2017 IEEE International Parallel and Distributed Processing Symposium

MetaKV: A Key-Value Store for Metadata Management of Distributed Burst Buffers

Teng Wang† Adam Moody‡ Yue Zhu† Kathryn Mohror‡ Kento Sato‡ Tanzima Islam‡ Weikuan Yu†
†Florida State University ‡Lawrence Livermore National Laboratory
{twang, yzhu, yuw}@cs.fsu.edu {moody20, kathryn, kento, tanzima}@llnl.gov

***Abstract*—Distributed burst buffers are a promising storage architecture for handling I/O workloads for exascale computing. Their aggregate storage bandwidth grows linearly with system node count. However, although scientific applications can achieve scalable write bandwidth by having each process write to its node-local burst buffer, metadata challenges remain formidable, especially for files shared across many processes. This is due to the need to track and organize file segments across the distributed burst buffers in a global index. Because this global index can be accessed concurrently by thousands or more processes in a scientific application, the scalability of metadata management is a severe performance-limiting factor.**

In this paper, we propose MetaKV: a key-value store that provides fast and scalable metadata management for HPC metadata workloads on distributed burst buffers. MetaKV complements the functionality of an existing key-value store with specialized metadata services that efficiently handle bursty and concurrent metadata workloads: compressed storage management, supervised block clustering, and log-ring based collective message reduction. Our experiments demonstrate that MetaKV outperforms the state-of-the-art key-value stores by a significant margin. It improves put and get metadata operations by as much as 2.66× and 6.29×, respectively, and the benefits of MetaKV increase with increasing metadata workload demand.

I. Introduction

Distributed burst buffers are being deployed as a storage solution on many leadership-scale supercomputers [1], [4], [5], [6], [14], Their aggregate storage bandwidth scales linearly with compute node count. However, metadata operations over distributed burst buffers can quickly become a performance bottleneck. Typical HPC I/O workloads involve a large number (N) of processes accessing either N individual files (N-N workload) or a single shared file (N-1 workload). Both workloads have distinct metadata requirements. For N-N, the requirements are simple: each process writes an individual file to its node-local burst buffer, and the file metadata only needs to be managed locally. In contrast, for N-1, all processes concurrently write their segments of the shared file to their node-local burst buffers, with the consequence that segments of the shared file are distributed across burst buffers. Thus, in order for processes to locate shared file segments on subsequent accesses, we need a global index to track the segment locations.

Scientific applications using an N-1 I/O pattern can generate an enormous number of file segments in a single shared file [8], [13], resulting in an unmanageably large global index. Recently, several groups have researched using distributed key-value stores to manage the metadata on HPC systems [10], [16], [24], [28]. MDHIM [10] stands out as one of the few key-value stores developed for HPC systems, with several features that enhance its performance: it amortizes index workload over multiple key-value servers; it is implemented with MPI, for portability and leveraging native, high-speed transport layers; and it uses LevelDB [3], a fast persistent key-value store, to manage its key-value pairs and allow the pairs to be quickly reconstructed after failure. However, the performance of MDHIM is limited under bursty and concurrent metadata workloads. Primarily, this is due to MDHIM lacking awareness of the spatial and temporal locality of requests, resulting in a large quantity of small metadata operations on the same file (See Section II-B for details).

To address the limitations of metadata support for HPC N-1 workloads on distributed burst buffers, we developed MetaKV. MetaKV is a key-value store layered on top of MDHIM and improves on MDHIM's performance with several enhancements. MetaKV provides ordered index layout with fast writes using a novel compressed storage management framework, and reduces read service times via supervised read clustering. Moreover, to address the problem of highly-concurrent file attribute requests for shared files, it establishes a log-ring based overlay network across the servers to optimize common all-to-one and one-to-all communication patterns used for these metadata operations on shared files.

We make the following main contributions as part of our design, implementation, and evaluation of MetaKV.

- We introduce a novel compressed storage management framework to reshape the index layout without incurring frequent compaction operations.
- We design an efficient read clustering algorithm that dynamically clusters requested indices and services get requests, resulting in fewer and larger reads.
- We implement a log-ring based collective message reduction framework across key-value servers to optimize common all-to-one and one-to-all operations for shared file workloads.
- We evaluate MetaKV and show that it delivers fast and scalable metadata service under various scientific metadata workloads.

Overall, we show that MetaKV is a significant improvement

1530-2075/17 $31.00 © 2017 IEEE
DOI 10.1109/IPDPS.2017.39

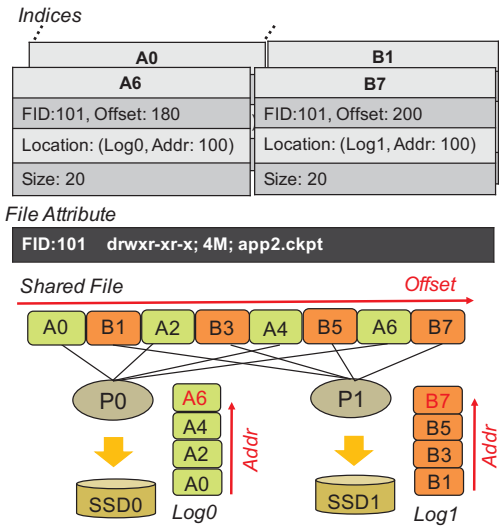


Fig. 1: File Attribute and Index Metadata Types.

over the state-of-the-art for HPC I/O workloads. MetaKV achieves as much as 2.66× and 6.29× performance improvement for put and get metadata operations, and the benefits of MetaKV are more pronounced for metadata workloads of increasing scales.

II. Background

In this section, we first present a high-level overview of the metadata challenges for distributed burst buffers. Then we elaborate on existing distributed key-value stores, and analyze the issues surrounding their use.

A. Metadata Challenges on Distributed Burst Buffers

Two primary types of metadata needed for distributed burst buffers are indices and file attributes, with an example of each type shown in Fig. 1. In the example in the figure, two processes concurrently write eight segments to a shared file. The segments written by each process are laid out contiguously in local log files (Log0 and Log1). Each segment has an index entry in the index table, where the index entry maps the logical offset of each segment in the shared file to a physical location on SSD. For instance, segment B7 is at offset 200 of the shared file, and it is mapped to physical address 100 of Log1. In addition, the index entry also records other important information, including segment size (20) and ID of the shared file (101). The attributes of the shared file are stored in the file attribute table, and include details such as file name, permission, and file size.

Handling each type of metadata and associated requests imposes distinct challenges. For file attribute metadata, the challenge arises from many identical, concurrent requests for a shared file. For example, if all processes concurrently create a shared file, all the processes send the file attribute information to the metadata server(s) at the same time. A similar situation occurs when all processes concurrently request or update metadata information, e.g., with a `stat` or `close` operation. This file attribute metadata problem has been described in several studies as a critical limiting factor of the system's scalability [24], [29], [32]. For the index metadata, the primary challenge occurs due to numerous concurrent I/O requests. Due to the nature of HPC I/O operations, shared file accesses may generate many small, concurrent write and read requests, followed by a large quantity of index operations to record those operations on the metadata servers.

B. Key-Value Stores for Metadata of Distributed Burst Buffers

Recently, distributed key-value stores have been explored as an alternative to improve metadata performance. These distributed key-value stores typically split a key space into numerous key ranges, and distribute these key ranges to all the key-value servers. Each key-value server is responsible for managing all the key-value pairs that fall in its own ranges.

In order to leverage key-value stores for metadata operations, the metadata has to be transformed into key-value pairs. For index metadata, using our example from Fig.1, the key can be a tuple (FID, Offset), and the value can be (LogID, Addr, Size). With this key-value format, we can retrieve the physical location (LogID, Addr) and the size of a segment based on its logical location (FID, Offset) in the shared file. For file attributes, we can use the FID as the key, and the value will correspond to the file attributes (e.g. file name, size in Fig. 1).

As discussed in Section I, MDHIM is a state-of-the-art key-value store for HPC systems. However, it suffers from several performance issues that we seek to overcome with MetaKV. Bursty write workloads often involve all processes concurrently issuing write requests, resulting in a large number of concurrent index *put* requests. These indices often have strong spatial locality [8], [18], [33] meaning the indices for contiguous segments of a file are written together to the same key-value server. However, since the index *put* requests are issued from multiple processes, when they arrive at MDHIM, the spatially contiguous indices are interleaved with other indices for different locations, resulting in disordered index layout. In order to recover the index layout, MDHIM introduces extra compaction overhead that can significantly prolong the index *put* time. Similarly, during a bursty read workload, the contiguous index *get* requests are directed to the same key-value servers and interleaved with other *get* requests. MDHIM does not realize the spatial locality and issues a separate read to retrieve each requested index, introducing massive small and redundant read operations that restrict the read performance. Moreover, concurrent file operations, e.g., `open`, `stat` or `close`, on the shared file often involve all the clients simultaneously sending the attributes to the same key-value server, and receiving the attributes from the same key-value server. This all-to-one/one-to-all communication pattern can present a serious hurdle to the system's scalability at scale.

MDHIM relies on LevelDB to manage the key-value pairs. Fig. 2 gives a high-level overview of how LevelDB manages the indices. The received indices are first appended to a write-ahead log (WAL), and also ordered in LevelDB's write buffer (*MemTable*) based on their keys. Once the MemTable is full,

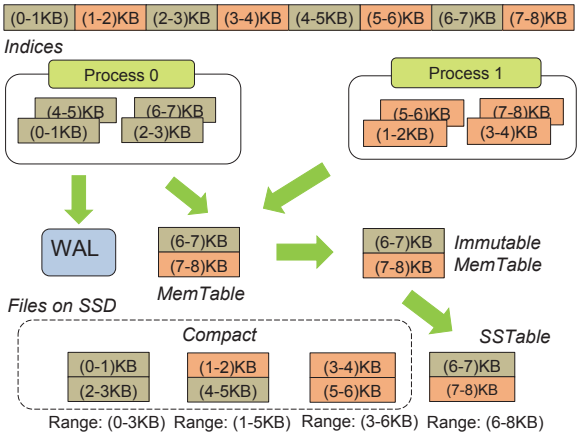
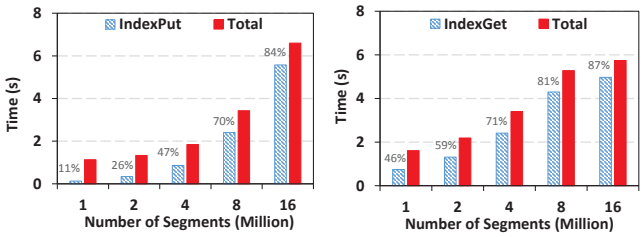


Fig. 2: Example of Index Metadata in LevelDB.



(a) Put Overhead (b) Get Overhead

Fig. 3: MDHIM Index Put and Get Overhead.

it is mirrored to a shadow write buffer as an *immutable MemTable*. This immutable MemTable is then flushed to the disk as an *SSTable*. LevelDB records the key range of each SSTable for fast search (see Fig. 2). The key ranges of different SSTables are often overlapping. This is because the contiguous indices are issued by different processes (e.g. in Fig. 2, (0-1)KB is from Process 0 and (1-2)KB is from Process 1). These contiguous indices arrive at the write buffer at a different time; when the write buffer is full, they are flushed to different SSTables, resulting in overlapping SSTables. When an index on the overlapping SSTables is requested, LevelDB has to go through multiple overlapping SSTables to search this requested index. To avoid repeated search operations, LevelDB intermittently compacts the overlapping SSTables into non-overlapping ones. For instance, in Fig. 2, the ranges of three SSTables are overlapping, so they are compacted together. More details about the compaction and the hierarchy of SSTables can be found in [22].

C. Motivating Study with MDHIM

To observe the potential issues of directly using MDHIM, we performed a small motivating study. In our experiments, we have 256 processes on 16 nodes concurrently write their file segments to a 16 GB shared file distributed over 16 SSDs following *N-1 strided write*, as depicted in Fig. 1 by P0 and P1. These processes then read this file back in the same pattern. We use MDHIM to store and query the indices of the file segments. We vary the size of each write request from 16KB to 1KB. This produces an increasing number of file segments, ranging from 1 million to 16 million. Fig. 3a and Fig. 3b reveal the metadata overhead for write and read, respectively. Here, each process is able to write/read segments of the shared file to/from the node-local burst buffer, delivering fast and sequential writes/reads. However, as we can see from Fig. 3a, the metadata put time is initially negligible, but it grows sharply with the segment count and gradually dominates the write time. This is counterintuitive since the size of index for each segment is 40B, which is much smaller than the smallest segment size (1KB).

After a detailed analysis, we find that a major reason for the increasing put overhead is due to more frequent compaction operations incurred at larger segment counts. When the segment index put operations are concurrently issued by all the processes, the indices for different file segments arrive at the MDHIM servers randomly. As a result, when these indices are stored in LevelDB, the spatially contiguous indices are separated into different SSTables (see Fig. 2), resulting in many overlapping SSTables. As we increase the segment count, more and more SSTables are generated, which results in more frequent compaction operations.

We also observed the index get time accounts for a high percentage of read time, and the ratio grows with segment count in Fig. 3b. This high overhead for getting indices is due to the growing number of random and redundant read operations on each key-value server with increasing segment count. Specifically, the read unit of LevelDB is a block (default 4KB). For each get, LevelDB loads the block containing the requested index and sequentially iterates over the block indices until it finds the target, resulting in numerous redundant seek and read operations.

III. MetaKV

In order to address the metadata challenges for distributed burst buffers, we propose MetaKV, a specialized key-value store on top of MDHIM to handle the management of indices and file attribute workloads. As we have elaborated in Section II-B, we have designed MetaKV to address several key scalability problems: compaction overhead, read overhead, and attribute handling overhead. MetaKV minimizes the compaction overhead via a novel *compressed storage management* framework for fast put. To eliminate the read overhead, MetaKV uses *supervised read clustering* to cluster the requested blocks into a few regions, and service the requested indices within these regions with fewer and larger read operations. Furthermore, in order to provide scalable file attribute service for metadata operations on a shared file, MetaKV employs *log-ring based collective message reduction* by organizing all the key-value servers on top of a log-ring overlay network and minimizing the number of messages exchanged by a single key-value server.

A. Compressed Storage Management

As we found in our motivating study in Section II-C, MDHIM suffers from the overhead of compaction, which is

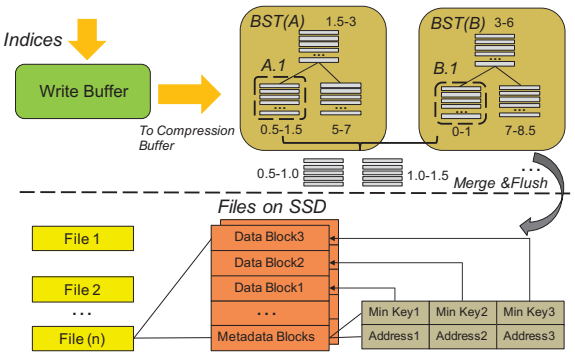


Fig. 4: Compressed Storage Management in MetaKV.

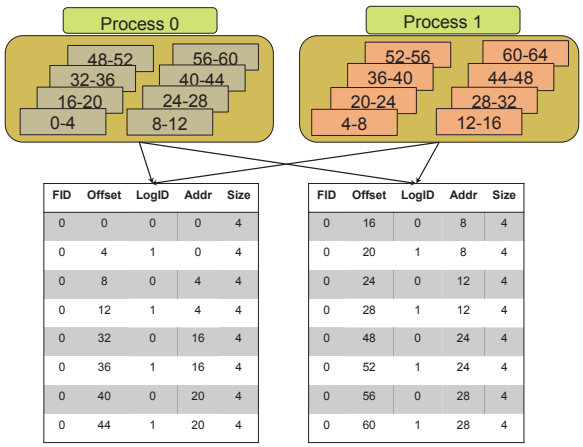


Fig. 5: Indices from Fixed-Stride Writes in Key-Value Servers.

triggered when LevelDB merges overlapping SSTables into non-overlapping ones. The overlap results from contiguous indices being placed in different write buffers. Our approach to addressing this issue is to devise a method to allow more indices to be ordered in the write buffer, avoiding the need for frequent flushing and compaction on SSD. At first glance, an intuitive solution is to compress the MemTable after the write buffer is full, and put the compressed MemTable into the mirrored write buffer holding Immutable MemTable (See Fig. 2). In this way, the same mirrored write buffer can fit more MemTables. Once the write buffer is full, these compressed MemTables are decompressed, and the ordered indices in MemTables are merged and flushed to SSDs as non-overlapping SSTables. With an efficient compression algorithm, this MemTable-level compression can significantly reduce the compaction operations on SSD by buffering more indices in the write buffer. However, on further inspection, this solution is not ideal. First, when the reader retrieves an index from the compressed MemTable, the whole MemTable has to be decompressed. The performance penalty for lookup in compressed MemTable is prohibitively high. Second, once the write buffer is full, decompressing and merging the MemTables incurs very high memory consumption, as much as the total size of all the decompressed MemTables.

Our solution to avoid compaction overhead in MetaKV is block-level compression, as shown in Fig. 4. After the write buffer is full, all the ordered indices are clustered into fixed-size blocks (by default 4KB), then compressed and flushed to the compression buffer as compressed blocks. To facilitate fast lookup, these compressed blocks are ordered on a balanced binary search tree (BST) based on their key ranges. For instance, in Fig. 4, BST(A) contains 3 compressed blocks, and each block has a distinct key range that records the minimum key and maximum key of the indices in this block. These blocks are ordered as three tree nodes on BST(A) based on the minimum keys, i.e. 0.5, 1.5, and 5. The compressed blocks from different MemTables are organized separately in the compression buffer. For example, in Fig. 4, the compressed MemTable A and compressed MemTable B are organized separately in the compression buffer as BST(A) and BST(B), respectively. Once the compressed buffer is full, a 2-way merge is triggered on the two BSTs only with the overlapping blocks. In Fig. 4, compressed block A.1 and compressed block B.1 have overlapping key ranges, so they are merged into two non-overlapping blocks with ranges (0.5-1) and (1-1.5), respectively. Along with this merge operation, all the compressed blocks in BST(A) and BST(B) are flushed to SSD based on the order of their minimum keys.

Compared with the intuitive MemTable-level compression described earlier, our block-level compression has two benefits. First, when a get request falls into the compression buffer, instead of decompressing the whole MemTable, only the block holding that index is decompressed, which avoids the heavy performance penalty. Second, merging operations are only performed on overlapping blocks. This significantly reduces memory consumption and decompression overhead.

The bottom of Fig 4 shows how data are organized on SSD. Like LevelDB, each file on SSD is composed of multiple blocks (by default 256). A metadata region with one or multiple metadata blocks is attached to the end of each file to keep track of the key range of each block for binary search. An important consideration for this design is the selection of compression algorithm. Specifically, since MetaKV relies on the compression algorithm to cluster together contiguous indices, the compression algorithm should deliver both high compression ratio and fast compression/decompression speed. MetaKV captures the fact that most of the scientific workloads have their processes concurrently write/read structured data. The indices of these structured data follow regular patterns [13]. When the spatially contiguous indices are distributed to the same key-value server in MetaKV, these indices also follow regular patterns. Fig. 5 gives an example of a typical *fixed-stride pattern* (e.g., the write pattern of P0 and P1 exhibited in Fig. 1 with fixed segment size of A and B), when multiple processes concurrently access a region of a regular multi-dimensional array (e.g., odd columns of a 2-D matrix). In the example of Fig. 5, two processes issue interleaved writes to the shared file with fixed stride. The indices for these writes are distributed to two key-value servers. As we can see from the table in Fig. 5, the elements of each column vary regularly. We can reduce the storage overhead by recording

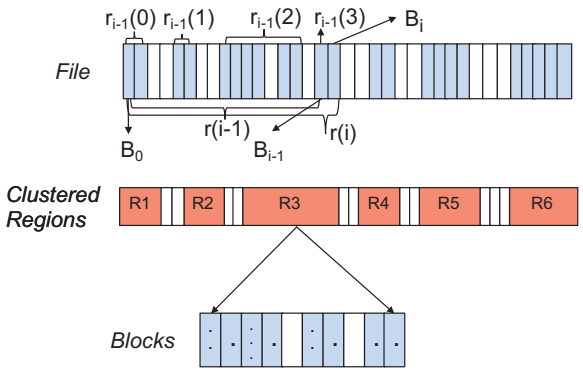


Fig. 6: Supervised Block Clustering

the initial value and the deltas between consecutive values. We use the LZZ [34] algorithm to compress every column of these indices in each block and store each compressed column in a contiguous region of the block. LZ77 has been previously shown to be an efficient compression algorithm that identifies the regularly varying sequence of deltas [13].

B. Supervised Block Clustering

As we describe in Section II-B, LevelDB issues a block read for each requested index, and iterates over all the items until it finds that index. When multiple indices fall on the same block, the performance suffers from a number of redundant seek and read operations. MetaKV addresses this issue using supervised block clustering. It identifies the temporarily hot regions in the on-disk files and clusters together all the blocks in these file regions, so that all the queued read requests in the same region can be serviced in one large read.

A key issue for block clustering is how to dynamically identify the hot regions. We introduce locality factor (LF), defined as Equation 1.

$$LF(first, last) = \frac{\sum_{i=first}^{last} requested[i]}{last - first + 1} \quad (1)$$

In this equation, $first$ and $last$ are the indices of the first and last blocks of a region in the file; $requested[i]$ is set to 1 if the i-th block contains the requested indices. A high LF value means a large percentage of blocks in the file region contain the requested indices (we refer to these blocks as *requested blocks*), and the few intermittent *non-requested blocks* are likely to be requested later. For example, in Fig. 6, ten contiguous blocks are clustered into Region 3 (R3), and the requested indices (points inside the block) fall into eight requested blocks. The two non-requested blocks (blank) in the middle are likely to be requested later. According to the definition of LF, LF of R3 is 80%.

MetaKV decides a region as hot when LF is equal to or larger than a threshold α (default 80%). The purpose of supervised clustering is to cluster the contiguous blocks in the file into a minimum number of hot regions under the constraint $(LF(first, last) \geq \alpha)$, where $first$ and $last$ are the index of the first and last block of each region. In this way, all the read requests can be satisfied in a few large reads on these regions.

We consider two approaches to resolve the aforementioned optimization problem. As a brute force approach, we can recursively search all the combinations of contiguous blocks in the file and calculate their LF. However, this has exponential complexity. Alternatively, we can apply existing clustering algorithms, such as K-means clustering [12], to cluster the blocks into K regions in each round and iterate n rounds for all the possible K values $(1 \leq K \leq n)$, where n is the number of blocks containing the read requests. The complexity for this solution is $O(n^3 t)$, where t is the number of iterations until convergence for each K value.

To avoid the heavy computation associated with these approaches, we solve this problem more efficiently based on dynamic programming. We define B_i as the i-th requested block along the file, where i in B_i only applies to requested block. In this optimization problem, we can derive the clustered regions on the file extent that spans from B_0 to B_i (e.g. r(i) in Fig. 6) based on the clustered regions on the file extent spanning from the B_0 to B_{i-1} (e.g., r(i-1) in Fig. 6), by reversely iterating over the clustered regions of $r(i-1)$ to check if these regions can be combined with their followed regions into a larger region. In Fig. 6, $r(i-1)$ contains four clustered regions: $r_{i-1}(0)$, $r_{i-1}(1)$, $r_{i-1}(2)$ and $r_{i-1}(3)$. These four clustered regions in $r(i-1)$ are searched in the order of $r_{i-1}(3)$, $r_{i-1}(2)$, $r_{i-1}(1)$ to judge if they can be combined into one region once B_i joins as a new element. At the threshold α of 80%, $r_{i-1}(3)$ is firstly combined with B_i as a new region $\{r_{i-1}(3), B(i)\}$, since LF of $\{r_{i-1}(3), B(i)\}$ is 100%. $\{r_{i-1}(3), B(i)\}$ is further combined with $r_{i-1}(2)$ to form a larger region since LF of $\{r_{i-1}(2), r_{i-1}(3), B(i)\}$ is 80%. The operation stops at $r_{i-1}(1)$ since LF of the combined region $\{r_{i-1}(1), r_{i-1}(2), r_{i-1}(3), B(i)\}$ is smaller than α. So the new clustered regions of *r(i)* contains 3 regions: $\{r_{i-1}(2), r_{i-1}(3), B(i)\}$, $r_{i-1}(1)$ and $r_{i-1}(0)$. Following this rule, we can iteratively derive the solution for the whole file.

The detailed algorithm is listed in Algorithm 1. Line 3 initializes the region set (R) with B_0. Lines 4–15 iteratively calculate the region set for the file range that ends at B_i, until the whole file range is covered at B_n. The region set R in each iteration is derived from the resultant R in the previous iteration. In each iteration, the regions in R are iterated reversely starting from $R_{R.count} - 1$ (Lines 5–15). Along the iteration, LF of the region spanning from R_j to B_i is continuously calculated, until LF is below α at some point j (Lines 8–9). The regions between R_{j+1} and B_i are then combined into one region as a new region in R (Lines 10–15). Using dynamic programming, the problem can be solved in $O(mn)$, where n is the number of requested blocks in the file, m is the average number of clustered regions in each iteration. To control the clustering frequency, MetaKV monitors the total request count on the file and classifies the request count into $n/1000$ levels. It performs request clustering once the request count reaches a different level.

When the blocks are clustered into regions, a remaining issue is which region should MetaKV prioritize for service. MetaKV selects the next region for service based on the

Algorithm 1: Supervised Clustering

Input: n: the number of requested blocks in the file ;
B: the list of all the requested blocks ;
Output: R: the list of clustered regions

```
1  begin
2      if n ≥ 1 then
3          R ← B_0
4          for i ← 1...n-1 do
5              for j ← R.count-1...0 do
6                  calculate LF of the region spanning
7                  from R_j to B_i
8                  if LF<α then
9                      break
10             if j+1 == R.count then
11                 R ← R ∪ B_i
12             else
13                 R ← R \ R_{j+1} \ R_{j+2}, ..., \ R_{R.count-1}
14                 C ← R_{j+1} ∪ R_{j+2} ∪, ..., ∪ R_{R.count-1} ∪ B_i
15                 R ← R ∪ C
16     else
17         R ← ∅
```

request density (RD), defined in Equation 2, where $first$ and $last$ are the indices of the first and last blocks in this region, and $count[i]$ is the number of queued requests in i-th block.

$$RD(first, last) = \frac{\sum_{i=first}^{last} count[i]}{last - first + 1} \quad (2)$$

A region with high RD means reading this region is efficient since loading each block in this region can satisfy a large number of read requests. In making scheduling decision, MetaKV selects the regions with the highest RD. Prioritizing regions with high RD allows other less loaded regions to accumulate more read requests for higher read efficiency.

C. Log-Ring Based Collective Message Reduction

As we described in Section II-A, a typical N-1 metadata workload involves all participating clients concurrently sending and receiving file attribute requests to and from the same key-value server, resulting in all-to-one and one-to-all communication patterns, a key factor that limits the metadata scalability.

MetaKV addresses this issue using log-ring based collective message reduction. Here, all servers collectively forward and reduce metadata messages to the same key-value server via two-level reductions: client-server reduction and server-server reduction. Fig. 7(a) uses an example to show our approach. Eight servers are organized on the tree structure to collectively reduce the number of messages sent to Server 0. Each server delegates the attribute requests and acknowledgements (ACKs) for a distinct set of clients. Once a server receives attribute requests from all its clients, it performs a reduction on the

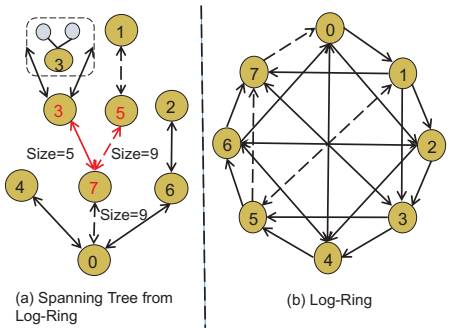


Fig. 7: Log-Ring Based Collective Message Reduction

clients' requests (e.g., Server 3 in Fig. 7 reduces 2 clients), and then forwards the metadata requests to the next hop. Fig. 7(a) depicts a scenario when all the processes concurrently send the file attributes of a shared file after checkpointing and the resulting size attribute of the file is the largest size among all clients' requests. Both Server 3 and Server 5 take Server 7 as the next hop and send the file sizes to Server 7. After receiving the size information, Server 7 only forwards the larger request to the root (Server 0). In this way, Server 0 determines the final file size from the messages of only 3 Servers (4, 7, and 6). Server 0 then broadcasts the ACKs reversely along the tree to all the servers, which will forward the ACK to their clients. In this way, Server 0 only needs to send 3 ACKs and all the broadcast operations are conducted in parallel.

A key consideration is how to build a spanning tree for each server that takes this server as the root. In particular, to ensure scalability, an efficient tree topology should allow any server to forward a message to the target server in $O(log(n))$ hops. Additionally, the fan-in and fan-out of the root node should be $O(log(n))$; otherwise, the root has to send to/receive from $n-1$ servers in an all-to-one/one-to-all communication pattern. Most existing key-value stores use point-to-point messages for communication. With all servers being connected as a complete graph, this spanning tree has $n-1$ fan-in and fan-out. This large fan-in/fan-out allows each server to reach the destination in one hop, which is appropriate for point-to-point communication. However, the large fan-in/fan-out is a scalability issue for all-to-one and one-to-all communication, since the number of messages sent/received by each root on the spanning tree is $n-1$.

Our solution for scalable file attribute handling is based on a log-ring topology [7] shown in Fig. 7(b). The fan-in and fan-out of each node in the ring is $O(log(n))$, meaning each server can directly send to/receive from $O(log(n))$ other servers. For instance, in Fig. 7(b), 0 can directly send to 1, 2, and 4, and receive from 4, 6 and 7. This feature ensures the number of send/receive operations is $O(log(n))$ at scale. In addition, each message can be routed to the destination server in a maximum of $O(log(n))$ hops. For example, in Fig. 7(b), the message from Server 1 is routed to Server 0 along the path $1-5-7-0$. This path is aligned with the path from 1 to 0 on the spanning tree in Fig. 7(a).

With a log-ring based spanning tree, we can calculate the

path from any Server i to Server j along j's spanning tree. The path is derived from simple bit operations using the distance between two servers. The distance is defined as Equation 3.

$$Distance(i,j) = (j - i + size)\%size \quad (3)$$

In Equation 3, *size* is the total number of servers on the ring. Using the example in Fig. 7(b), $Distance(1,0)$ is calculated as 7. With this distance, the path is further derived from the binary sequence of the distance, i.e., $7 = (111)_2 = 2^2 + 2^1 + 2^0$, where 2^2, 2^1, and 2^0 represent the distances between two neighboring hops along the path. In this example, Server 1 can reach Server 0 via path $1 + 2^2 = 5$, $1 + 2^2 + 2^1 = 7$, $1 + 2^2 + 2^1 + 2^0 = 0$, respectively, which is exactly the path along the spanning tree.

IV. Experimental Evaluation

We evaluate MetaKV in several ways. First, we compare the ability of MetaKV to handle concurrent index workloads in comparison with other state-of-the-art key-value stores. Next, we evaluate the performance of MetaKV index management against the baseline performance of MDHIM using HPC I/O benchmark workloads. Finally, we measure MetaKV's ability to handle concurrent file attribute requests using log-ring collective message reduction.

A. Experimental Setup

Here, we briefly describe our experimental setup, including compute system, key-value stores used for comparison, and HPC I/O benchmarks used for performance evaluation.

Compute System: We evaluate MetaKV on Catalyst [2], a 384-node Linux cluster at Lawrence Livermore National Laboratory. Each compute node has two 12-core Intel Xeon E5-2695v2 processors, 128 GB DRAM, 800 GB of PCI-e SSD, and two InfiniBand QDR Qlogic network cards.

Key-Value Stores: We compare MetaKV to several key-values stores as a baseline. First, we use MemcacheDB, a key-value store that is a derivative of Memached [20] that provides reliable cache service based on BerkeleyDB [21]. We also compare against two MDHIM configurations: MD-MySQL and MD-LevelDB, which refer to MDHIM using MySQL and LevelDB as local key-value stores, respectively. MetaKV uses its local storage management for put and get as described in Sections III-A and III-B.

Benchmarks: We evaluate MetaKV's metadata service based on three widely adopted parallel I/O benchmarks: IOR [26], MPI-Tile-IO [25], and BTIO [30]. IOR is a synthetic benchmark that emulates HPC I/O patterns. We focus on N-1 fixed-stride I/O, a typical I/O pattern exposed by scientific applications that utilize one-dimensional partitioning (where each process writes/reads a column of a 2-D matrix). MPI-Tile-IO simulates processes concurrently accessing a two-dimensional dense data set, utilizing 2-D partitioning, where each process writes/reads a tile of the data set. BTIO tests the output capability of the NAS BT (Block Tri-diagonal) parallel benchmark. Its global data set is partitioned using 3-D triagonal partitioning, where each process writes/reads multiple small 3-D regions.

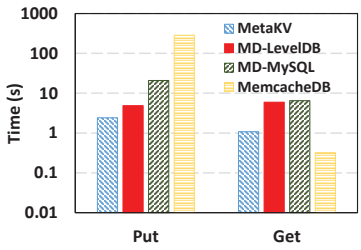


Fig. 8: MetaKV Performance vs. Existing Key-Value Stores.

In all the three workloads, each client process writes its segments of a shared file to its local SSD, and then reads the file back using the same I/O pattern. After each bursty write phase, processes concurrently issue put requests for the file segment indices to MetaKV. Before each bursty read phase, all processes simultaneously get the indices for their requested file segments. Each client records its total put and get time for indices. The longest put and get time among all clients is collected. Each test is run 10 times. The average from the 10 collected results is plotted in the figures.

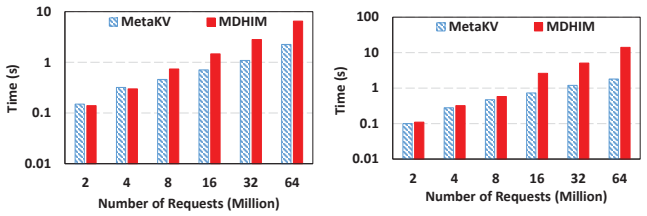
Unless explicitly stated, we run 16 clients on each node to make the total client count a power of 2. This configuration balances the number of clients per server as we scale the log ring. By default, we launch one MetaKV server as a service thread in the first client process on each node.

B. Comparison with Contemporary Key-Value Stores

We first compare MetaKV against the contemporary key-value stores to evaluate their handling of bursty index requests. We launch 16 client processes and one key-value server on a single node, and stress the key-value stores by having all client processes concurrently issue interleaved writes to a shared 1GB file, and then read the shared file using the same pattern. The transfer size is configured as 1KB, which results in 1 million ($1GB/1KB$) index puts/gets on the key-value server.

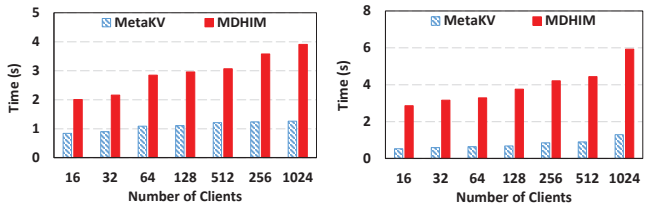
Fig. 8 compares the performance of the key-value stores. We see that both MD-LevelDB and MetaKV perform better than other key-value stores for put. The poor put performance for MySQL results from each put request going through the MySQL daemon, leading to substantial inter-process communication overhead. The worst performance is with MemcacheDB, because MemcacheDB is designed for reliable cache service. It periodically checkpoints the in-memory data for data persistence, and much more data are checkpointed than actually received. In contrast, both MetaKV and MD-LevelDB use write-ahead logging for data persistence and they do not use a daemon for key-value service. Thus, they deliver the best put performance. MemcacheDB performs well for get operations because it retrieves indices directly from DRAM via the cache service.

Overall, MetaKV and MD-LevelDB deliver both faster and more balanced put/get performance compared with the other key-value stores. In addition, it is critical that their data persistence support be implemented without heavy impact on



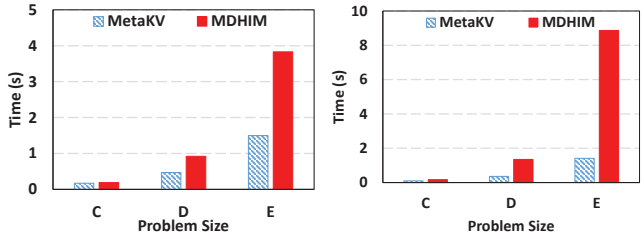
(a) Put Performance (b) Get Performance

Fig. 9: Index Put and Get Overhead with IOR.



(a) Put Performance (b) Get Performance

Fig. 10: Index Put and Get Performance with MPI-Tile-IO.



(a) Put Performance (b) Get Performance

Fig. 11: Index Put and Get Overhead with BTIO.

put performance, since bursty writes are common in HPC applications [11], and index put falls on the critical path of write operations. Due to these benefits, in the rest of this paper, we focus on the comparison between MetaKV and MDLevelDB (referred to as simply MDHIM in the rest).

C. MetaKV Performance with IOR

We evaluate MetaKV's ability in handling N-1 fixed-stride indices of IOR. We stress the servers using index puts and gets of 1024 processes writing/reading a 64GB shared file to/from node-local burst buffers. We vary the write and read request sizes from 32KB to 1KB, which results in 2 to 64 million put/get requests to the key-value servers.

Fig. 9a compares the put performance of MetaKV and MDHIM. The put time of both MetaKV and MDHIM is comparable at small request counts. However, as we increase the number of requests, MetaKV gradually outperforms MDHIM and the gap becomes larger with increasing request count. This is because the increasing number of requests triggers more frequent compaction operations in MDHIM. MetaKV addresses this issue using the technique described in Section III-A: instead of frequently flushing indices to SSD, it orders and compresses indices in memory as fine-grained compressed blocks. In this way, more indices are ordered in memory without going through the external compaction operations. On average, MetaKV delivers 1.83× performance improvement over MDHIM.

We show the get performance of MetaKV and MDHIM in Fig. 9b. The performance of MetaKV exhibits greater benefit with a larger request count. This is because the large number of read requests incurs heavy redundant seek and read operations. Instead, MetaKV clusters the spatially contiguous requested blocks into large regions, and services all the get requests in each region with fewer and larger reads, substantially reducing this redundant read overhead. On average, MetaKV delivers 3.17× performance improvement over MDHIM.

D. MetaKV Performance with MPI-Tile-IO

We evaluate the weak scalability of MetaKV under the index workloads of MPI-Tile-IO. We fix the dimensionality of the tile produced by each process as 4096×32768, the number of tiles along Y axis as 16, and increase the number of the tiles along the X axis from 1 to 64. This results in a range of 16 to 1024 client processes and 1 to 64 servers. Correspondingly, the number of index put and get operations increase from 0.5 million to 32 million.

Fig. 10a compares the put performance of MetaKV and MDHIM. Although the put time of both MetaKV and MDHIM increases with more processes, the put time of MDHIM grows more rapidly. This is because with a larger process count, each MDHIM server receives batched put operations from more processes, resulting in more put operations in LevelDB. These intermittent put operations interfere with the LevelDB's compactions, further delaying the put time. In contrast, MetaKV compresses batched put operations inside the write buffer to avoid frequent compactions on SSD, hence better performance. We observed that get time of MetaKV grows with client count in Fig. 10b. This is because the requests arriving from a larger number of clients incur heavier sending and packing overhead to transfer the requested data back to every client. On average, MetaKV delivers 2.66× and 5.22× performance improvement over MDHIM, respectively for put and get.

E. MetaKV Performance with BTIO

We evaluate MetaKV's ability in handling BTIO indices using problem sizes C, D, and E. We use 64, 144, and 400 clients to write and read data respectively for C, D, and E problem sizes, resulting in 0.2 million, 2.0 million, and 20.8 million put/get operations, respectively.

Fig. 11a shows the put performance. MetaKV achieves 1.1×, 1.9×, and 2.5× better performance than MDHIM for problem sizes C, D, and E, respectively. The larger performance benefit with greater problem size is because MetaKV efficiently reduces the compaction overhead that MDHIM incurs.

In Fig. 11b we show the get performance for BTIO. Because MetaKV eliminates redundant read operations via read clustering, It delivers 1.7× and 3.75×, and 6.29× performance

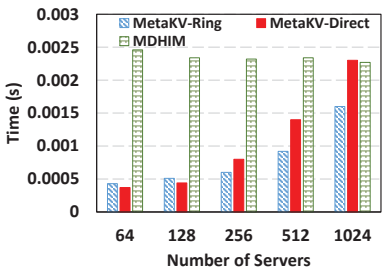


Fig. 12: Performance of Log-Ring Based Collective Messaging

improvement for get operations, respectively for problem sizes C, D, and E.

F. Performance of Log-Ring Based Collective Message Reduction

To assess the benefit of MetaKV in handling the file attribute workload for shared files, we launch 1024 client processes across 64 compute nodes. These clients concurrently send the attributes of a shared file to the same key-value server, which then broadcasts the attributes to all clients after having received the metadata. For MetaKV, we launch 1 to 16 servers on each compute node as the delegators for the clients on the same node. While we are aware that launching 1 delegator can deliver the best performance since it reduces the metadata requests on each node from 16 to 1, we choose to vary the server count to show the benefit of server-side collective optimization (MetaKV-Ring in Fig. 12) over the non-collective solution (MetaKV-Direct in Fig. 12) with different server counts. This helps us assess the scaling performance of log-ring based collective message reduction.

Fig. 12 compares the performance of MetaKV-LogRing, MetaKV-Direct, and MDHIM. In MDHIM, each client directly sends a message to the same server. This server then sends the attribute back to every client. In MetaKV-Direct, after each client sends the attributes to the delegating server collocated with the client on the same node, this server reduces the attributes to one, and forwards it to the target server. The target then broadcasts the attribute to all the delegators. In MetaKV-LogRing, after clients send the attributes to the delegators, the delegators collectively forward and reduce the attribute along the spanning tree until the attributes reach the target server. After that, the target server broadcasts the file attributes to all the clients following a reverse path along the spanning tree. As shown by Fig. 12, both MetaKV-LogRing and MetaKV-Direct outperform MDHIM since client-side messages are reduced by the delegators. On the other hand, MetaKV-LogRing delivers lower latency at a large server count (e.g., from 256 to 1024) compared with MetaKV-Direct. This is because without the collective reduction, the target server has to receive/send attributes from/to all the $n-1$ servers. Instead, with MetaKV-LogRing, all the servers perform reduction along the ring, so the number of send/receive operations are reduced to $O(log(n))$.

V. Related Work

Data indexing is a technique widely used to manage scientific data. For instance, ADIOS [19], NetCDF [15], HDF5 [27], and the future I/O stack of DAOS [17] manage indices of scientific data for fast data query. Their indices define a multi-dimensional region in an array. For N-1 workload, their indices are stored in a metadata section of the shared file. In order to read data, all processes need to load the metadata section into their memory. In contrast, the index format of MetaKV is based on POSIX, which is similar to PLFS index [8]. Each index defines a range of one-dimensional file. Though different in the format, we believe the idea of MetaKV can be ported to accelerate these high-level I/O libraries. DataSpaces [9] provides a distributed indexing service for multi-dimensional data. However, its index service is designed as part of its data service, and the indices are in the distributed DRAM space. Different from DataSpaces, MetaKV is designed for metadata management on node-local burst buffers. It provides a separate metadata service independent of data service.

Several state-of-the-art techniques have been proposed to reduce the overhead associated with managing a large number of indices. Wu *et al.* developed FastBit [31] to compress indices for structured scientific data, and to support fast queries on these compressed indices. It has been used in many existing I/O libraries (e.g., HDF5 and ADIOS). He *et al.* [13] proposed a technique to compress PLFS indices based on pattern detection. Both of these works efficiently exploit the compressibility of indices for structured scientific data. However, under parallel workloads, the indices of scientific data are distributed. One process has to gather the indices from all other processes, and construct the global index for compression. Index gathering can become the bottleneck. MDHIM [10] is designed to manage the PLFS indices and the indices for multi-dimensional data. It distributes the index service over multiple key-value servers. However, its performance is dependent on its local key-value store, which is not optimized for the bursty HPC metadata workloads. MetaKV inherits both the benefits of index compression and distributed key-value service of the aforementioned works. In addition, MetaKV complements these solutions with clustered read service for bursty index workloads.

Many solutions have been proposed to amortize the overhead of the concurrent metadata requests on file systems [23], [24], [29]. Their common strategy is to distribute metadata for different files or directories over distinct metadata servers. However, this strategy does not address the problem of concurrent metadata requests on the same shared file. Our approach differs because MetaKV is optimized to manage the requests for shared files using log-ring based collective message reduction.

VI. Conclusion

In this paper, we have examined various metadata requirements on the distributed burst buffer systems, a critical storage architecture for the next-generation supercomputers. We have also analyzed the challenges in managing these

metadata for scalable I/O service. In particular, when a massive number of processes concurrently operate on a shared file, the performance of a burst buffer system can be handicapped by large indexing overhead and the vast number of all-to-one/one-to-all metadata operations.

To overcome these challenges, we have proposed MetaKV, a distributed key-value store that delivers fast and scalable metadata services for distributed burst buffer systems. While inheriting many good virtues of existing key-value stores, it is designed with compressed storage management, supervised clustering, and log-ring based collective message reduction to vastly reduce the metadata service time on index put/get operations, and optimize the all-to-one/one-to-all metadata operations. Our evaluation demonstrates that, compared with the state-of-the-art key-value stores, MetaKV can improve index put and get operations by as much as 2.66× and 6.29×, respectively. It also delivers 5.7× performance improvement in handling the all-to-one/one-to-all metadata operations.

Acknowledgments

This work was supported in part by a research grant from Lawrence Livermore National Laboratory to Florida State University. This work was also under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-CONF-705913.

REFERENCES

[1] Aurora. http://aurora.alcf.anl.gov/.

[2] Catalyst. http://computation.llnl.gov/computers/catalyst.

[3] LevelDB. https://github.com/google/leveldb.

[4] Summit. https://www.olcf.ornl.gov/summit/.

[5] Theta. https://www.alcf.anl.gov/articles/alcf-selects-projects-theta-early-science-program.

[6] TSUBAME2. http://tsubame.gsic.titech.ac.jp/en/hardware-architecture.

[7] T. Angskun, G. Bosilca, and J. Dongarra. Binomial Graph: A Scalable and Fault-Tolerant Logical Network Topology. In *International Symposium on Parallel and Distributed Processing and Applications*, pages 471–482. Springer, 2007.

[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] C. Docan, M. Parashar, and S. Klasky. Dataspaces: an Interaction and Coordination Framework for Coupled Simulation Workflows. *Cluster Computing*, 15(2):163–181, 2012.

[10] H. N. Greenberg, J. Bent, and G. Grider. MDHIM: A Parallel Key/Value Framework for HPC. In *HotStorage*. USENIX Association, 2015.

[11] R. Gunasekaran, S. Oral, J. Hill, R. Miller, F. Wang, and D. Leverman. Comparative I/O Workload Characterization of Two Leadership Class Storage Clusters. In *Proceedings of the 10th Parallel Data Storage Workshop*, pages 31–36. ACM, 2015.

[12] J. A. Hartigan and M. A. Wong. Algorithm AS 136: A K-Means Clustering Algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

[13] J. He, J. Bent, A. Torres, G. Grider, G. Gibson, C. Maltzahn, and X.-H. Sun. I/O Acceleration with Pattern Detection. In *Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing*, pages 25–36. ACM, 2013.

[14] J. He, A. Jagatheesan, S. Gupta, J. Bennett, and A. Snavely. DASH: a Recipe for a Flash-Based Data Intensive Supercomputer. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

[15] 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.

[16] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, and I. Raicu. ZHT: A Light-Weight Reliable Persistent Dynamic Scalable Zero-Hop Distributed Hash Table. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 775–787. IEEE, 2013.

[17] J. Lofstead, I. Jimenez, C. Maltzahn, Q. Koziol, J. Bent, and E. Barton. DAOS and Friends: a Proposal for an Exascale Storage System. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 50. IEEE Press, 2016.

[18] J. Lofstead, M. Polte, G. Gibson, S. Klasky, K. Schwan, R. Oldfield, M. Wolf, and Q. Liu. Six Degrees of Scientific Data: Reading Patterns for Extreme Scale Science IO. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, pages 49–60. ACM, 2011.

[19] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan. Adaptable, Metadata Rich IO Methods for Portable High Performance IO. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10. IEEE, 2009.

[20] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling Memcache at Facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, 2013.

[21] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, pages 43–43, Berkeley, CA, USA, 1999. USENIX Association.

[22] P. ONeil, E. Cheng, D. Gawlick, and E. ONeil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33(4):351–385, 1996.

[23] S. V. Patil, G. A. Gibson, S. Lang, and M. Polte. Giga+: Scalable Directories for Shared File Systems. In *Proceedings of the 2nd International Workshop on Petascale Data Storage: held in conjunction with Supercomputing'07*, pages 26–29. ACM, 2007.

[24] K. Ren, Q. Zheng, S. Patil, and G. Gibson. IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 237–248. IEEE, 2014.

[25] R. B. Ross. Parallel I/O Benchmark Consortium.

[26] H. Shan and J. Shalf. Using IOR to Analyze the I/O Performance for HPC Platforms. *Lawrence Berkeley National Laboratory*, 2007.

[27] The HDF Group. Hierarchical data format version 5, 2000-2010.

[28] T. Wang, K. Mohror, A. Moody, K. Sato, and W. Yu. An Ephemeral Burst Buffer File System for Scientific Applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 21. ACM, 2009.

[29] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller. Dynamic Metadata Management for Petabyte-Scale File Systems. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, page 4. IEEE Computer Society, 2004.

[30] 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.

[31] K. Wu, S. Ahern, E. W. Bethel, J. Chen, H. Childs, E. Cormier-Michel, C. Geddes, J. Gu, H. Hagen, B. Hamann, et al. FastBit: Interactively Searching Massive Data. In *Journal of Physics: Conference Series*, volume 180, page 012053. IOP Publishing, 2009.

[32] W. Yu, J. S. Vetter, and H. S. Oral. Performance Characterization and Optimization of Parallel I/O on the Cray XT. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–11. IEEE, 2008.

[33] X. Zhang, K. Davis, and S. Jiang. IOrchestrator: Improving the Performance of Multi-Node I/O Systems via Inter-Server Coordination. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

[34] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.