BurstMem: A High-Performance Burst Buffer System for Scientific Applications

Teng Wang[†] Sarp Oral[‡] Yandong Wang[†] Brad Settlemyer[‡] Scott Atchley[‡] Weikuan Yu[†]
Auburn University[†] Oak Ridge National Laboratory[‡]
Auburn University, AL 36849 Oak Ridge, TN 37831
{tzw0019,wangyd,wkyu}@auburn.edu {oralhs,settlemyerbw,atchleyes}@ornl.gov

***Abstract*—The growth of computing power on large-scale systems requires commensurate high-bandwidth I/O systems. Many parallel file systems are designed to provide fast sustainable I/O in response to applications' soaring requirements. To meet this need, a novel system is imperative to temporarily buffer the bursty I/O and gradually flush datasets to long-term parallel file systems. In this paper, we introduce the design of BurstMem, a high-performance burst buffer system. BurstMem provides a storage framework with efficient storage and communication management strategies. Our experiments demonstrate that BurstMem is able to speed up the I/O performance of scientific applications by up to 8.5$\times$ on leadership computer systems.**

I. INTRODUCTION

The astonishing growth of top 500 supercomputers suggests that, around the time frame of 2018-2020, the computation power of "leadership-class" systems is likely to surpass 1 ExaFlop/sec (10^{18} flops/sec). This unprecedented speed requires a commensurate I/O bandwidth of nearly 60 TB/s to enable the timely storage of application checkpoint data [13]. Unfortunately, the I/O systems have not been able to keep up with such computation power. For I/O intensive scientific applications, the I/O time still accounts for a high percentage of their life cycle. For example, a study on a massively parallel electromagnetic solver system (NekCEM) shows 60% of the overall application execution time is spent on checkpointing on Blue Gene/P systems [14].

For a long time, research has been concentrated on designing Parallel File Systems (PFS) for aggregated I/O bandwidth. Several PFSs are deployed on current leadership-class computers, such as PVFS [28], GPFS [29] and Lustre [9]. More recently, research has been centered on the development of an I/O forwarding layer [5] and data staging [4]. These strategies are effective in bridging the gaps between computation and I/O performance. However, none offers sufficient support to application checkpointing, a bursty I/O behavior dominating 75%-80% of current HPC I/O workloads [26, 1].

Recently, the idea of *burst buffer* has been proposed to cope with the exploding data pressure from scientific applications. Many consider this as a promising solution to the I/O crisis on HPC systems. The main strategy is to leverage an external storage system with tiers of high-speed storage devices between compute nodes and PFS [21]. Despite the attractive benefits of burst buffer, most existing studies focus on modeling and simulation of burst buffer [21, 33]. Design and implementation are rarely documented. In such simulation studies, burst buffers are generally simulated as a write-back cache without providing any details on buffer management.

In this paper, we systematically design a burst buffer system named BurstMem. It provides a simple interface that allows applications to quickly dump checkpoint data, and asynchronously flush the data to PFS without interfering with application computation. With a set of storage management strategies, it efficiently leverages the capacity and bandwidth of storage devices and reduces the overall I/O time. In addition, BurstMem is designed with a novel tree-based indexing technique that can support fast data flush in two phases. Finally, BurstMem is implemented with a portable and fast communication layer that enables its portability to systems with diverse network configurations [6].

We implement BurstMem by customizing and extending the functionality of a cutting-edge caching system named Memcached. It is a lightweight, distributed DRAM-based caching system. Though not designed for scientific applications, it includes all the features for distributed buffer management. Its fast storage solution and great extensibility for complex applications distinguish it from many other distributed storage systems (e.g. MongoDB [11], HBase [15]), as a decent candidate for burst buffer services. We customize Memcached by modifying its data placement strategy, communication layer and memory management module. Furthermore, we design BurstMem with a mechanism of coordinated data shuffling and flushing to PFS. In summary, we make the following contributions in this paper:

- We have examined the storage management issues in Memcached. Based on our analysis we introduce a log-structured storage management scheme with a novel tree-based indexing technique. This allows us to efficiently utilize storage resources.
- We have applied a coordinated shuffling scheme for efficient data flushing, and designed a portable communication layer that supports high-speed data transfer.
- A systematic evaluation of BurstMem is conducted using both synthetic benchmarks and a real-world application. Our results demonstrate that on average BurstMem can improve the I/O performance by as much as 8.5$\times$.

II. BACKGROUND AND MOTIVATION

In this section, we first provide an overview of Memcached software architecture and the I/O characteristics of scientific applications, and then motivate the design of burst buffer framework on top of Memcached for scientific applications.

A. Memcached

Memcached is an open-source, distributed caching system deployed to address the web-scale performance and scalability challenges. Two of its key components (client and server) function as a distributed key-value store that mutually resolve web servers' caching requirements. Fig. 1 shows the general architecture along with its main components. The Memcached client can interact with a number of servers for its data store and data retrieval purposes. As a distributed caching system, Memcached incorporates several key architectural aspects indispensable to the design of burst buffer system. First, the Memcached client adopts a two-stage hashing mechanism for balanced data placement. Second, the Memcached server offers a lossy key-value store that involves all the major functionality of a local storage system, such as space, data and metadata management.

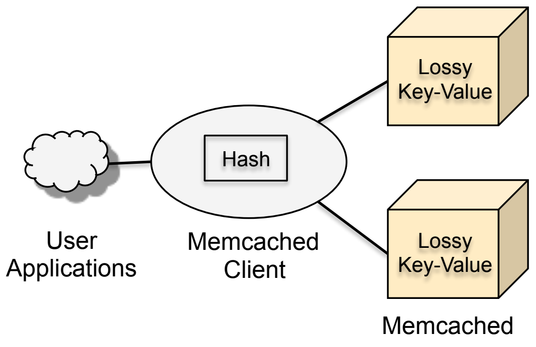


Fig. 1: Component diagram of Memcached

1) Two-stage Hashing: Memcached applies a two-stage hashing mechanism to store or retrieve a certain key-value pair (KVP). In the first stage, the key is hashed to the server responsible for storing the KVP. Once the KVP is stored on the server, it is further hashed to an entry in the server's local hash table that records the address of this KVP.

2) A Lossy Key-Value Store with Disconnected Servers: Memcached server is designed as a simple but powerful in-memory key-value store. A server pre-allocates many groups of 1MB slabs, each slab contains serveral chunks. Chunks that belong to the same group are of equal size, but they are different among different groups. Each group possesses a unique ID ranked from 0 to 42. The chunk size for different group increases with a factor of 1.25 with Group IDs. For instance, Group 1 contains chunks sized 96B, Group 2 contains chunks sized 96B*1.25, and so on. Thus, Group 42 contains chunks sized $96B * 1.25^{41} = 1048576B$, which is exactly 1MB. So each slab only contains one chunk in Group 42. To insert a KVP, Memcached server selects a chunk from the group with the closest chunk size and copy it to such chunk, the chunk address is then recorded on the hash table. Upon a conflict, a chained list of entries are provided to hold the address of multiple KVPs.

B. Challenges from Scientific Applications

For its simple, scalable and powerful design and performance, Memcached has been used by many web applications that require fast cache storage for their temporary data. The requests of these applications are generally distributed, arriving in a random order, with very little synchronization.

While scientific applications also generate large volumes of data, their data patterns are significantly different from the web applications. Particularly, they possess a number of distinct characteristics described below that warrant a new perspective on how to leverage the strengths of Memcached.

- **Lock-step I/O from many synchronized clients**: Scientific applications typically consist of many parallel processes who enter their I/O phase in a lock-step manner. These processes have frequent and well coordinated sychronization which means that processes need to exchange data with their neighbors. Their I/O operations are also closely synchronized.
- **Bursty and non-overlapping I/O**: Scientific applications usually have well-defined execution phases. For example, they alternate between computation and I/O phases. This characteristic provides the fundamental requirement for designing burst buffer. The file extent that each process writes on does not overlap. In each I/O phase, a process does not rewrite the content already written.
- **Frequent writes and few reads**: Scientific applications periodically create snapshots (via checkpointing) of their intermediate results and datasets. They are typically write-intensive but read only sparingly. In-memory variables such as arrays and meshes are written at each snapshot. Checkpoint data is only read during application restart. Given these characteristics, we design burst buffer mainly for application write throughput. This is different from many other existing buffering systems such as PredatA [34], DataSpaces [12] in ADIOS [31] that focus on in-situ data sharing and analysis for scientific simulation.

The distinct features of scientific applications lead us to rethink the design of Memcached. In this paper, we carry out a study on the design of the burst buffer system on top of Memcached framework. While preserving many features of Memcached architecture, we modify Memcached from three critical aspects: storage management, coordination with the PFS, and the communication efficiency, respectively.

III. BURSTMEM: A BURST BUFFER FRAMEWORK ON TOP OF MEMCACHED

In this section, we first present an architectural overview of the proposed burst buffer system called BurstMem. Then we elaborate on internal details of system components.

A. Software Architecture of BurstMem

Fig. 2 shows the software architecture of BurstMem and its relationship with other system components in a typical HPC environment. As a data buffering system, BurstMem is located between the processing elements and the backend persistent storage hosted by the PFS. It connects to all application

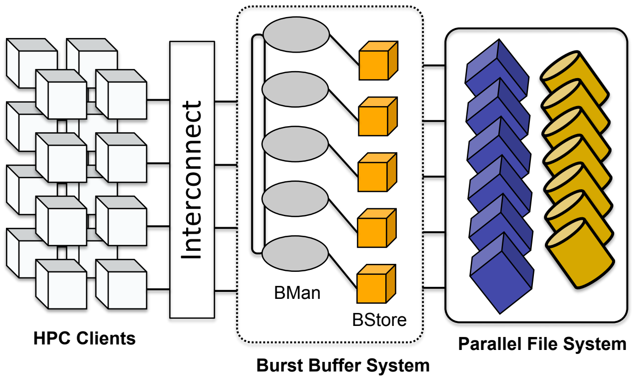


Fig. 2: Software architecture of a burst buffer system

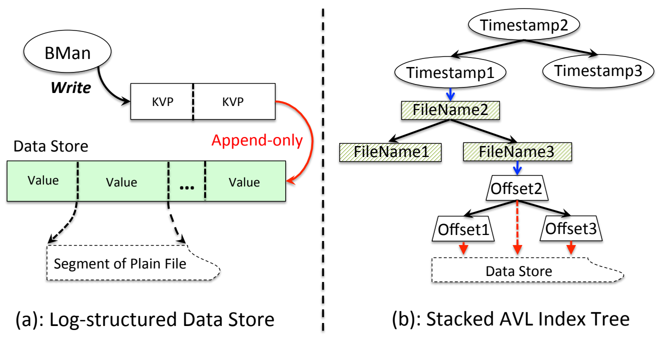


Fig. 3: Data structures for absorbing writes

processes via a high-speed interconnect, temporarily buffers bursty datasets from these processes, gradually flushing the datasets to the PFS.

BurstMem is composed of two main components: BurstMem Managers (*BMan*s) and BurstMem Stores (*BStore*s). Each BMan is designed with a BStore as an internal Memcached server. All BMans form a parallel set of burst buffer daemons to intercept application data. Each BMan keeps track of the address and health status of neighboring BMans. These BMans are in charge of the bulk of responsibilities for system maintenance and resource management. They coordinate all the BStores for fast data buffering, balanced data distribution and long-term persistent storage.

Using BurstMem, scientific applications can follow a new checkpoint flow. After each phase of computation, it coordinates with BMan for checkpointing, BMans absorb and store all the checkpoint dataset to BStore. The application then returns to the computation of next phase, leaving the ensuing data flushing operation to BStore. In this way, data flushing is overlapped with computation.

BurstMem is built on top of Memcached, it exposes to application a simple checkpoint API, which invokes a customized light-weight Memcached client library for data shipping. Once BMan receives the KVP, it leverages BStore for storage management. BStore follows the same data processing flow as Memcached server's storage management. It allocates memory for the accepted KVP, records its location so that the KVP can be retrieved in the future.

IV. Internal Design

The goal of BurstMem is to efficiently absorb large amounts of write requests, and provide high-throughput service to migrate the data into the PFS. First, we review how BurstMem copes with bursty writes, then describe our strategy to flush the data from BurstMem to the underlying PFS.

A. *Log-Structured Data Organization with AVL indexing*

We introduce a **L**og-**S**tructured data organization with Adelson-Velskii and Landis (**A**VL) tree [18] based indexing (LSA) to absorb the large amount of bursty write requests. LSA resolves three major issues in Memcached that prevent it from achieving efficient write. First, the original Memcached preallocates fix-sized memory chunks to accommodate the incoming write requests. However, this results in underutilized memory resources when a write request is not aligned with the memory chunk, and additional memory allocation is needed each time the chunks with a given size are used up. Second, contemporary HPC platforms are embracing tiered storage with both DRAM and SSD, Memcached is oblivious to this architecture. Third, the hash-based indexing used by Memcached is unable to support range queries, a requirement for bulk data flushing and HPC application restart.

The key idea of LSA is to compact the received write requests following an append-only manner to avoid memory waste, as shown in Fig. 3 (a). When used memory reaches a high watermark, we append in-memory data to SSD.

As illustrated in Fig. 3(a), we design a hierarchical data store to log the concrete file data (value) from write requests. A large (by default 4GB) DRAM block is maintained for logs at the first level, a separate intermediate file is reserved for logs on SSD. The address space of data store covers the storage from both the DRAM and SSD.

Upon its arrival in BMan, each write request is converted into a KVP. The key records the information that uniquely identifies the request, including the current checkpointing timestamp, the name of its targeted checkpoint file, and the offset (i.e. position of the write request on the checkpoint file) as well as the length of the value. The value points to the concrete data (e.g. a segment of a plain file) in the write request to be stored into data store and then flushed to the checkpoint file, as shown in Fig. 3(a). When BStore receives a write request, it separates the key from the value and appends the value to the end of the data store. By doing so, BStore eliminates the random access caused by following a strict order of checkpoint file offsets, and maximizes the write throughput.

In order to facilitate data retrieval (e.g. application restart) from the data store, all the keys for each KVP are organized in a stacked AVL-tree structure that records the metadata of absorbed write requests. An AVL-tree [18] is a self-balancing binary search tree that supports lookup, insertion and deletion with $O(\log N)$ complexity for both average and worst cases, thus achieving better performance than binary search trees. It also delivers an ordered node sequence that allows in-order traversal. Although inheriting many AVL's virtues, our design differs greatly from the conventional AVL tree, exhibiting a stacked structure. It consists of three categories of layers: *timestamp*, *filename*, and *offset*, as shown in Fig. 3(b). Each write request is first indexed by the *timestamp*, then the *filename*, and finally by the *offset* pointing to its position in the checkpoint file to be stored on the PFS. A pointer recording the address of each write request in the data store is maintained together with the *offset* index. The intuition behind such design is to accelerate retrieval and traversal for the data in a specific range. Take checkpointing as an example, flushing the dataset that belongs to a single timestamp is important. Our stacked AVL tree allows to locate such dataset in a single range search operation. After pinpointing the index of such timestamp, each *filename* subtree under such index is traversed, restoring the order of each checkpoint file through an in-order traversal of all its *offset* metadata. Taken together, such tree index supports diverse query patterns. For example, retrieving all the data under timestamp 1, from timestamp 1 to 3, or under timestamp 1 belonging to filename 1, etc. These query patterns are not supported by hash-based indexing in Memcached.

However, one major issue faced by LSA is to determine when to conduct the garbage collection necessary to reclaim the used space in data store and to trim the stacked AVL tree. We address such issue by leveraging a key characteristic of checkpointing. After completing a checkpointing operation, the data belonging to a specific timestamp can be discarded since they have been flushed to the underlying file system. Thus we first mark the timestamp node on the AVL tree as unused. A process is invoked periodically in the background to traverse the tree for those unused timestamps and compact the in-memory data store to reclaim the memory space used by values of such timestamp. When all values within the timestamp have been recycled, we trim the timestamp subtree off our stacked AVL tree. In addition, we leave the log on the SSD untouched. Only when the size of the log on SSD is close to a threshold, and all the data within the log has been transfered into the PFS, do we discard the log in its entirety and generate a new log to absorb the data from memory.

B. Coordinated Shuffling for Data Flushing

BurstMem is responsible for flushing the data into the PFS. In the current design, data flushing takes place after checkpointing. We also allow clients to trigger the data flushing explicitly. There are two general checkpointing patterns in scientific applications, $N-N$ and $N-1$ checkpointing. In N-N checkpointing every process writes to a separate file. In N-1 checkpointing, all processes write to a single shared file. BurstMem supports both patterns. Under the N-N pattern, each client's checkpoint data is hosted by one BStore. These BStores can flush data into different checkpoint files without interfering each other. In contrast, under N-1 checkpoint pattern, the shared checkpoint file spreads across many BStores. Naively flushing data content into a shared file can incur significant lock overhead, leading to drastically degraded throughput. Coordinated shuffling is applied to address such issue for the N-1 case.

Before elaborating the coordinated shuffling, we briefly describe the cause of lock overhead. Many PFSs use distributed locks to guarantee data consistency. Taking Lustre as an example, locking is performed at the granularity of Lustre stripes (by default 1 MB). When there is a need to write a stripe for data flushing, a BStore needs to first acquire the lock for that stripe. If another BStore owns the lock, Lustre has to revoke the prior ownership before granting the lock ownership to the first BStore. Once the stripe lock is acquired, the lock and the data are buffered in the Lustre Object Storage Client side (OSC) inside the BStore. Under the N-1 case, write requests from multiple BStores may overlap on the same stripe, causing frequent ownership changes on the stripe lock. Associated with the change of lock ownership, data flushing can cause frequent network traffic and delay the entire process. Therefore, the contiguous, stripe-aligned write requests is preferred compared to the noncontiguous, stripe-unaligned write requests since the former can efficiently reduce the degree of lock contention.

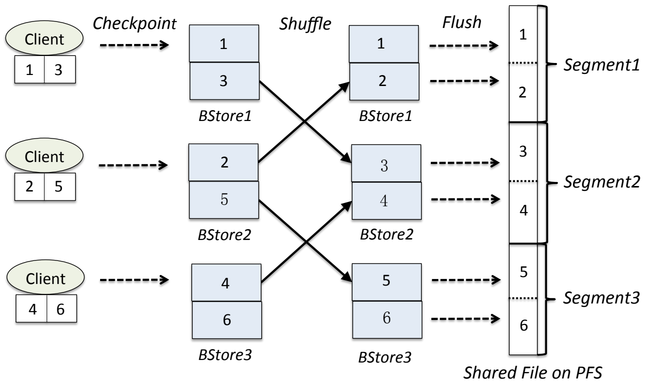


Fig. 4: Coordinated shuffling for N-1 data flushing

In most of our targeted cases, each BStore possesses several noncontiguous, small segments for the shared file, thus amplifying the lock overhead. Therefore, *coordinated shuffling* is designed to reshuffle the segments among BStores so that each BStore can flush contiguous segments into the PFS with alleviated lock contention.

As illustrated in Fig. 4, each shared file is logically divided into several contiguous segments. The total number of segments is equal to the number of BStores. The purpose of data shuffling is to have each BStore store all the data that belong to the same segment. Fig. 4 details this process. BStores 1, 2 and 3 possess data chunks from Client 1, 2 and 3, respectively. These chunks belong to the same shared file. This file is divided into 3 segments, which is mapped to the three BStores. Before data shuffling, each BStore stores noncontiguous data chunks. Data shuffling begins after such mapping is established. Following this mapping, Chunk 2 is shuffled from BStore 2 to BStore 1, Chunk 3 is shuffled from BStore 1 to BStore 2, and so on. Once shuffling operation is completed, each BStore can then flush the data to the PFS.

Our data flushing scheme is inspired by the idea of ROMIO [30]. We do not directly use ROMIO library since it is coupled with MPI environment. Such environment restricts BurstMem's potential for future extention on fault tolerance.

C. Enabling Native Communication Performance

Memcached relies on the BSD Sockets interface and uses the reliable stream (*i.e.* TCP) to transfer data. Although Sockets eases the implementation, performance of socket-based communication cannot fully exploit the advantage of leadership scale HPC systems, such as Remote Direct Memory Access (RDMA), or OS-bypass. In addition, its performance is not optimized and is highly subject to data sizes.

Therefore, we have employed the *Common Communication Interface* (CCI) [6] to efficiently leverage the performance advantage of HPC facilities. It is designed by Oak Ridge National Laboratory. It exposes the performance of using native network interfaces to scientific applications. CCI has now been fully deployed on Titan supercomputer to serve various scientific applications. In our optimization, we leverage CCI to accelerate checkpointing from clients to BMans, as well as data shuffling among BMans.

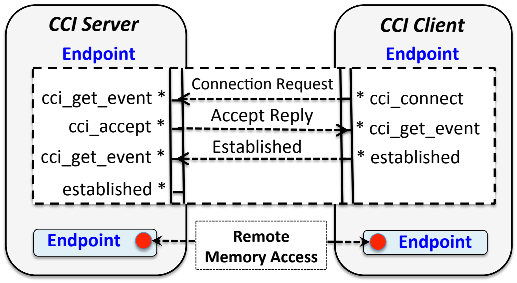


Fig. 5: CCI-based network communication among BMans

Figure 5 illustrates our implementation of CCI-based network communication. CCI uses client/server semantics to establish a connection. Checkpointing or data shuffling triggers a connection request to the peer. In both Server and Client, CCI abstracts a network device as the *Endpoint*, which is a virtualized device containing many resources, such as send and receive queues, as well as the buffers that are associated with the queues. Once the connection is established, we leverage the Remote Memory Access (RMA) feature that enables zero-copy in CCI to transfer the data over the network.

In our design, we use an event-driven model to improve the throughput. We poll the CCI Endpoint for new events (via cci_get_event). On the client side, one thread is dedicated to establishing connections with remote servers. Meanwhile, data-transferring thread uses non-blocking RMA to transfer the data. RMA is typically one-sided (i.e. only the initiator is actively involved in the transfer). In order to notify the completion of a RMA operation, we use the completion message option that sends a message and generates a receive event on the remote endpoint. A central thread, which polls the events from the Endpoint, orchestrates all of above threads. Similarly, on the server side, a thread detects events from the Endpoint and dispatches the requests, e.g. connection requests or completion events of RMA, to the corresponding threads for further processing.

V. Experiment Evaluation

A. Methodology

Testbed: All experiments are conducted on the Titan supercomputer [3] hosted at Oak Ridge National Laboratory. Each node is equipped with a 16-core 2.2GHZ AMD Opteron 6274 (Interlagos) processor, 32 GB of RAM, and a connection to the Cray custom high-speed interconnect. Two nodes share 1 *Gemini* high-speed interconnect router. Since there is no I/O server deployed for I/O buffering, we use a separate set of compute nodes for BurstMem. Out of the 256 allocated to the experiment, 128 of the compute nodes are used as clients that write data into the BurstMem. The other 128 compute nodes are allocated as the BurstMem servers. In every experiment, we place one process on one physical node.

Titan is connected to Spider II, a center-wide Lustre-based file system. It features 30 PB of disk space, offering 1 TB/s aggregated bandwidth organized in two non-overlapping identical file systems, each providing 500 GB/s I/O performance. The default stripe size of each created file is 1 MB. The default stripe count is 4.

In all the experiments, we have pinned 16 MB DRAM buffer for each RMA channel among two communication entities.

Benchmarks: To evaluate the performance of BurstMem, we have employed a synthetic workload using IOR [23] and also a real-world scientific application called S3D [10]. We report the average of 5 test run results.

IOR is a flexible synthetic benchmarking tool that is able to emulate diverse I/O access patterns. It was initially designed for measuring the I/O performance of parallel file system (PFS). We add BurstMem support to IOR by redirecting all writes from the processes to BurstMem instead of the PFS. This new version of IOR is referred to as *BB-IOR*. To emulate bursty I/O behavior as described in Section II-B, we set *interTestDelay* to 20 seconds between any two I/O phases, and iterate 10 times. For comparison, we also redirect the writes to Memcached, refered to as MemCache-IOR.

To evaluate the performance of real applications, we have integrated BurstMem into S3D, which we refer to as *BB-S3D*. S3D is a parallel turbulent combustion application using a direct numerical simulation solver developed at Sandia National Laboratories. It solves fully compressible Navier-Stokes, total energy, mass continuity equations coupled with detailed chemistry. The problem domain is a conventional 3-D structured Cartesian mesh. All the MPI processes are partitioned along the X-Y-Z dimensions. S3D exhibits bursty patterns during the execution. Its checkpointing phase alternates with computation regularly. Each checkpointing phase outputs four global arrays representing the variables of mass, velocity, pressure and temperature.

B. Ingress Bandwidth

We first investigate the ingress bandwidth that BurstMem can support to absorb the write requests. We use IOR bench-

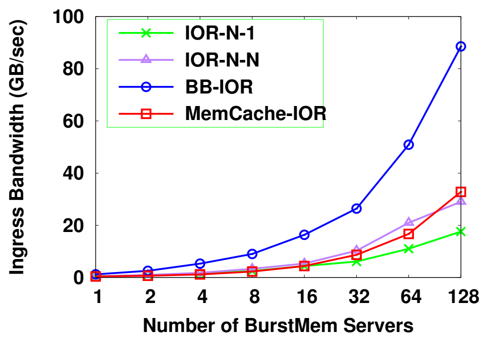


Fig. 6: Ingress I/O bandwidth versus number of BurstMem servers

mark and increase the number of BurstMem servers from 1 to 128. In each test, we use the same number of IOR clients as that of BurstMem servers to stress the system. We use 1MB (default stripe size) transfer unit to alleviate the lock contention issue in the Lustre file system. We have IOR-N-1 and IOR-N-N respectively represent the N-1 and N-N pattern as mentioned in Section IV-B for original IOR. In the IOR-N-1 case, we set the stripe count as the number of clients. For fair comparison, we set the stripe count as 1 for IOR-N-N case so that same number of Object Storage Targets (OST) are utilized as the number of clients. On average, each IOR client writes 4 GB data.

Figure 6 compares the ingress bandwidth IOR receives with and without BurstMem support, as well as MemCache-IOR. Overall, BurstMem delivers significantly higher ingress bandwidth than the other three alternatives. As seen in Figure 6, BB-IOR is able to achieve 278.2%, 246.9% and 174.5% improvement on average, when compared to the IOR-N-1, MemCache-IOR and IOR-N-N respectively. Such improvement is consistent across different number of BurstMem servers.

BB-IOR achieves substantial improvement over the original IOR by buffering the write requests instead of writing directly to Lustre file system. Such performance improvement is what we expect. However, as also shown in Figure 6, simply using Memcached as the buffering system cannot maximize the ingress bandwidth. BurstMem effectively outperforms Memcached with LSA described in IV-A and CCI support in IV-C. Specifically, BurstMem benefits from Gemini's native transport using CCI and avoids frequent memory allocation using LSA.

To further examine the ingress bandwidth under different workloads, we reduce the number of BurstMem servers to 4, which equals the default stripe count. We also set stripe count as 4 and have all the clients write on one shared file for the original IOR. In both BB-IOR and the original IOR, we increase the number of IOR clients from 4 to 128, thereby increasing the workload per BurstMem server and OST. Figure 7 illustrates the performance comparison with respect to increasing number of IOR clients. On average, BB-IOR outperforms the original IOR and MemCache-IOR by 508.62% and 408.30%, respectively. We observe an increasing bandwidth from 4 to 16 IOR clients. This is because when the number of IOR clients is fewer than 16, the supplied bandwidth of each BurstMem server is not fully saturated. Once the number reaches 16, such bandwidth is fully utilized, and BurstMem is able to provide stable ingress bandwidth regardless of the workloads.

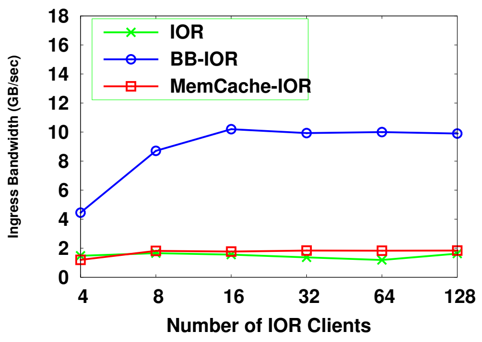


Fig. 7: Ingress I/O bandwidth versus number of I/O clients

C. Egress Bandwidth

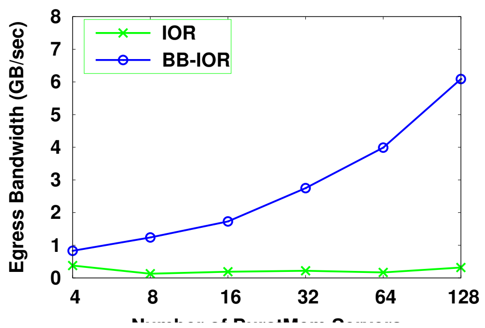
Efficiently flushing data to the PFS to spare space for future writes is another essential feature for BurstMem. In this section, we measure the performance of egress bandwidth and evaluate the effectiveness of coordinated shuffling introduced in Section IV-B. To emulate the common I/O access pattern in scientific applications, we interleave the writes from multiple IOR clients and use 16 KB transfer size, one of the dominant transfer sizes for scientific applications [17]. Similar to ingress bandwidth evaluation, we set the number of IOR clients equal to that of BurstMem servers and have each client output 4 GB of data to a shared file. Because Memcached does not support flushing, the comparison does not include Memcached.

Figure 8 shows the performance of egress bandwidth. Cumulative egress bandwidth of BB-IOR increases from 0.83 GB/s at 4 processes to 6.09 GB/s at 128 processes. BB-IOR is able to achieve 2× to 19× higher bandwidth when compared to the original IOR whose performance is consistently below 0.4 GB/s for all cases. Such low performance is mainly due to large overhead from lock contention caused by unaligned writes. Coordinated shuffling rearranges unaligned write requests into sequential, stripe-aligned writes, thereby significantly improving the overall egress bandwidth.

In Figure 9, we show the time spent on shuffling and flushing. The shuffling operation can incur over 30% overhead. However, it enables flushing to achieve better performance due to large, sequential writes to PFS in a stripe-aligned manner and delivers orders of magnitude better performance at massive scale. Hence, extra overhead on data shuffling is worth the trade-off given the significant benefit it delivers.

D. Scalability

Scalability is a critical factor for BurstMem. We want to ensure that BurstMem is able to provide increasing bandwidth when given more resources; such as more BurstMem nodes



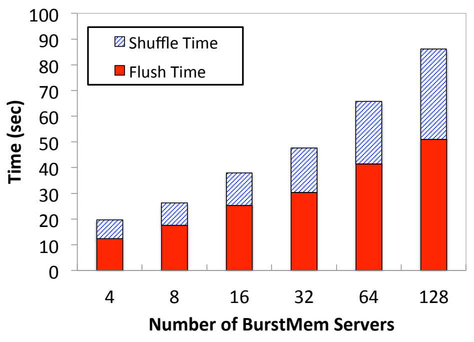


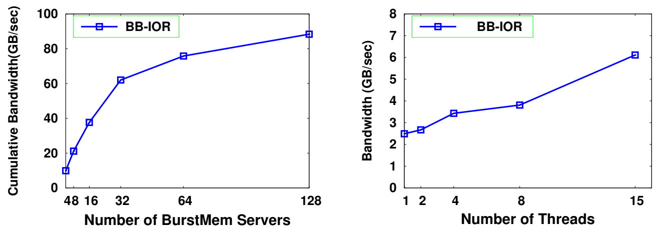
Fig. 9: Dissection of coordinated shuffling process

and additional CPU cores on each node. In this section, we evaluate the scalability of BurstMem from two perspectives, horizontal scalability (scale-out), vertical scalability (scale-up). We continue using IOR as the benchmark tool for our evaluation. In the horizontal scaling experiment, we increase the number of BurstMem servers and measure the cumulative bandwidth delivered by BurstMem. In the vertical scalability experiment, we increase the number of threads in each individual BurstMem server.

1) **Horizontal Scaling:** We evaluate the Horizontal scalability by fixing the number of clients as 128, and increase the number of BurstMem servers from 4 to 128. The I/O request size is set to 1 MB, each client writes 512 MB of data, featuring 64 GB of input data in total for each iteration.

Figure 10 (a) shows the performance results of horizontal scaling. As shown in the figure, cumulative bandwidth improves linearly from 9.9 GB/s with 4 BurstMem servers to 62.04 GB/s with 32 BurstMem servers. However, the increasing rate declines when going from 64 to 128 BurstMem servers. This is because the supplied bandwidth of each BurstMem server can efficiently absorb I/O requests from more than 2 clients. When the number of BurstMem servers is fewer than a quarter of the clients (32), they are mostly saturated. However, further increasing the number of BurstMem servers from that point (32 servers) leads to underutilized bandwidth provided by BurstMem system, and the bandwidth gradually becomes client-bound.

In summary, the linear horizontal scalability is achievable when ingress bandwidth is bounded by BurstMem. In addition, there are some other factors that can affect cumulative bandwidth, including varying end-to-end network bandwidth on Titan due to locality, and the contention of network resources. These factors cause the cumulative bandwidth to be lower than the theoretical maximum bandwidth.



(a) Horizontal Scaling (b) Vertical Scaling

Fig. 10: Scalability

2) **Vertical Scaling:** We evaluate BurstMem's vertical scalability by scaling the number of threads in each BurstMem server from 1 to 15, one thread per core. The remaining one core is used to run system daemons. We measure the bandwidth that can be supplied by each individual BurstMem. On average, each BurstMem serves 16 clients. Each client sends 512 MB data to the server, featuring 8 GB input data for each BurstMem server. Figure 10 (b) shows the bandwidth increasing from 2.49 GB/s at one thread to 6.11 GB/s at 15 threads. There is a sharp increase at 15 processes because each compute node contains 2 NUMA nodes, and each NUMA node contains 8 cores. Titan schedules the first 8 threads to the first NUMA node and the last 7 threads to the second NUMA node. When we use 15 threads, we include the capability from another NUMA node; such as memory bandwidth and computing power.

E. Case Study: S3D, A Real-World Scientific Application

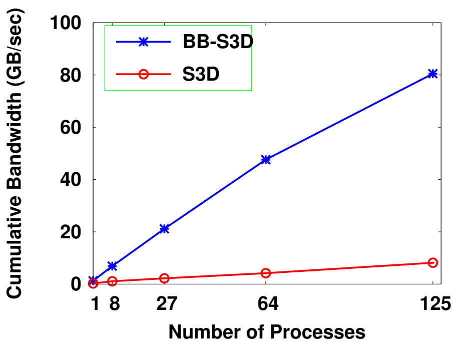


Fig. 11: S3D application I/O performance evaluation with BurstMem

During the experiments with S3D, we keep the size of X, Y, Z dimensions as 50, 50, 50 respectively and have each process write about 2 GB checkpoint data. We have compared the cumulative bandwidth of BurstMem-enabled S3D (BB-S3D) with that of the original S3D implementation.

Figure 11 shows the I/O performance comparison between BB-S3D and S3D. The bandwidth of BB-S3D increases linearly from 1.27 GB/s at 1 process to 80.49 GB/s at 125 processes. This yields a performance improvement of up to 10× over the original S3D when the number of MPI processes is 125.

We have observed that the original S3D bandwidth is lower than that of the original IOR. This is because, in IOR tests, we set the transfer size as the stripe size, which optimizes the performance under Lustre file system. In contrast, the transfer units of Fortran I/O in S3D varies from 0.95 MB, 2.86 MB to 10.49 MB). This is less favored by Lustre.

VI. Related Work

Improving I/O performance on large-scale HPC systems has gained broad attention over the past decades.

A number of studies have introduced new I/O middleware libraries. MPI-IO [30], PnetCDF [19, 22], HDF5 [2] boost I/O performance using parallel I/O that involves a massive number of participating processes. PLFS [8] introduces an extra I/O layer that converts the noncontiguous, interspersed I/O into contiguous, sequential I/O. All these studies aim to optimize I/O on the parallel file system (PFS). Therefore, their performance is still restricted by the bandwidth of PFS.

I/O forwarding [5] is another key technique applied on Blue Gene/P systems. It leverages two I/O forwarding components, named CIOD [24] and ZOID [16], both of which use synchronous I/O forwarding. Venkatram et al. [32] replaces synchronous I/O forwarding with asynchronous staging, thus enhancing an application's overall performance. However, such techniques only applies to the Blue Gene/P architecture.

Orthogonal to this work, asynchronous data staging is proposed by many other researchers. Such work generally falls into two categories: local staging [27, 20] and remote staging [25, 4]. In the former case, an application uses local storage of compute nodes as staging area. However, the performance can be highly affected by computation jitters, as any perturbation caused by asynchronously copying data from local storage to parallel file system can cascade through the tightly-coupled computation [7]. Remote staging buffers I/O in additional partitions of compute nodes. Although remote staging is immune to computation jitters, it is confined by available resources of compute nodes, such as the supplied bandwidth and storage capacity.

Burst buffer in HPC is relatively recent idea. Currently, most work on burst buffer stays at theoretical stage. Liu et al. [21] designed a simulator of burst buffer for the IBM Blue Gene/P architecture. Bing et al. [33] characterized output burst absorption on Jaguar and made an important step toward quantitative models of storage system performance behaviors. Different from them, our work focuses on designing and implementing a prototype burst buffer system and analyzing its performance benefit.

VII. Conclusion

In this paper, we have designed a high-performance burst buffer system on top of Memcached. Through in-depth analysis, we have identified that Memcached has many issues to be directly used as burst buffer, such as the lack of efficient storage management to absorb large amounts of bursty writes and the incapability to exploit modern high-speed network interconnects. Based on our analysis, we introduce several techniques to enhance Memcached as the BurstMem system for bursty I/O in scientific applications. Our techniques include a log-structured data organization with stacked AVL indexing for fast I/O absorption and low-latency, semantic-rich data retrieval, coordinated data shuffling for efficient data flushing, and CCI-based communication for high-speed data transfer. Our experiments on the Titan supercomputer with synthetic benchmark and real-world applications demonstrate that BurstMem can efficiently provide high-performance I/O services to current HPC scientific applications with good scalability.

Our future work will focus on optimizing BurstMem's data flushing operation by reducing the amount of data being shuffled, and providing fault tolerance to BurstMem. We will also extend BurstMem's support for more file formats such as NetCDF and HDF5.

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] The ASC sequoia draft statement of work. https://asc.llnl.gov/sequoia/rfp/02_SequoiaSOW_V06.doc.

[2] HDF5. http://www.hdfgroup.org/HDF5/.

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

[4] 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.

[5] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan. Scalable I/O forwarding framework for high-performance computing systems. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–10. IEEE, 2009.

[6] S. Atchley, D. Dillow, G. Shipman, P. Geoffray, J. M. Squyres, G. Bosilca, and R. Minnich. The common communication interface (CCI). In *High Performance Interconnects (HOTI), 2011 IEEE 19th Annual Symposium on*, pages 51–60. IEEE, 2011.

[7] J. Bent, S. Faibish, J. Ahrens, G. Grider, J. Patchett, P. Tzelnic, and J. Woodring. Jitter-free co-processing on a prototype exascale storage stack. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–5. IEEE, 2012.

[8] 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 *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 21. ACM, 2009.

[9] P. J. Braam and R. Zahir. Lustre: A scalable, high performance file system. *Cluster File Systems, Inc*, 2002.

[10] J. H. Chen, A. Choudhary, B. de Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W. K. Liao, K. L. Ma, J. Mellor-Crummey, N. Podhorszki, R. Sankaran, S. Shende, and C. S. Yoo. Terascale direct numerical simulations of turbulent combustion using S3D. *Computational Science & Discovery*, 2(1):015001, 2009.

[11] K. Chodorow. *MongoDB: the definitive guide*. " O'Reilly Media, Inc.", 2013.

[12] C. Docan, M. Parashar, and S. Klasky. Dataspaces: an interaction and coordination framework for coupled simulation workflows. *Cluster Computing*, 15(2):163–181, 2012.

[13] J. Dongarra. Impact of architecture and technology for extreme scale on software and algorithm design. In *The Department of Energy Workshop on Cross-cutting Technologies for Computing at the Exascale*, 2010.

[14] J. Fu, M. Min, R. Latham, and C. D. Carothers. Parallel I/O performance for application-level checkpointing on the Blue Gene/P system. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 465–473. IEEE, 2011.

[15] L. George. *HBase: the definitive guide*. " O'Reilly Media, Inc.", 2011.

[16] K. Iskra, J. W. Romein, K. Yoshii, and P. Beckman. ZOID: I/O-forwarding infrastructure for petascale architectures. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 153–162. ACM, 2008.

[17] Y. Kim, R. Gunasekaran, G. M. Shipman, D. A. Dillow, Z. Zhang, and B. W. Settlemyer. Workload characterization of a leadership class storage cluster. In *Petascale Data Storage Workshop (PDSW), 2010 5th*, pages 1–5. IEEE, 2010.

[18] 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.

[19] J. Li, W.-k. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netCDF: A high-performance scientific I/O interface. In *Supercomputing, 2003 ACM/IEEE Conference*, pages 39–39. IEEE, 2003.

[20] W.-k. Liao, A. Ching, K. Coloma, A. Nisar, A. Choudhary, J. Chen, R. Sankaran, and S. Klasky. Using MPI file caching to improve parallel write performance for large-scale scientific applications. In *Supercomputing, 2007. SC'07. Proceedings of the 2007 ACM/IEEE Conference on*, pages 1–11. IEEE, 2007.

[21] 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.

[22] 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 *Computer Communications and Networks (ICCCN), 2013 22nd International Conference on*, pages 1–7. IEEE, 2013.

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

[24] J. Moreira, M. Brutman, J. Castaños, T. Engelsiepen, M. Giampapa, T. Gooding, R. Haskin, T. Inglett, D. Lieber, P. McCarthy, et al. Designing a highly-scalable operating system: The Blue Gene/L story. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 118. ACM, 2006.

[25] 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.

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

[27] R. Rajachandrasekar, A. Moody, K. Mohror, and D. K. Panda. A 1 PB/s file system to checkpoint three million MPI tasks. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 143–154. ACM, 2013.

[28] R. B. Ross, R. Thakur, et al. PVFS: A parallel file system for linux clusters. In *in Proceedings of the 4th Annual Linux Showcase and Conference*, pages 391–430, 2000.

[29] F. B. Schmuck and R. L. Haskin. GPFS: A shared-disk file system for large computing clusters. In *FAST*, volume 2, page 19, 2002.

[30] 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.

[31] 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 *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*, pages 1–10. IEEE, 2013.

[32] V. Vishwanath, M. Hereld, K. Iskra, D. Kimpe, V. Morozov, M. E. Papka, R. Ross, and K. Yoshii. Accelerating I/O forwarding in IBM Blue Gene/P systems. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–10. IEEE, 2010.

[33] B. Xie, J. Chase, D. Dillow, O. Drokin, S. Klasky, S. Oral, and N. Podhorszki. Characterizing output bottlenecks in a supercomputer. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11. IEEE, 2012.

[34] F. Zheng, H. Abbasi, C. Docan, J. Lofstead, Q. Liu, S. Klasky, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf. Predata–preparatory data analytics on peta-scale machines. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.