the system experienced a hardware failure every 7-10 days [15]. As a common defensive mechanism for failure recovery, checkpointing dominates 75%-80% of the I/O traffic on current High Performance Computing (HPC) systems [25, 7].

Though checkpointing is necessary for fault tolerance, it can introduce significant overhead. For instance, a study from Sandia National Laboratories predicts that a 168-hour job on 100,000 nodes with Mean Times Between Failures (MTBF) of 5 years will spend 65% of its time in checkpointing [14]. A major reason for such high overhead is that during checkpointing applications usually issue tens of thousands of concurrent write requests to the underlying parallel filesystem (PFS). Since the number of compute nodes is typically 10x$\sim$100x more than those on storage systems [23, 20], the excessive write requests to each server incur heavy contention, which raises two performance issues. First, when competing I/O requests exceed the capabilities of each storage server, its bandwidth will degrade [16]. Second, when storage servers are competed by multiple jobs, checkpointing for mission-critical jobs can be frequently interrupted by low-priority jobs. I/O requests from small jobs may also be delayed due to concurrent accesses from large jobs, prolonging the average I/O time [13].

Previous efforts to mitigate I/O contention generally fall into two categories: client-side and server-side optimizations. Client-side optimizations mostly resolve I/O contention in a single application, by buffering the dataset in staging area [16, 21] or optimizing application's I/O pattern [11]. Server-side optimizations generally embed their solutions inside the storage server, overcoming issues of contention by dynamically coordinating data movement among servers [29, 12, 32].

Recently, the idea of Burst Buffer (BB) is proposed as an additional layer with fast memory devices such as DRAM and SSDs for bursty I/O from compute applications [17]. Many consider it as a promising solution of I/O contention for next-generation computing platforms. With the mediation of BB, applications can directly dump their large checkpoint datasets to BB and minimize their direct interactions with PFS. BB can flush the data to PFS at a later point of time. However, existing solutions generally consider BBs as a reactive intermediate layer to avoid applications' direct interactions with PFS, the issues of contention still remain when checkpointing dataset is flushed from BBs to PFS. Therefore, a proactive BB orchestration framework that mitigates the contention on PFS carries great significance. Compared with the client-side optimization, an orchestration framework on BB is able to coordinate I/O traffic between different jobs, mitigating I/O contention at a larger scope. Compared with the server-side optimization, an orchestration framework on BB can free storage servers from the extra responsibility of handling I/O contention, making it highly portable to other PFSs.

In this work, we propose TRIO, a burst buffer orchestration framework, to efficiently move large checkpointing dataset to PFS. It is accomplished by two component techniques: Stacked AVL-Tree Based Indexing (STI) and Contention-Aware Scheduling (CAS). STI organizes the checkpointing write requests inside each BB according to their physical layout among storage servers and assist data flush operation with enhanced sequentiality. CAS orchestrates all BB's flush operations to mitigate I/O contention. Taken together, our contributions are three-fold.

- We have conducted a comprehensive analysis on two issues that are associated with checkpointing operations in HPC systems, i.e., degraded bandwidth utilization of storage servers and prolonged average job I/O time.

Time(s)	BTIO	MPI-TILE-IO	IOR	AVG	TOT
MultiWL	41	121.83	179.75	114.19	179.75
SigWL	9.79	72.28	161	81.02	161

TABLE I: The I/O Time of individual benchmarks when they are launched concurrently (*MultiWL*) and serially (*SigWL*).

- Based on our analysis, we propose TRIO to orchestrate applications' write requests that are buffered in BB for enhanced I/O sequentiality and alleviated I/O contention.
- We have evaluated the performance of TRIO using representative checkpointing patterns. Our results reveal that TRIO is able to deliver 30.5% higher I/O bandwidth and reduce average job I/O time by 37%.

The rest of this paper is organized as follows. Section II experimentally analyzes the major issues restricting applications' I/O performance. Section III presents the design of TRIO. Section IV systematically evaluates the benefits of TRIO. Related work and conclusion are discussed in Section V and Section VI.

II. Motivation

In this section, we experimentally study two issues resulting from I/O contention, namely, prolonged average job I/O time and degraded storage server bandwidth utilization.

A. Experimental Environment

Testbed: Our study was conducted on the Titan supercomputer [4] at Oak Ridge National Laboratory (ORNL). Each compute node is equipped with a 16-core 2.2GHZ AMD Opteron 6274 (Interlagos) processor and 32 GB of RAM. These nodes are connected to each other via Gemini high-speed interconnect. Spider II filesystem [24] serves as the backend storage system for Titan. It is composed of two Lustre-based filesystems: Atlas1 and Atlas2, which provide 30 PB of storage space and 1 TB/s of aggregated bandwidth. Files are striped across multiple object storage targets (OSTs) using the default stripe size of 1 MB and stripe count of four.

Benchmarks: For the performance examination of PFS, we used multi-job workloads composed of IOR [19], MPI-Tile-IO, and BTIO [31]. IOR is a flexible benchmarking tool capable of emulating diverse I/O access patterns. It was initially designed for measuring the I/O performance of parallel filesystems. MPI-Tile-IO utilizes a common workload wherein multiple processes concurrently access a dense two-dimensional dataset using MPI-IO. BTIO is derived from computational fluid dynamics applications. It produces a block-tridiagonal partitioning pattern on a three-dimensional array across a square number of processes. Each process dumps multiple Cartesian subsets of the entire data set during its I/O phase. IOR and MPI-Tile-IO are configured to use N-1 checkpointing pattern in which multiple processes write to a non-overlapping, contiguous extent of a shared file, while BTIO is configured to follow the N-N checkpointing pattern that many processes each write to one of many separate files.

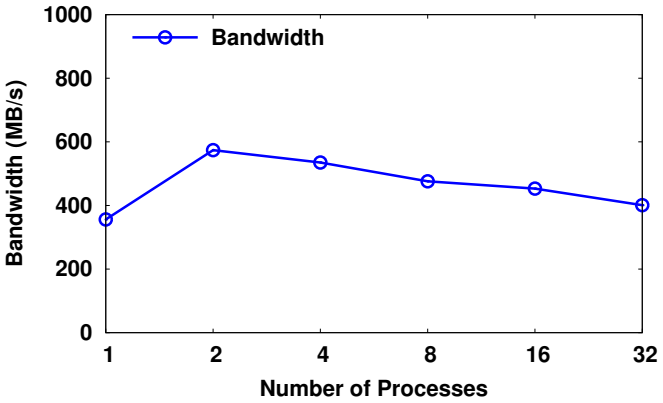

Fig. 1: The impact of increasing number of concurrent processes to the bandwidth of a single OST.

We dedicate 4, 8, and 16 nodes to run BTIO, MPI-Tile-IO and IOR, respectively. In accordance with their requirements on minimum process counts, 16 processes were launched on each node for both BTIO and MPI-Tile-IO, while one process per node was launched for IOR.

B. Prolonged I/O Time under Contention

In general, PFS services I/O requests in a timely manner, i.e., in a First-Come-First-Serve (FCFS) order, which results in undifferentiated I/O service to concurrent jobs. This undifferentiated I/O service can lead to prolonged I/O time. To emulate its impact in multi-job environment, we run BTIO, MPI-Tile-IO and IOR concurrently but differentiate their output data sizes as 13.27GB, 64GB and 128GB respectively. This launch configuration is referred to as *MultiWL*. Competition for storage is assured by striping all benchmark output files across the same four OSTs. We compare their job I/O time with that when these benchmarks are launched in a serial order, which is referred to as *SigWL*.

The I/O time of the individual benchmarks are shown in Table I as three columns, BTIO,MPI-TILE-IO and IOR, respectively. The average and total I/O time of the three benchmarks are shown in columns *AVG* and *TOT*. As we can see, the average I/O time of MultiWL is 1.41× longer than SigWL. This is because a job's storage service is affected by the contention from other concurrent jobs. And the contention from large jobs can significantly delay the I/O of small jobs. In our tests, the most affected benchmark is BTIO, which generates the smallest workload, its I/O time in MultiWL is 4.18× longer than SigWL.

C. Degraded Bandwidth Utilization Due to Contention

On the storage server side, the aforementioned I/O contention can degrade the effective bandwidth utilized by applications. A key reason for contention is that each process can access multiple OSTs, and each OST is accessible from multiple processes. Such N-N mapping poses two challenges: first, each OST suffers from the competing accesses from multiple processes; second, since the requests of each process are distributed to multiple OSTs instead of one, each process

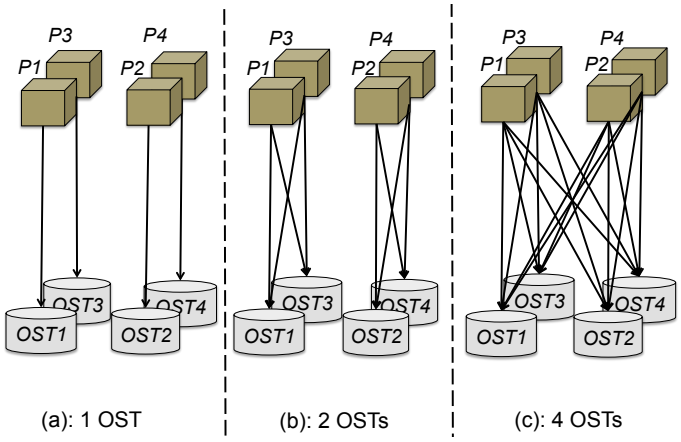

Fig. 2: Scenarios when individual writes are distributed to different number of OSTs. "N-OST" means that each process's writes are distributed to N OSTs.

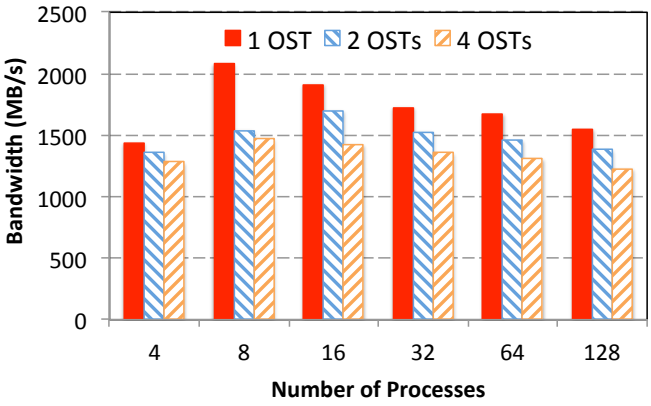

Fig. 3: Bandwidth when individual writes are distributed to different number of OSTs.

is involved in the competition for more than one OST. We use IOR benchmark to analyze the impacts of both challenges. Similar to previous experiment, we place one IOR process on each compute node.

To investigate the first challenge, we use an increasing number of processes to concurrently write in total 32GB data to a single OST. The result is exhibited in Fig. 1. The bandwidth first increases from 356MB/s with 1 process to 574MB/s with 2 processes, then decreases constantly to 401MB/s with 32 processes, resulting in 30.1% bandwidth degradation from 2 to 32 processes. The improvement from 1 to 2 processes is because the single-process I/O stream is not able to saturate OST bandwidth. Specifically, each OST is organized as RAID-6 arrays, yielding a raw bandwidth of 750MB/s [24]. When an I/O request is issued, it is relayed multiple hops from peer compute nodes to I/O router, then goes through SION network, Object Storage Server (OSS) and eventually arrives at OST. Despite of the high network bandwidth along the critical path, the extra data copy and data processing overhead at each hop cause additional delays. On top of these factors, in this experiment, the bandwidth utilization is 75.6% when there are only two concurrent processes, but drops to 53.5% when there are 32 processes.

Our intuition suggests that contention from 2 to 32 processes can incur heavy disk seeks; however, our lack of privilege to gather I/O traces at the kernel level on ORNL's Spider filesystem prevents us from directly proving our intuition. We repeat our experiments on our in-house Lustre filesystem (running the same version as that on Spider) and observe that up to 32% bandwidth degradation are caused by I/O contention. By analyzing I/O traces using blktrace [1], we find that disk access time accounts for 97.1% of the total I/O time on average. This information indicates that excessive concurrent accesses to OSTs can degrade the bandwidth utilization.

To emulate the second challenge, we distribute each IOR's write requests to multiple OSTs. In our experiment, we spread the write requests from each process to 1, 2, 4 OSTs, which are presented in Fig. 2 as 1 OST, 2 OSTs, 4 OSTs, respectively. We fix the total data size as 128GB, the number of utilized OSTs as 4, and measure the bandwidth under the three scenarios using a varying number of processes. The result is demonstrated in Fig. 3. The scenario of 1 OST consistently delivers the highest bandwidth with the same number of processes, outperforming 2 OSTs and 4 OSTs by 16% and 26% on average, respectively. This is because, by localizing each process's writes on 1 OST, each OST is competed by fewer processes. Another interesting observation is that the bandwidth under the same scenario (e.g. 1 OST) degrades with the increasing number of processes. This trend can be explained by the impact of first issue as we measured in Fig. 1.

Based on our characterization, under a contentious environment where numerous processes concurrently access a smaller number of OSTs, bandwidth can be more efficiently utilized by localizing each process's access on one OST, and scheduling a proper number of processes to access each OST.

III. TRIO: A Burst Buffer Based Orchestration

The aforementioned two I/O performance issues result from direct and eager interactions between applications and storage servers. Many computing platforms, such as the supercomputer Tianhe-2 [8] and the two future generation supercomputers Coral [2] and Trinity [9], have introduced Burst Buffer (BB) as an intermediate layer to mitigate these issues. Buffering large checkpoint dataset in BB gives more visibility to the I/O traffic, which provides a chance to intercept and reshape the pattern of I/O operations on PFS. However, existing works generally use BB as a middle layer to avoid applications' direct interaction with PFS [17], few works [30] focus on the interaction between BB and PFS, i.e. how to orchestrate I/O so that intensive datasets can be efficiently flushed from BB to PFS. To this end, we propose TRIO, a burst buffer-based orchestration framework, to coordinate the I/O traffic from compute nodes to BB and to storage servers. In the rest of the section, we first highlight the main idea of TRIO through a comparison with a reactive data flush approach for BB management; then we detail two key techniques in

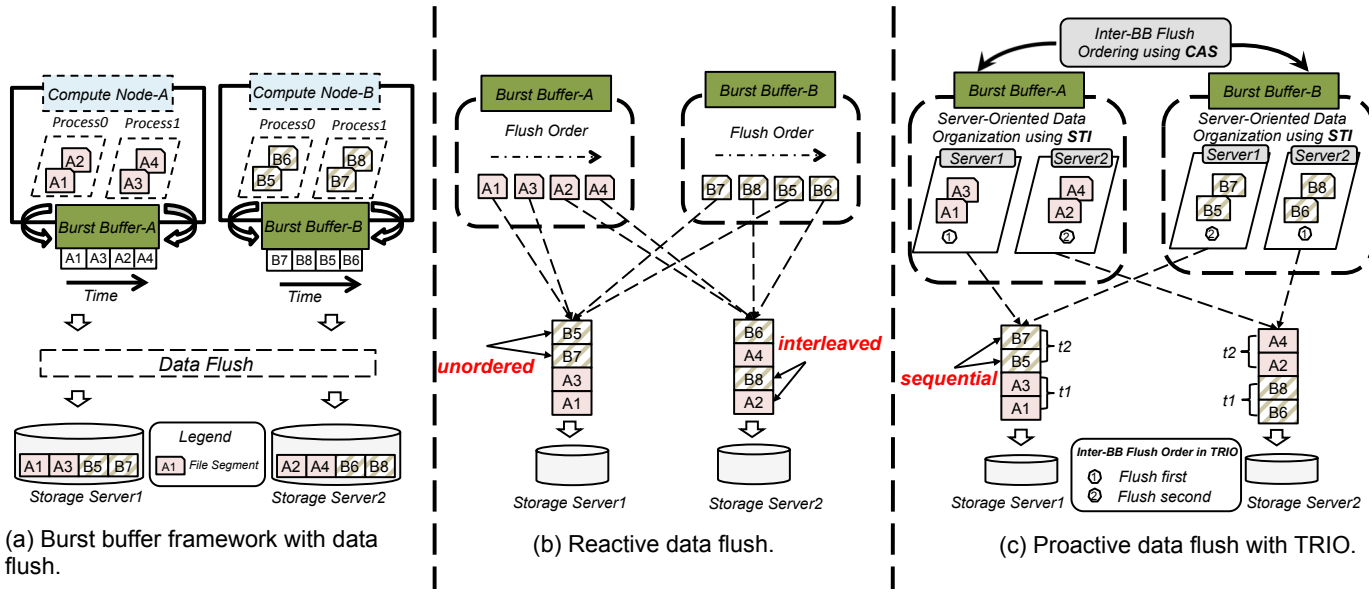

Fig. 4: A conceptual example comparing TRIO with reactive data flush approach. In (b), Reactive data flush incurs unordered arrival (e.g. B7 arrives earlier than B5 to Server1) and interleaved requests of BB-A and BB-B. In (c), *Server-Oriented Data Organization* increases sequentiality while *Inter-BB Flush Ordering* mitigates I/O contention.

Section III-B and Section III-C.

A. Main Idea of TRIO

Fig. 4(a) illustrates a general framework of how BB interacts with PFS. On each Compute Node (CN), 2 processes are checkpointing to a shared file that is striped over 2 storage servers. A1, A2, A3, A4, B5, B6, B7 and B8 are contiguous file segments. These segments are first buffered on the BB located on each CN during checkpointing, then flushed from BB to two storage servers on PFS.

An intuitive strategy for the BB is to reactively flush the datasets to the PFS as they arrive at the BB. Fig. 4(b) shows the basic idea of such a reactive approach. However, reactive approach has two drawbacks. First, directly flushing the *unordered* segments from each BB can degrade the chance of sequential writes (We refer to this chance as *sequentiality*). In this figure, segments B5 and B7 are contiguously laid out on storage server 1, but arrive at BB-B out of order. Due to reactive flushing, B7 will be flushed earlier than B5, losing the opportunity to retain sequentiality. Second, it suffers from the same server-side contention resulting from N-N mapping. As indicated by this figure, BB-A and BB-B concurrently flush A2 and B8 to Server2, so the two segments are *interleaved*. This will degrade the bandwidth with frequent disk seeks. In a multi-job environment, segments to a storage server come from files of different jobs. Interleaved accesses to storage servers can prolong the average job I/O time and delay the timely service for mission-critical and small jobs.

In contrast to our analysis, we propose a proactive data flush framework, named TRIO, to address these two drawbacks. Fig. 4(c) gives an illustrative example of how it enhances the sequentiality in flushed data stream and mitigates contention on storage server side. Before flushing data, TRIO follows a server-oriented data organization to group together segments to each storage server and establishes an intra-BB flushing order based on their offsets in the file. This is realized through a server-oriented and stacked AVL-Tree based indexing (STI) technique, which is elaborated in Section III-B. In this figure, B5 and B7 in BB-B are organized together and flushed sequentially, which enhances sequentiality on Server 2. However, contention arises when both BB-A and BB-B flush to the same servers. TRIO addresses this problem using a second technique, Contention-Aware Scheduling (CAS), which is discussed in Section III-C. CAS establishes an inter-BB flushing order that specifies which BB should flush to which server each time. In this simplified example, BB-A flushes its segments to Storage Server 1 and Storage Server 2 in sequence, while BB-B flushes to Storage Server 2 and Storage Server 1 in sequence. In this way, during the time periods t1 and t2, each server is accessed by a different BB, avoiding contention. More details about these two optimizations are discussed in the rest of this section.

B. Server-Oriented Data Organization via Stacked AVL-Tree Based Indexing

As mentioned earlier, directly flushing unordered segments to PFS can degrade I/O sequentiality on servers. Many state-of-the-art storage systems apply tree-based indexing [27, 26] to increase sequentiality. These storage systems leverage conventional tree structures (e.g. B-Tree) to organize file segments based on their locations on the disk. Sequential writes can be enabled by in-order traversal of the tree.

Although it is possible to organize all segments in BB using a conventional tree structure (e.g. indexing only by offset), it will result in a flat metadata namespace. This cannot satisfy the complex semantic requirements in TRIO. For instance, each

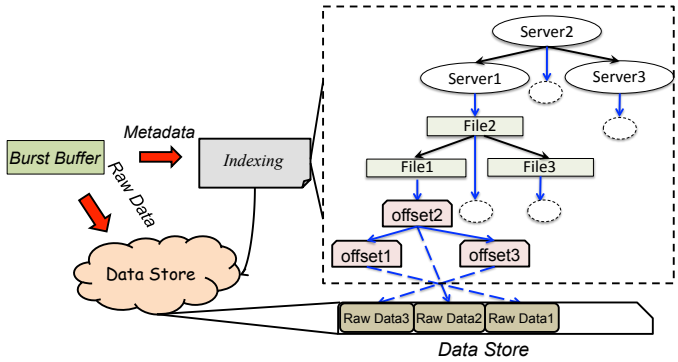

Fig. 5: Server-Oriented Data Organization with Stacked AVL Tree. Segments of each server can be sequentially flushed following in-order traversal of the tree nodes under this server.

BB needs to sequentially flush all segments belonging to a given storage server each time. Since these segments can come from different files, it is necessary to group together segments belonging to the same file and then sequentially flush segment groups (i.e., files). A conventional tree structure requires a full tree traversal to retrieve all the segments belonging to a given server and group these segments for different files.

We introduce a technique called Stacked AVL-Tree Based Indexing to address these requirements. Like many other conventional tree structures, the AVL-tree is a self-balancing tree that supports lookup, insertion and deletion in logarithmic complexity. It can also deliver an ordered node sequence following an in-order traversal of all nodes in the tree. STI differs in that all the tree nodes are organized in a stacked manner, based on the fact that each server hosts segments from multiple files, and each file contains multiple segments starting at different offsets. As shown in Fig. 5, this example of stacked AVL-tree enables two semantics: sequentially flushing all segments of a given file (e.g., offset1, offset2, and offset3 of File1), and sequentially flushing all files in a given server (e.g., File1, File2, and File3 of Server1). The semantic of server-based flushing is stacked on top of the semantic of file-based flushing. STI is also extensible for new semantics (e.g. flushing all segments under a given timestamp) by inserting a new layer (e.g. timestamp) in the stacked AVL-tree.

The stacked AVL-tree of each BB is dynamically built during runtime. When a file segment arrives at BB, three types of metadata that uniquely identify this segment are extracted: server ID, file name, and offset. BB first looks up the first layer (e.g. the layer of server ID in Fig. 5) to check if the server ID already exists (it may exist if another segment belonging to the same server has already been inserted). If not, a new tree node is created and inserted in the first layer. Similarly, its file name and offset are inserted in the second and third layers. Once the offset is inserted as a new tree node in the third layer (there is no identical offset under the same file because of the append-only nature of checkpointing), this tree node is associated with a pointer that points to the raw data of this segment in data store.

With this data structure, each BB can sequentially issue all segments belonging to a given storage server by in-order traversal of the subtree rooted at the server node. For instance, flushing all segments to Server1 in Fig. 5 can be accomplished by traversing the subtree of the node "Server1", sequentially retrieving and writing the raw data of all segments (e.g. raw data pointed by offset1, offset2, offset3) of all the files (e.g. file1, file2, file3). Once all the data in a given server is flushed, all the tree nodes belonging to this server are trimmed from this tree.

Our current design for data flush is based on a general BB use case. That is, after an application finishes one or multiple rounds of computation, it dumps the checkpointing dataset to BB, and begins next round of computation. Though we use a proactive approach in reshaping the I/O traffic inside BB, flushing checkpointing data to PFS is still driven by the demand from applications. After flushing, storage space on BB will be reclaimed entirely. We leave it as our future work to investigate a more aggressive and automatically triggering mechanism for flushing inside the burst buffer.

C. Inter-BB Ordered Flush via Contention-Aware Scheduling

Server-oriented organization enhances sequentiality by allowing each BB to sequentially flush all file segments belonging to one storage server each time. However, contention can arise when multiple BBs flush to the same storage server. For instance, in Fig. 4(c), contention on Storage Server 2 can happen if BB-A and BB-B concurrently flush their segments belonging to Storage Server 2 without any coordination, leading to multiple concurrent I/O operations at Storage Server 2 within a short period. We address this problem by introducing a technique called Contention-Aware Scheduling (CAS). CAS orders all BBs' flush operations to minimize competitions for each server. For instance, in Fig. 4(c), BB-A flushes to Server1 and Server2 in sequence, while BB-B flushes to Server2 and Server1 in sequence. This ordering ensures that, within any given time period, each server is accessed only by one BB. Although the flushing order can be decided statically before all BBs starts flushing, this approach needs all BBs to synchronize before flushing and the result is unpredictable under real-world workloads. Instead, CAS follows a dynamic approach, which adjusts the order during flush in a bandwidth-aware manner.

1) Bandwidth-Constrained Data Flushing: In general, each storage server can only support a limited number of concurrent BBs flushing before its bandwidth is saturated. In this paper, we refer to this threshold as α, which can be measured via offline characterization. For instance, our experiment in Fig. 1 of Section II reveals that each OST on Spider II is saturated by the traffic from 2 compute nodes; thus, setting α to 2 can deliver maximized bandwidth utilization on each OST. Based on this bandwidth constraint, we propose a Bandwidth-aware Flush Ordering (BFO) to dynamically order the flush operations of each BB so that each storage server is being used by at most α BBs. For instance, in Fig. 4, BB-A buffers segments of Server1 and Server2. Assuming $\alpha = 1$, it needs to select a server that has not been assigned to any BB. Since BB-

B is flushing to Server2 at time t1, BB-A picks up Server1 and flushes the corresponding segments (A1, A3) to this server. By doing so, the contention on Server1 and Server2 are avoided and consequently the two servers' bandwidth utilization is maximized.

A key question is how to get the usage information of each server. BFO maintains this information via an arbitrator located on one of the compute nodes. When a BB wants to flush to one of its targeted servers, it sends a flushing request to arbitrator. This request contains several pieces of information about this BB, such as its job ID, job priority, and utilization. The arbitrator then selects one from the targeted servers being used by fewer than α BBs, returns its ID to BB, and increases the usage of this server by 1. The BB then starts flushing all its data to this server. After flushing, it requests to flush to other targeted servers. Arbitrator then decreases the usage of the old server by 1 and assigns another qualified server to this BB. When there is no qualified BB, it temporarily queues the BB's request.

2) Job-Aware Scheduling: In general, compute nodes greatly outnumber storage servers, so there may be multiple BBs queued to the same storage server for flushing. When this storage server becomes available, the arbitrator needs to assign this storage server to a proper BB. A naive approach to select a BB would be to follow FCFS. Since each BB is allocated to one job along with its compute node, treating BBs equally can delay service of critical jobs, and prolong job I/O time of small jobs. Instead, the arbitrator categorizes BBs based on their job priorities and job sizes. It prioritizes the service for BBs of high-priority jobs, including those that are important at the beginning, or the ones that have higher criticality (e.g. the usages of some BB in this job reach their capacity). Among BBs with equal priority, it selects the one belonging to the smallest jobs (e.g. jobs with smallest checkpointing data size) to reduce average job I/O time.

Sometimes multiple servers may be available to serve a BB's request. Since the checkpointing datasets of each job are generally evenly striped over multiple storage servers, the arbitrator selects the server least utilized by this BB's job (i.e. size of data that has been already flushed to this server is the smallest). This can balance server utilization for this job and minimize wait time for the last server to complete its service. When the available servers are equally utilized, the arbitrator selects the server being utilized by the least number of processes to balance BBs' concurrent accesses to each server.

IV. Experiment Evaluation

For evaluation, we implemented an initial prototype of the TRIO framework with the support of Stacked AVL-Tree Based Indexing (STI) and Contention-Aware I/O Scheduling (CAS). Our experiments were conducted on the Titan supercomputer [4], which uses Spider II as the backend Lustre filesystem. More details on this platform can be found in Section II. Of the 32GB memory on each compute node, we allocated 16GB to the TRIO client and reserved 16GB for BB. We used IOR [28] as a representative benchmark for performance evaluation. IOR is a flexible benchmarking tool that is able to generate diverse I/O patterns. Each experiment was executed 15 times, and we took the median value as the result.

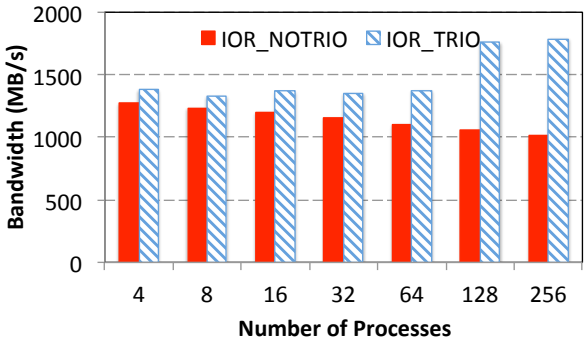

Fig. 6: The overall performance of TRIO under both inter-node and intra-node I/O traffic.

As discussed in Section III-C, CAS mitigates contention by restricting the number of concurrent BBs flushing to each storage server to α. In all our experiments, we set α to 2, thus limiting the number of BBs on each OST to at most two. This was in accordance with our characterization in Section II.

A. Overall Performance of TRIO

Fig. 6 demonstrates the overall performance of TRIO under competing workloads with an increasing number of IOR processes. We evaluated TRIO by first having all processes on the same compute node copy their datasets to burst buffer space, then using TRIO to coordinate the data flush to PFS. This configuration is shown in Fig. 6 as IOR_TRIO. We compared the aggregated OST bandwidth under this configuration with the configuration that used the same number of IOR processes to directly write their datasets to PFS (referred to as IOR_NOTRIO in Fig. 6). In both configurations, all processes wrote to a shared file striped over 4 OSTs. Each process wrote a non-overlapping, contiguous segment of this shared file. We initially used 4 processes to write to the 4 OSTs, and placed each process on 1 compute node. Then gradually increased the number of processes in each compute node. Once each node had 16 processes, additional nodes were used to host the rest of processes (e.g. 8 and 16 compute nodes were used for 128 and 256 processes in Fig. 6).

As we can see from Fig. 6, bandwidth of IOR_NOTRIO dropped with increasing number of processes involved. This was due to the exacerbated contention from both intra-node and inter-node I/O traffic. By contrast, IOR_TRIO demonstrated much more stable performance by optimizing intra-node traffic using STI and inter-node I/O traffic using CAS. The lower bandwidth observed with fewer than 64 processes was due to OST bandwidth not being fully utilized (4 BBs were used to flush to 4 OSTs in these cases). Overall, TRIO improved I/O bandwidth by 30% on average by coordinating I/O traffic.

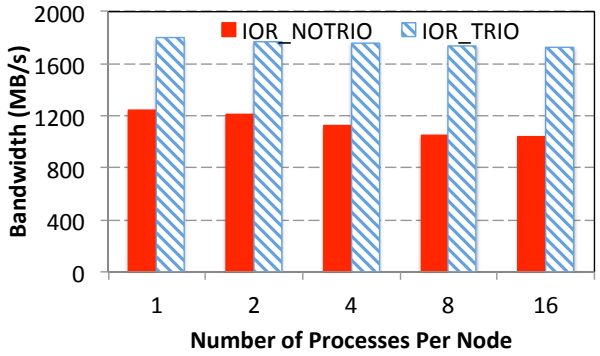

Fig. 7: Bandwidth of TRIO under varying intra-node traffic. Intra-node I/O traffic is amplified by fixing the number of compute nodes and increasing the number of processes per node.

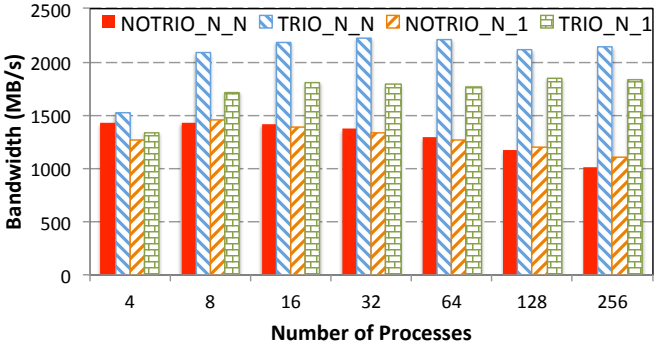

Fig. 8: The bandwidth of TRIO under I/O contention with increasing number of processes.

B. Benefit of STI in Optimizing Intra-Node I/O Traffic

To evaluate the benefit of STI in optimizing intra-node I/O traffic, we fixed the number of compute nodes utilized as 16 (same as 256-process in Fig. 6), then increased the number of processes launched on each node from 1 to 16. Fig. 7 compared the bandwidth of IOR_NOTRIO and IOR_TRIO.

As we can see from this figure, a lack of intra-node I/O traffic coordination resulted in the bandwidth of IOR_NOTRIO constantly degrading as the number of processes on each node increased. This performance degradation resulted from the destroyed sequentiality by the mixed workload. In contrast, IOR_TRIO demonstrated stable performance. Sequentiality was maintained by reordering all write requests using STI. Overall, IOR_TRIO delivered 66% higher bandwidth than IOR_NOTRIO with 16 processes on each compute node. Optimizing the intra-node I/O traffic accounted for 16.4% of the bandwidth improvement.

C. Avoiding Bandwidth Degradation under Contention using CAS

We evaluated TRIO's ability to mitigate I/O contention under two dominating checkpointing patterns: N-1 and N-N. In N-1 pattern, all processes write to a shared file. In N-N pattern, each process writes a separate file. We placed 1 IOR process on each compute node. This allowed us to focus on the evaluation of CAS without consideration for intra-node I/O traffic.

CAS's support for the two checkpointing patterns was evaluated by having each IOR process dump a 16GB dataset to its local BB and using TRIO to flush these datasets to the PFS. Such configurations for N-1 and N-N are referred to as TRIO-N-1 and TRIO-N-N respectively. For comparison, we had each IOR process dump its 16GB in-memory data directly to the underlying PFS; Such configurations for the two patterns are referred to as NOTRIO-N-1 and NOTRIO-N-N respectively. For the N-1 case, each IOR process wrote on a non-overlapping, contiguous extent of a shared file striped over 4 OSTs. For N-N, each IOR process wrote a separate file. Contention for N-N was assured by striping files over the same 4 OSTs.

Fig. 8 reveals the bandwidth of both TRIO and NOTRIO with an increasing number of IOR processes. In N-1 case, the bandwidth of TRIO first grew from 1.3GB/s with 4 processes to 1.7GB/s with 8 processes, then stabilized around 1.8GB/s with more processes. The lower performance with 4 processes was because CAS placed one BB on each of the 4 OSTs for balanced workload, I/O traffic from one BB was not able to saturate OST bandwidth. The stable performance with more than 8 processes occurred because TRIO scheduled 2 concurrent BBs on each OST. Therefore, even under heavy contention, each OST was being used by 2 BBs that consumed most OST bandwidth. In contrast, the bandwidth of NOTRIO peaked at 1.46GB/s with 8 processes, then dropped to 1.1GB/s with 256 processes. This accounted for only 60% of the bandwidth delivered by TRIO with 256 processes. This bandwidth degradation resulted from the contention generated by larger numbers of processes. Overall, by mitigating contention, TRIO delivered a 34% bandwidth improvement over NOTRIO on average.

We also observed similar trends for both TRIO and NOTRIO in N-N case: The bandwidth of TRIO ascended from 1.5GB/s with 4 processes to 2.1GB/s with 8 processes, then stabilized from this point on. The bandwidth of NOTRIO kept dropping as the number of processes increased. These performance trend resulted from the same reasons as discussed for N-1 case. Another observation from this experiment was that, although same number of OSTs were used in both N-1 and N-N cases, TRIO_N_N performed better than TRIO_N_1. After analyzing single OST bandwidth under both cases, and discussing with the administrator, we found the lower bandwidth of TRIO_N_1 resulted from a bug in existing Lustre version (v2.5.3) as reported by [5].

D. Evaluating CAS's Bandwidth with Increasing Provisioned Storage Resources

Sometimes applications tend to stripe their files over a large number of OSTs to utilize more resources. Though utilizing more OSTs can deliver higher bandwidth, writing in a conventional manner that issues write requests to servers

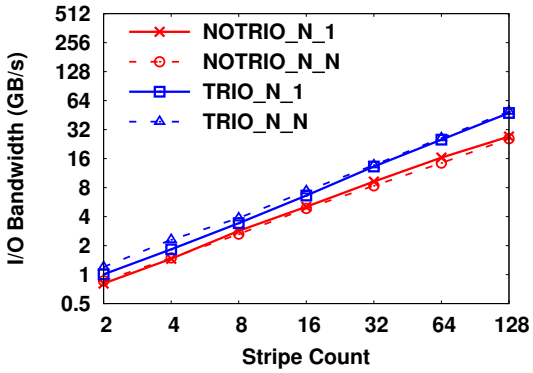

Fig. 9: Bandwidth of TRIO with Increasing Number of Provisioned OSTs.

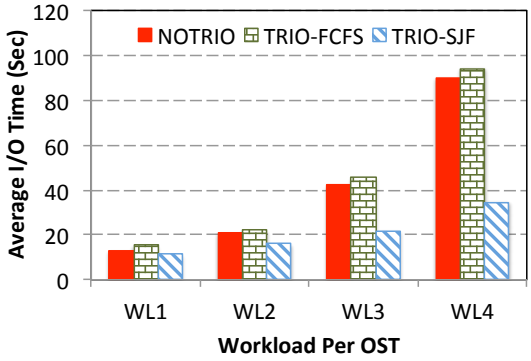

Fig. 10: Comparison of Average I/O Time.

in a round-robin manner can distribute each write request to more OSTs, incurring greater contention and preventing I/O bandwidth from further scaling. We emulated this scenario by striping each file over an increasing number of OSTs and using double the number of IOR processes to write on these OSTs. Contention for both N-1 and N-N patterns was assured by striping each file over the same set of OSTs.

Fig. 9 compares the bandwidth of TRIO and NOTRIO under this scenario. It can be observed that the bandwidth of NOTRIO-N-1 increased sublinearly from 0.81GB/s with a stripe count of 2 to 27GB/s with a stripe count of 128. In contrast, the bandwidth of TRIO increased with a much faster speed, resulting in on average a 38.6% improvement over NOTRIO. A similar trend was observed with the N-N checkpointing pattern. By localizing the writes of each BB on one OST each time and assigning the same number of BBs to each OST, CAS minimized the interference between different processes, thereby better utilizing the bandwidth. In some scenarios, localization may not help utilize more bandwidth. For instance, when the number of available OSTs is greater than the number of flushing BBs, localizing on one OST may underutilize the supplied bandwidth. We believe a similar approach can also work for these scenarios. For instance, we can assign a few OSTs to each BB, with each BB only distributing its writes among the assigned OSTs to mitigate interference.

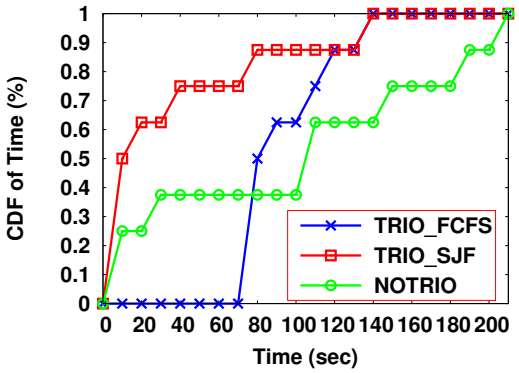

Fig. 11: The CDF of Job Response Time

E. Minimizing Average Job I/O Time using CAS

As mentioned in Section III-C, TRIO reduces average job I/O time by prioritizing small jobs. To evaluate this feature, we grouped 128 processes into 8 IOR jobs, each with 16 processes. We had each IOR process dump its dataset to its local BB and coordinated the data flush using TRIO. When multiple BBs request the same OST, TRIO selects a BB via the Shortest Job First (SJF) algorithm, which first serves a BB belonging to the smallest job. This configuration is shown in Fig. 10 as TRIO_SJF. For comparison, we applied FCFS in TRIO to select a BB. This configuration serves the first BB requesting this OST, and we refer to it as TRIO_FCFS. We also included the result of having each IOR process directly write its dataset to PFS, which we refer to as NOTRIO. We varied the data size such that each process in the next job wrote a separate file whose size was twice that of the prior job. Following this approach, each process in the smallest job wrote a 128MB file, and each process in the largest job wrote a 16GB file. To enable resource sharing, we striped the file so that each OST was shared by all 8 jobs. We increased the ratio of the number of processes over the number of OSTs to observe scheduling efficiency under different workloads.

Fig. 10 reveals average job I/O time for all the three cases. Workload 1 (WL1), WL2, WL3, and WL4 refer to scenarios when the number of processes was 2, 4, 8, and 16 times the number of OSTs, respectively. The average I/O time of TRIO_SJF was the shortest for all workloads, accounting for on average 57% and 63% of TRIO_FCFS and NOTRIO, respectively. We also observed that the I/O time of TRIO_SJF increased with growing workloads at a much slower rate than the other two. This was because, with the heavier workload, each OST absorbed more data from each job. This gave SJF more room for optimization. Another interesting phenomenon was that TRIO_FCFS demonstrated no benefit over NOTRIO in terms of the average I/O time. This was because, using TRIO_FCFS, once each BB acquired an available OST from the arbitrator, it drained all of its data striped over this OST. Since FCFS is unaware of large and small jobs, it is likely that the requests from the large job were scheduled first on a given OST. The small job requesting the same OST could only start draining its data after the large job finished. This

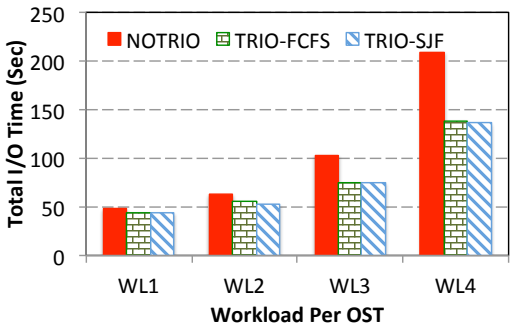

Fig. 12: Comparison of Total I/O Time

monopolizing behavior significantly delayed small jobs' I/O time.

For a further analysis, we also plotted the cumulative distribution functions (CDF) of job response time with WL4 as shown in Fig. 11, it is defined as the interval between the arrival time of first request of the job at the arbitrator and the time when the job complete its I/O task. By scheduling small jobs first, 87.5% of jobs in TRIO-SJF were able to complete their work within 80 seconds. By contrast, jobs in TRIO-FCFS and NOTRIO completed at much slower rates.

Fig. 12 shows the total I/O time of draining all the jobs' datasets. There was no significant distinction between TRIO_FCFS and TRIO_SJF because, from OST's perspective, each OST was handling the same amount of data for the two cases. By contrast, I/O time of NOTRIO was longer than the other two due to contention. The impact of contention became more significant under larger workloads.

V. Related Work

I/O Contention: In general, research around I/O contention falls into two categories: client-side and server-side optimization. In client-side optimization, processes involved in the same job collaboratively coordinate their access to the PFS to mitigate contention. Abbasia *et al.* [10] and Nisar *et al* [21] address contention by delegating the I/O of all processes involved in the same application to a small number of compute nodes. Chen [11] and Liao *et al*[22] mitigate I/O contention in MPI-IO by having processes shuffle data in a layout-aware manner. Server-side optimization embeds some I/O control mechanisms on the server side. Dai *et al* [12] have designed an I/O scheduler that dynamically places write operations among servers to avoid congested servers. Zhang *et al* [12] propose a server-side I/O orchestration mechanism to mitigate interference between multiple processes. Different from these works, we address I/O contention issues using BB as an intermediate layer. Compared with client-side optimization, an orchestration framework on BB is able to coordinate I/O traffic between different jobs, mitigating I/O contention at a larger scope. Compared with the server-side optimization, an orchestration framework on BB can free storage servers from the extra responsibility of handling I/O contention, making it portable to other PFSs.

Burst Buffer: The idea of BB was proposed recently to cope with data handling challenges presented by the upcoming exascale computing era. Two of the largest next-generation HPC systems, Coral [2] and Trinity [9], are designed with BB support. The SCR group is currently trying to strengthen the support for SCR by developing a multi-level checkpointing scheme on top of BB [6]. DataDirect Networks is developing the Infinite Memory Engine (IME) [3] as a BB layer to provide real-time I/O service for the scientific applications. Most of these works use BB as an intermediate layer to avoid application's direct interaction with PFS. The focal point of our work is the interaction between BB and PFS. Namely, how to efficiently flush data to PFS.

Inter-Job I/O Coordination: Compared with the numerous research works on intra-job I/O coordination, inter-job coordination has received very limited attention. Liu *et al.* [18] design a tool to extract the I/O signatures of various jobs to assist the scheduler in making optimal scheduling decisions. Dorier *et al.* [13] propose a reactive approach to mitigate I/O interference from multiple applications by dynamically interrupting and serializing application's execution upon performance decrease. Our work differs in that it coordinates inter-job I/O traffic in a layout-aware manner to both avoid bandwidth degradation and minimize average job I/O time under contention.

VI. Conclusion

In this paper, we have analyzed the major performance issues of checkpointing operations on HPC systems: prolonged average job I/O time and degraded storage server bandwidth utilization. Accordingly, we have designed a burst buffer based orchestration framework, named TRIO, to efficiently intercept and reshape I/O traffic from burst buffer to PFS. By increasing intra-BB write sequentiality and coordinating inter-BB flushing order, TRIO is able to deliver on average 30.5% higher bandwidth and reduce average job I/O time by 37% in various checkpointing patterns.

Our future work is two-fold. First, we will investigate an aggressive mechanism that can automatically flush the data based on their utilizationt. Second, we will explore a distributed arbitration mechanism to determine the inter-BB flushing order among a distributed set of BB nodes and PFS storage servers.

Acknowledgments

This research is sponsored in part by the Office of Advanced Scientific Computing Research; U.S. Department of Energy and performed at the Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 and resources of the Oak Ridge Leadership Computing Facility, located in the National Center for Computational Sciences at Oak Ridge National Laboratory. This work is also funded in part by an Alabama Innovation Award and National Science Foundation awards 1059376, 1320016, 1340947, and 1432892.

REFERENCES

[1] blktrace. http://linux.die.net/man/8/blktrace.

[2] CORAL. https://www.olcf.ornl.gov/summit.

[3] IME. http://www.ddn.com/products/infinite-memory-engine-ime.

[4] Introducing Titan. http://www.olcf.ornl.gov/titan/.

[5] LU-4388. https://jira.hpdd.intel.com/browse/LU-4388.

[6] SCR. https://computation.llnl.gov/project/scr.

[7] The ASC Sequoia Draft Statement of Work. https://asc.llnl.gov/sequoia/rfp/02_SequoiaSOW_V06.doc.

[8] Tianhe-2. http://www.top500.org/system/177999.

[9] TRINITY. https://www.nersc.gov/assets/Trinity–NERSC-8-RFP/Documents/trinity-NERSC8-use-case-v1.2a.pdf.

[10] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng. Datastager: Scalable data staging services for petascale applications. *Cluster Computing*, 13(3):277–290, 2010.

[11] Y. Chen, X.-H. Sun, R. Thakur, P. C. Roth, and W. D. Gropp. Lacio: A new collective i/o strategy for parallel i/o systems. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 794–804. IEEE, 2011.

[12] D. Dai, Y. Chen, D. Kimpe, and R. Ross. Two-choice randomized dynamic i/o scheduler for object storage systems. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*, pages 635–646. IEEE, 2014.

[13] M. Dorier, G. Antoniu, R. Ross, D. Kimpe, S. Ibrahim, et al. Calciom: Mitigating i/o interference in hpc systems through cross-application coordination. In *IPDPS-International Parallel and Distributed Processing Symposium*, 2014.

[14] K. Ferreira, R. Riesen, R. Oldfield, J. Stearley, J. Laros, K. Pedretti, T. Kordenbrock, and R. Brightwell. Increasing fault resiliency in a message-passing environment. *Sandia National Laboratories, Tech. Rep. SAND2009-6753*, 2009.

[15] J. N. Glosli, D. F. Richards, K. Caspersen, R. Rudd, J. A. Gunnels, and F. H. Streitz. Extending stability beyond cpu millennium: a micron-scale atomistic simulation of kelvin-helmholtz instability. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, page 58. ACM, 2007.

[16] Y. Kim, S. Atchley, G. R. Vallée, and G. M. Shipman. Lads: Optimizing data transfers using layout-aware data scheduling. 2015.

[17] 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*, pages 1–11. IEEE, 2012.

[18] Y. Liu, R. Gunasekaran, X. Ma, and S. S. Vazhkudai. Automatic identification of application i/o signatures from noisy server-side traces. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, pages 213–228, Berkeley, CA, USA, 2014. USENIX Association.

[19] LLNL. IOR Benchmark. http://www.llnl.gov/asci/purple/benchmarks/limited/ior.

[20] A. Moody, G. Bronevetsky, K. Mohror, and B. R. De Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11. IEEE, 2010.

[21] A. Nisar, W.-k. Liao, and A. Choudhary. Scaling parallel i/o performance through i/o delegate and caching system. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–12. IEEE, 2008.

[22] A. Nisar, W.-k. Liao, and A. Choudhary. Delegation-based i/o mechanism for high performance computing systems. *Parallel and Distributed Systems, IEEE Transactions on*, 23(2):271–279, 2012.

[23] R. A. Oldfield, L. Ward, R. Riesen, A. B. Maccabe, P. Widener, and T. Kordenbrock. Lightweight i/o for scientific applications. In *Cluster Computing, 2006 IEEE International Conference on*, pages 1–11. IEEE, 2006.

[24] S. Oral, D. A. Dillow, D. Fuller, J. Hill, D. Leverman, S. S. Vazhkudai, F. Wang, Y. Kim, J. Rogers, J. Simmons, et al. Olcfs 1 tb/s, next-generation lustre file system. In *Proceedings of Cray User Group Conference (CUG 2013)*, 2013.

[25] F. Petrini. Scaling to thousands of processors with buffered coscheduling. In *Scaling to New Heights Workshop*, 2002.

[26] K. Ren and G. A. Gibson. Tablefs: Enhancing metadata efficiency in the local file system. In *USENIX Annual Technical Conference*, pages 145–156, 2013.

[27] O. Rodeh, J. Bacik, and C. Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):9, 2013.

[28] H. Shan and J. Shalf. Using ior to analyze the i/o performance for hpc platforms. *Lawrence Berkeley National Laboratory*, 2007.

[29] H. Song, Y. Yin, X.-H. Sun, R. Thakur, and S. Lang. Server-side i/o coordination for parallel file systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 17. ACM, 2011.

[30] T. Wang, S. Oral, Y. Wang, B. Settlemyer, S. Atchley, and W. Yu. Burstmem: A high-performance burst buffer system for scientific applications. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 71–79. IEEE, 2014.

[31] P. Wong and R. der Wijngaart. Nas parallel benchmarks i/o version 2.4. *NASA Ames Research Center, Moffet Field, CA, Tech. Rep. NAS-03-002*, 2003.

[32] X. Zhang, K. Davis, and S. Jiang. Iorchestrator: improving the performance of multi-node i/o systems via inter-server coordination. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.