TRIO: Burst Buffer Based I/O Orchestration

Teng Wang* Sarp Oral† Michael Pritchard‡ Bin Wang‡ Weikuan Yu*
Florida State University* Oak Ridge National Lab† Auburn University‡
{twang,yuw}@cs.fsu.edu {oralhs}@ornl.gov {mjp0009,bwang}@auburn.edu

***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 file systems can incur significant I/O contention on storage servers. Such contention in turn degrades bandwidth utilization of storage servers and prolongs the average job I/O time of concurrent applications. Recently burst buffers have 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 desirable to efficiently move checkpointing data from burst 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 servers. Meanwhile, TRIO coordinates the flushing orders among concurrent burst buffers to alleviate the contention on storage server. Our experimental results demonstrated that TRIO could efficiently utilize storage bandwidth and reduce the average job I/O time by 37% on average for data-intensive applications in typical 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 [18]. As a common defensive mechanism for failure recovery, checkpointing dominates 75%-80% of the I/O traffic on current High Performance Computing (HPC) systems [34, 7].

Though checkpointing is necessary for fault tolerance, it can introduce significant overhead. For instance, a study from Sandia National Laboratory 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 [17]. A major reason 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∼100x more than those on storage systems [32, 29], 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 [19]. Second, when multiple jobs compete to use the storage server, 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 [16].

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 [10, 30] or optimizing application's I/O pattern [13]. Server-side optimizations generally embed their solutions inside the storage server, overcoming issues of contention by dynamically coordinating data movement among servers [37, 14, 43].

Recently, the idea of Burst Buffers (BB) was proposed as an additional layer with fast memory devices such as DRAM and SSDs to absorb bursty I/O from scientific applications [22]. Many consider it as a promising solution of I/O contention for next-generation computing platforms. With the mediation of BBs, applications can directly dump their large checkpoint datasets to BBs and minimize their direct interactions with PFS. BBs can flush the data to PFS at a later point of time. However, as 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 BBs 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 BBs can free storage servers from the extra responsibility of handling I/O contention, making it 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 assists 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.
- 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.

978-1-4673-6598-7/15 $31.00 © 2015 IEEE
DOI 10.1109/CLUSTER.2015.38

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*).

- We have evaluated the performance of TRIO using representative checkpointing patterns. Our results demonstrated that TRIO efficiently utilized storage bandwidth and reduced average job I/O time by 37%.

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

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 contains 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. The Spider II filesystem [33] serves as the backend storage system for Titan. It is composed of two Lustre-based filesystems [12]: 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: We used multi-job workloads composed of IOR [26], MPI-Tile-IO [5], and BTIO [42]. 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 PFSs. 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 were configured to use N-1 checkpointing pattern in which each process writes to a non-overlapping, contiguous extent of a shared file, while BTIO was configured to follow the N-N checkpointing pattern that each process writes to a separate files. We dedicated 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. We ran each test 15 times and the median result was presented.

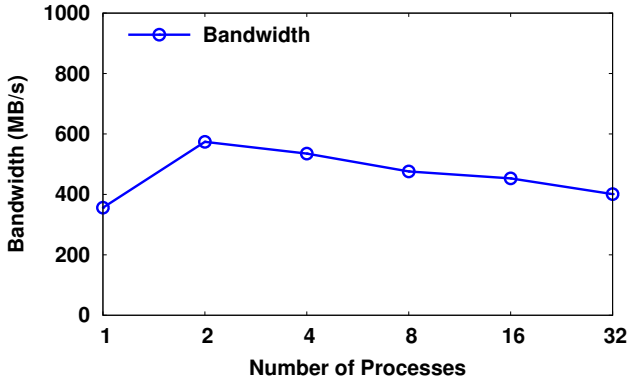


Fig. 1: The impact of increasing number of concurrent processes to the bandwidth of a single OST.

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 ran BTIO, MPI-Tile-IO and IOR concurrently but differentiated their output data sizes as 13.27GB, 64GB and 128GB respectively. This launch configuration is referred to as *MultiWL*. Competition for storage was assured by striping all benchmark output files across the same four OSTs. We compared their job I/O time with that when these benchmarks were 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 was 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 was BTIO, which generated the smallest workload, its I/O time in MultiWL was 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, each process is involved in the competition for multiple OSTs. We used IOR benchmark to analyze the impacts of both challenges. Similar to previous experiment, one IOR process ran on each compute node.

To investigate the first challenge, we used an increasing number of processes to concurrently write in total 32GB data to a single OST. Each process wrote a contiguous, nonoverlapping file segment. The result is exhibited in Fig. 1.

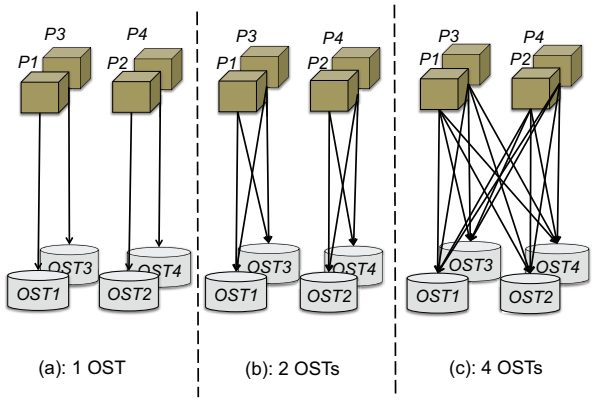


Fig. 2: Scenarios when individual writes are distributed to different number of OSTs. "N OST" means that each process's writes are distributed to N OSTs.

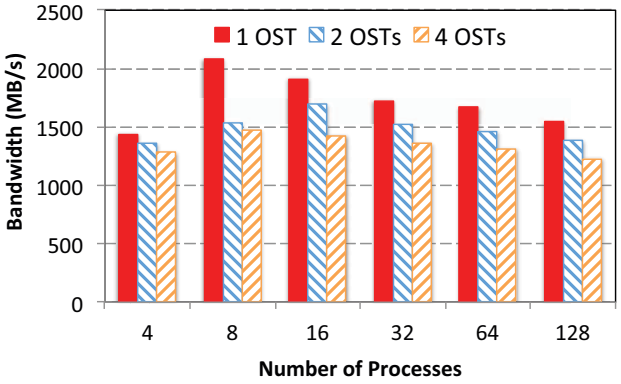


Fig. 3: Bandwidth when individual writes are distributed to different number of OSTs.

The bandwidth first increased from 356MB/s with 1 process to 574MB/s with 2 processes, then decreased 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 was because the single-process I/O stream was not able to saturate OST bandwidth. Specifically, each OST was organized as RAID-6 arrays, yielding a raw bandwidth of 750MB/s [33]. When an I/O request was issued, it was relayed multiple hops from peer compute nodes to I/O router, then went through SION network, Object Storage Server (OSS) and eventually arrived at OST. Despite of the high network bandwidth along the critical path, the extra data copy and data processing overhead at each hop caused additional delays [15]. Overall, the bandwidth utilization of one OST was 75.6% when there were only two concurrent processes, but dropped to 53.5% when there were 32 processes.

Our intuition suggested 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 prevented us from directly proving our intuition. We repeated our experiments on our in-house Lustre filesystem (running the same version as that on Spider) and observed that up to 32% bandwidth degradation were caused by I/O contention. By analyzing I/O traces using blktrace [1], we found that disk access time accounted for 97.1% of the total I/O time on average, indicating that excessive concurrent accesses to OSTs can degrade the bandwidth utilization.

To emulate the second challenge, we distributed each IOR process'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 fixed the total data size as 128GB, the number of utilized OSTs as 4, and measured 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 delivered the highest bandwidth with the same number of processes, outperforming 2 OSTs and 4 OSTs by 16% and 26% on average, respectively. This was because, by localizing each process's writes on 1 OST, each OST was competed by fewer processes. Another interesting observation is that the bandwidth under the same scenario (e.g. 1 OST) degrades with more processes involved. This trend can be explained by the impact of first issue as we measured in Fig. 1.

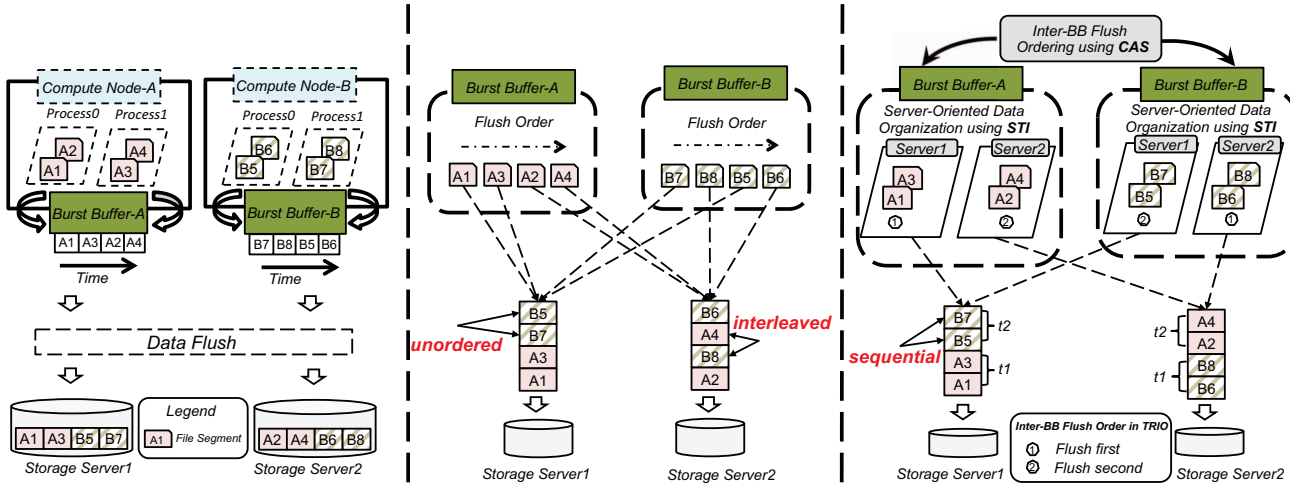
Based on our characterization, under a contentious environment where numerous processes concurrently access a smaller number of OSTs, bandwidth can be more efficiently utilized by localizing each process's access on one OST (e.g. Fig. 2(a)), and scheduling a proper number of processes to access each OST (e.g. 2 in Fig. 1).

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 [22], few works [40] have discussed the interaction between BB and PFS, i.e. how to orchestrate I/O so that large 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 Section III-B and Section III-C.

A. Main Idea of TRIO

Fig. 4(a) illustrates a general framework of how BBs interact 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



(a) Burst buffer framework with data flush.

(b) Reactive data flush.

(c) Proactive data flush with TRIO.

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.

located on each CN during checkpointing, then flushed from BB to two storage servers on PFS.

An intuitive strategy 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. This 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 Fig. 4(a), segments B5 and B7 are contiguously laid out on Storage Server 1, but they arrive at BB-B out of order in Fig. 4(b). 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 in Section II-C. As indicated by this figure, BB-A and BB-B concurrently flush A4 and B8 to Server 2, so the two segments are *interleaved*, their arrival order is against their physical layout on Server 2 (see Fig.4 (a)). 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 from different applications to the shared storage servers can prolong the average job I/O time and delay the timely service for mission-critical and small jobs as described in Section II-B.

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 1. Meanwhile, localizing BB-B's writes on Server 1 minimizes its interference on Server 2 during this interval, which mitigates the impact of N-N mapping as discussed in Section II-C. Similarly, BB-A organizes A2, A4 together and flush them sequentially to Server 2, minimizing its interference on Server 1. However, contention can arise if both BB-A and BB-B flush to the same servers. TRIO addresses this problem using a second technique, Contention-Aware Scheduling (CAS), as 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 Server 1 and Server 2 in sequence, while BB-B flushes to Server 2 and 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 [35, 36] 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, as mentioned in Section III-A, sequentially flushing all the file segments under a given storage server together is beneficial. To accomplish this I/O pattern, BB needs to group all segments

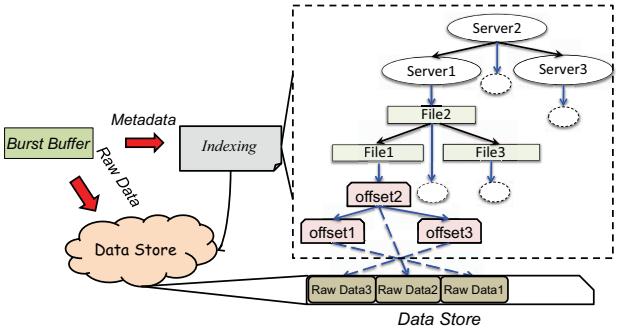


Fig. 5: Server-Oriented Data Organization with Stacked AVL Tree. Segments of each server can be sequentially flushed following in-order traversal of the tree nodes under this server.

on the same storage server together. Meanwhile, since these segments can come from different files (e.g. File1, File2, File3 on Server 1 in Fig. 5), sequential flush requires BB to group together segments of the same file and then order these segments based on the offset. Accomplishing the aforementioned purpose using 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 Adelson-Velskii and Landis (**A**VL) Tree based Indexing (STI) [40] to address these requirements. Like many other conventional tree structures, AVL tree [20] 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 tree nodes. STI differs in that all the tree nodes are organized in a stacked manner. 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 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 (see Fig. 5) 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 Server 1 in Fig. 5 can be accomplished by traversing the subtree of the node "Server 1", 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.

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 Server 1 and Server 2 in sequence, while BB-B flushes to Server 2 and Server 1 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-Oriented 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(c), BB-A buffers segments of Server 1 and Server 2. Assuming $\alpha = 1$, it needs to select a server that has not been assigned to any BB. Since BB-B is flushing to Server 2 at time t1, BB-A picks up Server 1 and flushes the corresponding segments (A1, A3) to this server. By doing so, the contention on Server 1 and Server

2 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, and 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 being queued for flushing to the same storage server. 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, an issue analyzed in Section II-B. 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.

IV. Implementation and Complexity Analysis

We have built a proof-of-concept prototype of TRIO using C with Message Passing Interface (MPI). To emulate I/O in the multi-job environment, more than one communicators can be created among all BBs, each corresponding to a disjoint set of BBs involved in their mutual I/O tasks. The bursty writes from all these sets are orchestrated by the arbitrator. We also leverage STI (See Section III-B) to organize all the data structures inside arbitrator for efficient and semantic-based lookup and update. For instance, when a storage server becomes available, the arbitrator needs to assign it to a waiting BB that has data on it. This BB should belong to the job with higher priority than other waiting BBs' jobs. Under this semantics, the profile of job, storage server, BB are stacked as three layers on STI. Assuming a system with m BB-augmented compute nodes and n storage servers, and each job uses k compute nodes. At most this STI contains $m/k+mn/k+mn$ nodes, where m/k, mn/k, mn are respectively the number of tree nodes in each layer. This means, for a system with 10000 compute nodes and 1000 storage servers, the number of tree nodes is at most 20M, incurring less than 1GB storage overhead. On the other hand, the time spent on arbitrator is dominated by its communication with each BB. Existing high-speed interconnects (e.g. QDR Infiniband) generally yield a roundtrip latency lower than 10 μs for small messages (smaller than 1KB) [28], this means a single arbitrator is able to handle the requests from 10^5 BBs within 1 second. A a scalable arbitration mechanism is presented in Section VII as one of the future works.

V. Experiment Evaluation

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 reserved 16GB as the burst buffer space. We evaluated TRIO against the workload from IOR benchmark, which was presented in Section II. Each test case was run 15 times for every data point, the median result was presented.

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 tests, 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 benefit of TRIO under competing workloads with an increasing number of IOR processes. We compared the aggregated OST bandwidth of two configurations. In the first configuration (shown in Fig. 6 as NOTRIO), all the processes directly issued their write requests to PFS. In the second configuration (shown in Fig. 6 as TRIO), all processes' write requests were buffered on the burst buffer space and flushed to PFS using TRIO. N-1 and N-N patterns (see Section II-A) were employed in both configurations. We ran 16 processes on each compute node and stressed the system by increasing the number of compute nodes involved from 4 to 256. Each process wrote in total 1GB data. Both request size and stripe size were configured as 1MB. I/O competition of all processes was assured by striping each file over the same four OSTs (default stripe count).

As we can see from Fig. 6, in both N-1 and N-N patterns, bandwidth of 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, 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 4 nodes than other node numbers in both TRIO-N-1 and TRIO-N-N cases was due to OST bandwidth not being fully utilized (4 BBs were used to flush to 4 OSTs in these cases). Overall, on average TRIO improved I/O bandwidth by 77% for N-1 pattern and 139% for N-N pattern.

B. Dissecting Performance of TRIO

To gain more insight into the contributions of each technique of TRIO, we incrementally analyzed its performance based on the five configurations shown in Fig. 7. In NAIVE, each

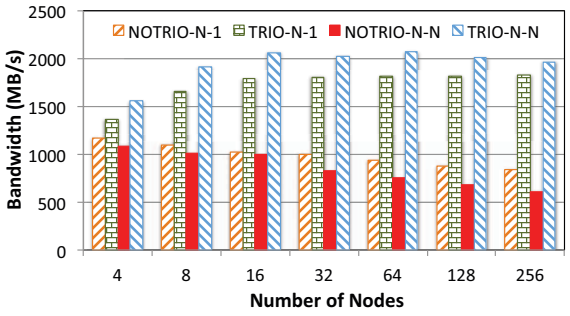


Fig. 6: The overall performance of TRIO under both inter-node and intra-node I/O traffic.

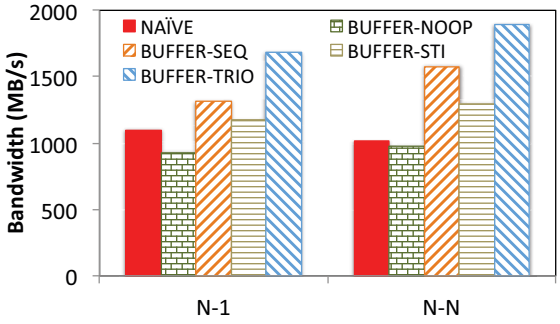


Fig. 7: Performance dissection of TRIO.

process directly wrote its data to PFS. In BUFFER-NOOP, all processes' write requests were buffered in BB and flushed without any optimization, this configuration corresponds to the reactive approach discussed in Section III-A. In BUFFER-SEQ, all the buffered write requests were reordered according to their offsets and flushed sequentially. In BUFFER-STI, all the write requests were organized using STI, each time a *random* OST was selected under the AVL tree and all write requests that belong to this OST were sequentially flushed. In BUFFER-TRIO, CAS was enabled on top of STI, which restricted the number of concurrent flushing BBs on each OST to 2, this configuration corresponds to the proactive approach discussed in Section III-A. We evaluated the five configurations using the workload of 8-node case (128 processes write to 4 OSTs) in Fig. 6. In this case, TRIO reaped its most benefits since the number of flushing BBs was twice the number of OSTs.

As we can see from Fig. 7, simply buffering the dataset in BUFFER-NOOP was not able to improve the performance over NAIVE due to issues discussed in Section III-A. In contrast, the sequential flush order in BUFFER-SEQ significantly outperformed NAIVE for both N-1 and N-N patterns. Interestingly, although STI sequentially flushed the write requests to each OST, it demonstrated no benefit over BUFFER-SEQ. This is because, without controlling the number of flushing BBs on each OST, each OST was flushed by an unbalanced number of BBs, the benefits of localization and sequential flush using STI were offset by prolonged contention on the

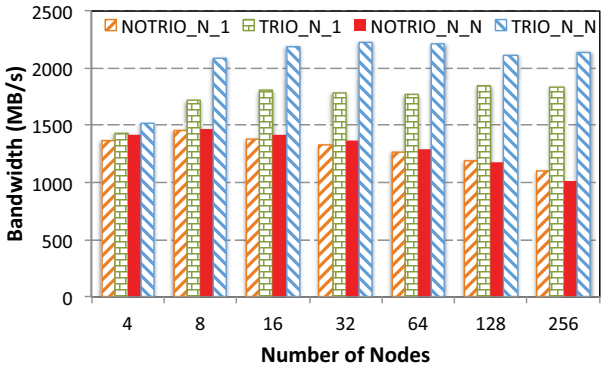


Fig. 8: The bandwidth of TRIO under I/O contention with increasing number of processes.

overloaded OSTs. This issue was alleviated when CAS was enabled in BUFFER-TRIO: by placing 2 BBs on each OST and localizing their flush, the bandwidth of each OST was more efficiently utilized than BUFFER-SEQ.

C. Alleviating Inter-Node Contention using CAS

To evaluate TRIO's ability to sustain inter-node I/O contention using CAS, we placed 1 IOR process on each node and had each process dump a 16GB dataset to its local BB, these datasets were flushed to the PFS using TRIO. 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. Contention for both patterns was assured by striping all 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 1.4GB/s with 4 processes to 1.7GB/s with 8 processes, then stabilized around 1.8GB/s. 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 1.4GB/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 inter-node contention generated by larger numbers of processes. Overall, by mitigating contention, TRIO delivered a 35% bandwidth improvement over NOTRIO on average.

We also observed similar trends for 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 more processes were involved. These performance trend resulted from the same reasons as discussed for N-1 case.

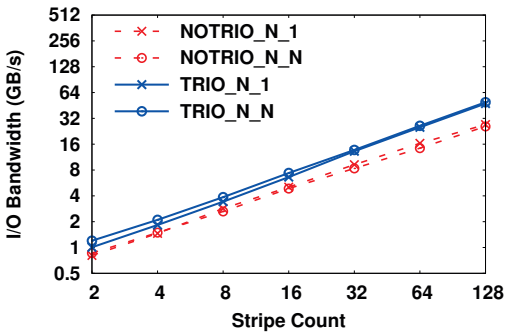


Fig. 9: Bandwidth of TRIO with increasing stripe count.

D. Performance of CAS with More OSTs

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 in a round-robin manner can distribute each write request to more OSTs, incurring wider 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. Sometimes 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 multiple OSTs to each BB, with each BB only distributing its writes among the assigned OSTs to mitigate interference.

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 jobs, each with 16 processes, and place 1 process on each node. We had each process dump its dataset to its local BB and coordinated the data flush using TRIO. When multiple BBs requested the same OST, TRIO selected a BB via the Shortest Job First (SJF) algorithm, which first served 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 served the first BB requesting this OST first, and

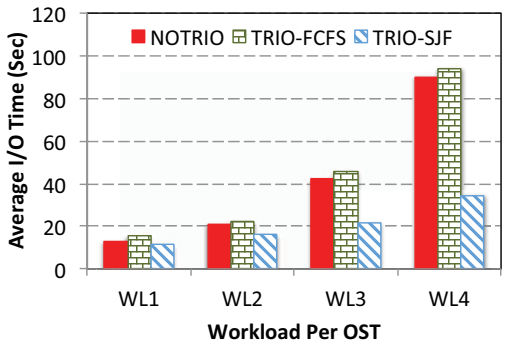


Fig. 10: Comparison of Average I/O Time.

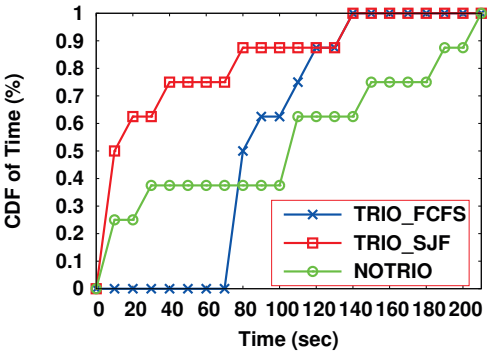


Fig. 11: The CDF of Job Response Time

we refer to it as TRIO_FCFS. We also included the result of having each 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 on 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

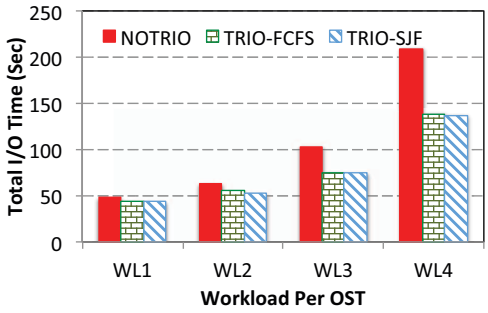


Fig. 12: Comparison of Total I/O Time

data after the large job finished. This 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 completes its I/O task. By scheduling small jobs first, 7 out of 8 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.

VI. 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.* [30] addressed contention by delegating the I/O of all processes involved in the same application to a small number of compute nodes. Chen *et al.* [13] and Liao *et al.* [31] mitigated I/O contention in by having processes shuffle data in a layout-aware manner. These mechanisms have been widely adopted in existing I/O middlewares, such as ADIOS [27, 25] and MPI-IO [38]. Server-side optimization embeds some I/O control mechanisms on the server side. Dai *et al.* [14] designed an I/O scheduler that dynamically places write operations among servers to avoid congested servers. Zhang *et al.* [14] proposed a server-side I/O orchestration mechanism to mitigate interference between multiple processes. Liu *et al.* [24] researched a low level caching mechanism that optimizes the I/O pattern on hard disk drives. 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 the exploding data pressure in 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.* [23] designed a tool to extract the I/O signatures of various jobs to assist the scheduler in making optimal scheduling decisions. Dorier *et al.* [16] proposed 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.

VII. Conclusions and Future Work

In this paper, we have analyzed 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 reshape I/O traffic from burst buffer to PFS. By increasing intra-BB write sequentiality and coordinating inter-BB flushing order, TRIO efficiently utilized storage bandwidth and reduced average job I/O time by 37% in the typical checkpointing patterns.

We plan our future work to strengthen the current design in three aspects. First, existing framework uses one arbitrator for orchestration, which limits its scalability. So one focus is to distribute the responsibility of the arbitrator to a number of BBs. This can be accomplished by partitioning storage servers into disjoint sets and assigning one arbitrator to orchestrate the I/O requests to each set. Second, existing framework is designed to handle large, sequential checkpointing workload, which is not favored by small or noncontiguous checkpointing workload. We believe the latter can be resolved by introducing additional support on top of existing framework [11, 41, 21, 39]. Third, we will investigate an aggressive mechanism that can automatically flush the data based on their utilization.

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.
[4] Introducing Titan. http://www.olcf.ornl.gov/titan/.
[5] MPI-Tile-IO. http://www.mcs.anl.gov/research/projects.
[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] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. PLFS: A checkpoint filesystem for parallel applications. In *SC*, 2009.
[12] P. J. Braam and R. Zahir. Lustre: A scalable, high performance file system. *Cluster File Systems, Inc*, 2002.
[13] 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 *IPDPS*.
[14] D. Dai, Y. Chen, D. Kimpe, and R. Ross. Two-choice randomized dynamic I/O scheduler for object storage systems. In *SC*, 2014.
[15] D. A. Dillow, G. M. Shipman, S. Oral, Z. Zhang, and Y. Kim. Enhancing I/O throughput via efficient routing and placement for large-scale parallel file systems. In *IPCCC*, 2011.
[16] 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*, 2014.
[17] 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.
[18] 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 *SC*, 2007.
[19] Y. Kim, S. Atchley, G. R. Vallée, and G. M. Shipman. LADS: optimizing data transfers using layout-aware data scheduling. In *FAST*, 2015.
[20] D. E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.
[21] J. Liu, B. Crysler, Y. Lu, and Y. Chen. Locality-driven high-level I/O aggregation for processing scientific datasets. In *IEEE BigData*, 2013.
[22] 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 *MSST*, 2012.
[23] Y. Liu, R. Gunasekaran, X. Ma, and S. S. Vazhkudai. Automatic identification of application I/O signatures from noisy server-side traces. In *FAST*, 2014.
[24] Z. Liu, B. Wang, P. Carpenter, D. Li, J. S. Vetter, and W. Yu. PCM-based durable write cache for fast disk I/O. In *MASCOTS*, 2012.
[25] Z. Liu, B. Wang, T. Wang, Y. Tian, C. Xu, Y. Wang, W. Yu, C. A. Cruz, S. Zhou, T. Clune, et al. Profiling and improving i/o performance of a large-scale climate scientific application. In *ICCCN*, pages 1–7. IEEE, 2013.
[26] LLNL. IOR Benchmark. http://www.llnl.gov/asci/purple/benchmarks/limited/ior.
[27] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan. Adaptable, metadata rich IO methods for portable high performance IO. In *IPDPS*, 2009.
[28] C. Mitchell, Y. Geng, and J. Li. Using one-sided RDMA reads to build a fast, cpu-efficient key-value store. In *USENIX ATC*, 2013.
[29] A. Moody, G. Bronevetsky, K. Mohror, and B. R. De Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *SC*, 2010.
[30] A. Nisar, W.-k. Liao, and A. Choudhary. Scaling parallel I/O performance through I/O delegate and caching system. In *SC*, 2008.
[31] A. Nisar, W.-k. Liao, and A. Choudhary. Delegation-based I/O mechanism for high performance computing systems. *TPDS*, 23(2):271–279, 2012.
[32] R. A. Oldfield, L. Ward, R. Riesen, A. B. Maccabe, P. Widener, and T. Kordenbrock. Lightweight I/O for scientific applications. In *CLUSTER*, 2006.
[33] 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 *CUG*, 2013.
[34] F. Petrini. Scaling to thousands of processors with buffered coscheduling. In *Scaling to New Heights Workshop*, 2002.
[35] K. Ren and G. A. Gibson. TABLEFS: Enhancing metadata efficiency in the local file system. In *USENIX ATC*, 2013.
[36] O. Rodeh, J. Bacik, and C. Mason. BTRFS: The linux B-tree filesystem. *TOS*, 9(3):9, 2013.
[37] H. Song, Y. Yin, X.-H. Sun, R. Thakur, and S. Lang. Server-side I/O coordination for parallel file systems. In *SC*, 2011.
[38] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective I/O in ROMIO. In *Frontiers of Massively Parallel Computation, 1999. Frontiers' 99. The Seventh Symposium on the*, pages 182–189. IEEE, 1999.
[39] Y. Tian, Z. Liu, S. Klasky, B. Wang, H. Abbasi, S. Zhou, N. Podhorszki, T. Clune, J. Logan, and W. Yu. A lightweight I/O scheme to facilitate spatial and temporal queries of scientific data analytics. In *MSST*, 2013.
[40] T. Wang, S. Oral, Y. Wang, B. Settlemyer, S. Atchley, and W. Yu. Burstmem: A high-performance burst buffer system for scientific applications. In *IEEE BigData*, 2014.
[41] T. Wang, K. Vasko, Z. Liu, H. Chen, and W. Yu. Bpar: A bundle-based parallel aggregation framework for decoupled I/O execution. In *DISCS*, 2014.
[42] 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.
[43] X. Zhang, K. Davis, and S. Jiang. IOrchestrator: improving the performance of multi-node I/O systems via inter-server coordination. In *SC*, 2010.