Optimizing Write Performance for Checkpointing to Parallel File Systems Using LSM-Trees

Serdar Bulut
serdar.bulut@york.ac.uk
University of York
York, UK

Steven A. Wright
steven.wright@york.ac.uk
University of York
York, UK

ABSTRACT

The widening gap between compute performance and I/O performance on modern HPC systems means that writing checkpoints to a parallel file system for fault tolerance is fast becoming a bottleneck to high-performance. It is therefore vital that software is engineered such that it can achieve the highest proportion of available performance on the underlying hardware; and this is a burden often carried by I/O middleware libraries. In this paper, we outline such an I/O library based on a Log-structured Merge Tree (LSM-Tree), not just for metadata, but also scientific data. We benchmark its performance using the IOR benchmark, demonstrating 2.4 to 76.7× better performance than alternative file formats, such as ADIOS2, HDF5, and IOR baseline when running on a Lustre Parallel File System. We further demonstrate that when our LSM-Tree I/O library is used as a storage layer for ADIOS2, the resulting I/O library still outperforms the default ADIOS2 implementation by 1.5×.

CCS CONCEPTS

• **Information systems → Distributed storage**.

KEYWORDS

high performance computing; input/output; checkpointing; MPI; distributed storage

ACM Reference Format:
Serdar Bulut and Steven A. Wright. 2023. Optimizing Write Performance for Checkpointing to Parallel File Systems Using LSM-Trees. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3624062.3624118

1 INTRODUCTION

Between 2008 and 2022, the compute performance of the #1 HPC system grew from a PetaFLOP/s (IBM Roadrunner[1]) to an ExaFLOP/s (Frontier[2]). During the same time period, the headline I/O bandwidth to the parallel file system grew from 216 GB/s on Roadrunner[3] in 2008 to 10 TB/s for the SSD tier and 5.5 TB/s for the HDD tier in Frontier[4] in 2022. This translates to approximately 46.3× and 25.5× growth respectively for I/O bandwidth, two orders of magnitude less than the compute growth of 1074.1× during the same time.

Figure 1 shows how the headline compute performance and parallel file system performance has grown on the #1 system between 2008 and 2023, as we moved from Petascale to Exascale. While compute performance has continued to approximate Moore's law, doubling approximately every 18 months, I/O performance has only doubled approximately every 3 years. Many applications previously considered to be compute- or memory-bound, are fast becoming I/O-bound.

Approximately 75-80% of HPC I/O is checkpoint data, which are bursty, immutable, and write-once-read-rarely [3, 40]. Real world examples include checkpoint sizes on the Titan supercomputer ranging from 0.83 GB for VULCAN, up to 160 TB for CHIMERA [43].

Many pre- and post-Exascale systems[4] add NVMe SSDs as part of their storage tiering. Due to SSDs constraints such as cost and more importantly write endurance, HDDs are still foundational building blocks. For example, Frontier[5] has 11.5 PB of NVMe SSD Storage vs. 700 PB HDD Storage.

In this paper we introduce LSMIO, an I/O library based on a log-structured merge tree (LSM-Tree). Our hypothesis is that using

[1]https://www.top500.org/lists/top500/2008/06/
[2]https://www.top500.org/lists/top500/2022/06/
[3]https://www.krellinst.org/doecsgf/conf/2009/pres/barney.pdf

[4]https://www.hpe.com/psnow/doc/a00062172enw
[5]https://www.nextplatform.com/2021/05/21/first-look-at-frontier-supercomputers-storage-infrastructure

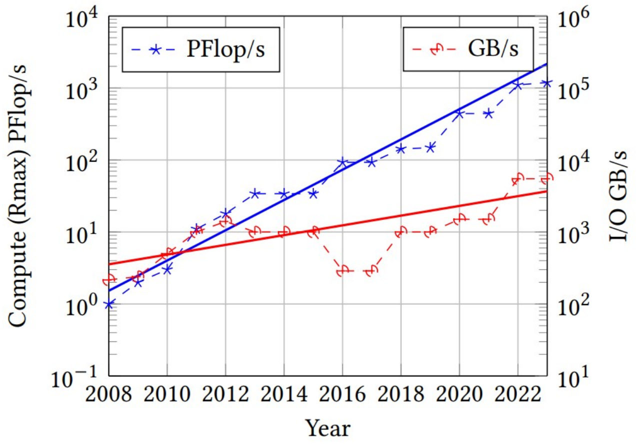

Figure 1: CPU and I/O performance growth between the start of the PetaFLOP era and the ExaFLOP era

an LSM-Tree for the scientific data write-path, not just for metadata, provides better performance – measured by utilisable write bandwidth – as the number of compute nodes increases and as available I/O capacity per node decreases. This paper then compares our library to different I/O libraries targeted at the writing of checkpoint data from compute nodes to the storage layer, such as the Lustre file system. Specifically, this paper makes the following contributions:

- We present LSMIO, an I/O library that leverages an LSM-Tree-based backing store implementation with 3 different interfaces: a C++ IOStream-like API (FStream API), a K/V interface that the library itself also uses internally, and an ADIOS2 API through ADIOS2's plugin mechanism;
- We demonstrate that our LSMIO library performs significantly better than the baseline IOR-benchmark once the number of compute nodes passes the Lustre stripe count;
- We then demonstrate that our LSMIO library's write performance is significantly better than ADIOS2 and, furthermore, performs an order of magnitude better than HDF5;
- Finally, we evaluate the read performance of our library, showing that the read performance exceeds IOR baseline and HDF5, and is on average within 23.3% of ADIOS2's read performance.

The remainder of this paper is structured as follows: Section 2 outlines the background and related work for this study; Section 3 provides an overview of our proposed I/O library; Section 4 shows the evaluation of our library against the IOR baseline as well as alternatives, such as HDF5 and ADIOS2; finally, Section 5 concludes this paper.

2 BACKGROUND AND RELATED WORK

The end of Dennard scaling around 2008 has meant that as clock rates have ceased to increase, performance increases instead require more cores and denser chips [6]. More components and higher density results in more failures. Mean Time Between Failures (MTBF) has reduced from days to minutes, where the estimated MTBF is 17 minutes for a system with 100,000 nodes [36]. This translates into a 20× increase in failure rate when an application moves from 10K to 22K CPU cores [5]. With more failures, more proactive and remedial actions are needed [29]. One such fault tolerance technique is checkpointing – enabling recovery with rollback [13].

Checkpointing comes with an associated overhead and this overhead is linearly proportional to the checkpointing size and I/O latency, and inversely proportional to the I/O bandwidth [37]. If the checkpointing time is close to the MTBF then an HPC system spends most of its time doing checkpoint and restart, effectively making little or no progress [6].

2.1 Checkpoint I/O Improvements

To reduce the checkpointing overhead numerous mechanisms have been proposed. One of the early mechanisms is reducing the blocking time for checkpointing by making it asynchronous, or using a faster tier in-between to reduce the time for checkpointing [13]. Other earlier mechanisms include reducing the checkpointing data size by using application-level checkpointing instead of system-/user-level checkpointing, performing incremental checkpointing, compression, memory exclusion, decimation, interpolation or timing such that the checkpoint size will be the smallest during the life of the application [13, 23, 47]. There are several novel methods introduced for incremental checkpointing [1, 20, 31, 32]

Buffering checkpoint data into RAM and then to the local disk was introduced by Plank et al. as early as in 1993 [30]. Making this write from RAM to disk asynchronously was introduced later by Li et al. in 1994 [20]. To improve the reliability of using local RAM as a checkpoint location, numerous methods have been introduced such as utilizing neighbor RAMs by mirroring or for parity (one dimensional, XOR or RAID-5) and DSM (Distributed Shared Memory) with distributed log updates [10, 11, 38, 39]. Novel approaches have been introduced for writing checkpoint data to local disks of HPC nodes such as using striping, staggering, as well as distributed RAID [7, 16, 30, 42].

Checkpoint data can also be stored at partner/neighbor disks using FTC-Charm++ [48]. Additionally, these neighbor nodes can be used as data transfer nodes as well as next in the data life cycle, such as compressing the data before the data is sent to the next layer in a tiered storage hierarchy [2, 35]. Alternatively, the local SSDs could be considered as burst buffers, or neighbor nodes could be leveraged as intermediary nodes [2, 46]. However, in the case of SSDs there needs to be additional optimizations to improve SSD endurance [43].

Multi-level checkpointing could traverse full storage tiering all the way from local RAM to local disk to neighbor nodes (RAM, CPU, Network, and Disk) and to a central Parallel File System (PFS). The Scalable Checkpoint/Restart (SCR) library from the Lawrence Livermore National Laboratory is one such system and its successor CRUISE/SCR adds memory spill support [27, 33].

Aggregation of checkpoint files using collective I/O dates back to 1999 by ROMIO [41]. Charm++ implements similar capability as well [26]. LACIO aggregates and later re-orders/swaps blocks to align with the PFS [8, 9]. T3PIO introduces HPC auto-tuning capability of the Lustre stripe count and the number of collective I/O writers to the PFS [24].

The file layout on a PFS also impacts the performance due to how both metadata and scientific data are laid out, and what their update patterns are on the PFS. Hence, if the number of nodes to number of checkpoint files ratio is greater than 1 the data performance improves at the expense of additional metadata operations [3, 45]. PLFS solves this problem by representing itself as an N-to-1 file but implementing an N-to-N mapping on the actual PFS while managing the metadata implosion by slab allocation [4].

There are multiple popular file formats which primarily try to solve ease of use and portability challenges of input and output formats, which also can be used for checkpointing such as HDF5 [17] and NetCDF [19], which also support collective I/O [14]. PLFS also integrates with and can be a backend layer for HDF5 [25].

ADIOS has incorporated a lot of the ideas mentioned previously, including N-to-1 to N-to-N file mapping, in-memory data aggregation, and buffering with the option of per group of processes within a host not just for all nodes [21, 22]. ADIOS2 significantly outperforms other file formats such as PNetCDF by more than 10× in I/O throughput as parallelism increases [18]. ADIOS2 has demonstrated over 30× and 10× I/O performance improvements at approximately

30 GB/s over their default implementations for S3D (96K cores) and PMCL3D (30K cores) [21].

2.2 LSM-Tree

Our proposed approach is to optimize the checkpoint writes such that they appear as sequential access to the disk which then enables maximum throughput on HDDs. An LSM-Tree is such an in-memory and on-disk data structure that we believe could be beneficial for checkpoint writes.

At its essence the log-structured merge-tree is a disk-based data structure that buffers and aggregates updates and then writes multi-page blocks to new locations on a disk optimized for sequential disk access, which later can be read in a way similar to a merge sort [28]. This was inspired by the log-structured file system, created by Rosenblum and Osterbout, hence the origin of the name log-structured merge-tree [34].

This aggregation and buffering enables large sequential writes because the data is packaged together to be written in contiguous multi-page disk blocks at a very high throughput on HDDs, as HDD performance is typically measured in sequential write throughput.

To make writes more durable and avoid a data loss due to this aggregation and buffering in memory, a write-ahead log file (a sequential log file containing a full history of the updates) is used until the data is flushed from memory to disk.

The in-memory data structure represents both the last N updates of the data before they are written to disk, as well as the metadata of the on-disk files in the LSM-Tree. The former is referred as the C_0 tree. When the data becomes large enough, the sorted data migrates to the tree on disk, which is referred as the C_1 tree. The leaf nodes in C_1 are never edited in-place but instead new ones are added as part of an asynchronous rolling-merge process where the old ones are deleted afterwards. To help contain the number of leaf nodes of the tree on disk as it gets larger, there can be more than one level of on-disk representation (i.e., C_1, C_2, ..., C_K) where the size of the on-disk files increase at each successive level. These C_0 and $C_{1..K}$ trees are referred as the MemTable and the Sorted-String-Table (SSTable) files in the popular implementations of this concept such as LevelDB and its fork RocksDB [12, 15]. We can, for example, create one such data structure for each rank in an MPI application. Figure 2 illustrates this data flow in details.

While this is likely to improve write performance, reads of sufficiently old entries are likely to be slower due to the merge-sort of several files in the C_1 tree from disk. Here the sufficiency is where the data is already migrated from C_0 to C_1. This makes the LSM-Tree implementation ideal for low read-to-write ratio data store use cases as well as when reads in the use case have a bias towards recent writes. Both are primary characteristics of checkpoint data.

3 LSMIO

3.1 LSMIO Library

To test our LSM-Tree hypothesis we have created an I/O library called LSMIO, which provides three different interfaces: a K/V interface, a C++ IOStream-like API (FStream API), and finally an ADIOS2 Plugin. Figure 3 shows the architecture of our LSMIO library.

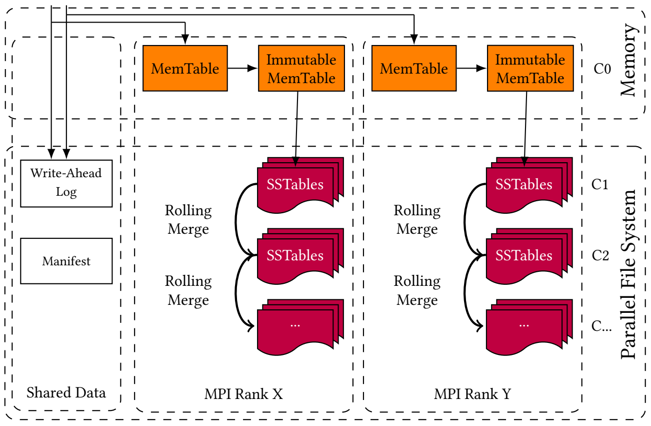

Figure 2: LSM-Tree Architecture

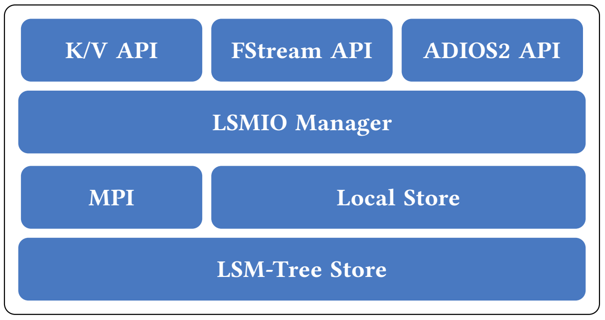

Figure 3: LSMIO Architecture

3.1.1 LSM-Tree Store: To evaluate our hypothesis, we base our LSMIO library on an existing implementation of an LSM-Tree store. We evaluated two of the most popular LSM-Tree implementations: LevelDB and RocksDB. Both implement full lifecycles of reads and writes, including a batch interface, customizations including buffer size, and threading options. However, our implementation uses RocksDB since it provides more customization options such as disabling the write-ahead log, which LevelDB does not.

To customize RocksDB to the specification required as a building block, we made the following changes:

- Disabled write-ahead log
- Disabled compression
- Disabled caching
- Disabled compaction
- Exposed an option to write either synchronously or asynchronously
- Exposed an option to use MMAP
- Exposed options to customize buffer size and inherit the value from ADIOS2 configuration when used as a plugin
- Exposed an option to change block size

We do not believe we need the write-ahead log capability for the checkpoint data use case because a write-barrier could be explicitly called by the user; ADIOS2 already provides such an API call.

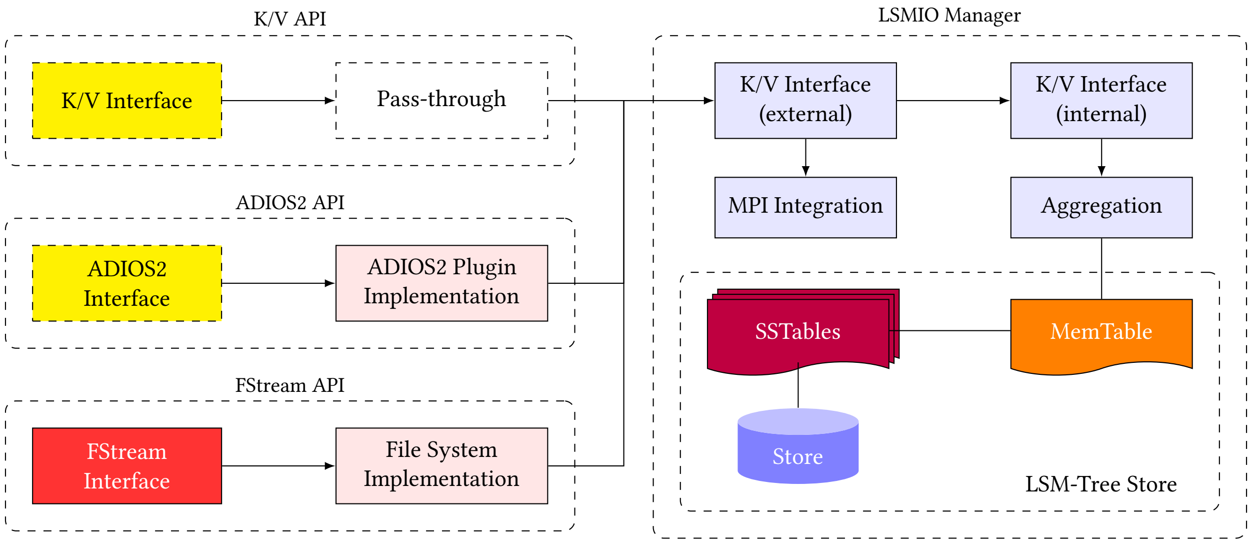

Figure 4: LSMIO Internals

Alternatively, LSMIO calls the write-barrier implicitly at the end of the checkpoint file write.

Fundamentally, the exact functionality needed to validate the hypothesis at a high-level are:

- Ability to manage the MemTable in RAM
- Ability to write/read immutable SSTables on disk

3.1.2 Local Store: This is the layer that encapsulates the LSM-Tree implementation using an internal K/V interface (i.e. get, put, delete) which additionally defines the interface for some of the primary building methods:

- All of the customizations mentioned for the underlying LSM-Tree earlier in this section are implemented here.
- When using LevelDB, buffering and aggregation is implemented using the WriteBatch interface since LevelDB does not allow disabling the write-ahead log file to stop triggering a disk write for writes. In the RocksDB implementation, the WriteBatch interface is not required.
- A write barrier is implemented by flushing the aggregation buffer as well as triggering a flush in the underlying LSM-Tree implementation.
- A single thread is configured for flushing writes.

Table 1 highlights some of the key functions in the LSMIO Store classes with LevelDB and RocksDB backends.

3.1.3 MPI:. We use MPI to implement an MPI barrier for our benchmarks. In the future, we will be able to implement a collective I/O capability for the HPC environment in this layer, where, for example, a single LSM-Tree store could be created for all or a group of nodes participating in checkpointing.

3.1.4 LSMIO Manager: The LSMIO manager manages the local store as well as the MPI integration. It also provides the functionality for the external K/V interface with needs such as an append function, enabling MPI options, multiple put methods for different data types, performance counters, and an optional factory method to manage the object instance for the caller. Table 2 lists some of the key functions in the LSMIO manager class.

Table 1: Summary of the key functions in the Local Store.

Method	Description
`startBatch()`	Start batching if the underlying concrete implementation (e.g. LevelDB) needs it and uses a protected data structure to manage batching.
`stopBatch()`	Stop batching and flushes the writes if the underlying concrete implementation needs it and uses a protected data structure to manage batching.
`get(...)`	Get a value from the local database using a key. Always executed synchronously.
`put(...)`	Write a value locally for the key passed. If the key exists, overwrite it. Has the option to execute asynchronously.
`append(...)`	Append the value locally to the existing value of the key passed. Can be synchronous or asynchronous.
`del(...)`	Delete the value locally given by the key. Can be synchronous or asynchronous.
`writeBarrier(...)`	Flush all buffered writes to disk. Block until flushing to disk is done.

3.1.5 K/V API:. This is the external K/V interface that applications can use to integrate with the LSMIO library. In our current implementation, this is the LSMIO manager itself as we have chosen to avoid another layer of abstraction at the expense of cleanliness. This API could be extended in the future to introduce additional

Table 2: Summary of the key functions in the LSMIO Manager.

Method	Description
`get(...)`	Get the value for the key passed. Always synchronously.
`put(...)`	Write the value locally or remotely (in the case of collective I/O) for the key passed. Can be synchronous or asynchronous.
`append(...)`	Append the value locally or remotely (collective I/O) to the existing value of the key passed. Can be synchronous or asynchronous.
`del(...)`	Delete the value locally or remotely (collective I/O) given by the key. Can be synchronous or asynchronous.
`writeBarrier(...)`	Flush all buffered writes to disk locally or remotely (collective I/O).

functionality, such as using batch reading to improve the read performance.

3.1.6 FStream API:. To do basic validations, we implement a C++ file stream API similar to C++'s IOStream libraries, including a factory method. In essence this becomes a user-space POSIX implementation that requires the developer to develop and link it to their application at compile time. In future, this layer could be linked at runtime using `LD_PRELOAD` (cf. [44]). Some of the key functions exposed by our IOStream-like API are shown in Table 3.

Table 3: Summary of the key functions in the FStream API.

Method	Description
`...`	Example implementations of the methods: `open`, `read`, `write`, `seekp`, `tellp`, `rdbuf`, `fail`, `good`, `flush`, and `close` similar to the C++ stream API, but using the LSMIO store.
`initialize(...)`	A static method to initialize an LSMIO store for FStream API to write to and read files from.
`cleanup(...)`	A static method to close the LSMIO store that FStream API was using for file read and write.
`writeBarrier(...)`	A static method to flush all the pending writes to disk. Blocks until flushing is done.

3.1.7 ADIOS2 API:. ADIOS2 provides an extensibility mechanism called "Plugin" where a developer can implement a custom backend for ADIOS2 without the ADIOS2 users needing to make any changes to their applications. Our ADIOS2 plugin enables applications that use ADIOS2 to use our library by simply updating their XML configuration file, which is read at the start of their application. Our plugin is implemented using LSMIO's external K/V interface. The ADIOS2 API provides a richer API for users, including additional data types. When implementing multi-dimensional writes as an ADIOS2 plugin we use a simple serialization into a string to be stored in the lower layers of our stack.

Figure 4 provides an in-depth view of the library's internal functionality.

4 EVALUATION

In this section we first conduct a baseline performance comparison between standard I/O and our LSMIO library using IOR, an I/O benchmark commonly used in parallel file system performance studies[6]. We then compare our LSMIO library against similar file I/O libraries such as HDF5 and ADIOS2. Additionally, we compare the performance of our LSMIO library through the ADIOS2 plugin interface against ADIOS2's BP5 file format. Finally, we compare the read performance of our library against the IOR baseline, HDF5, and ADIOS2.

We run our experiments on the University of York's Viking cluster. Relevant specification details can be found in Table 4. We run each test 10 times, with the Lustre stripe size $\in \{64\text{KB}, 1\text{MB}\}$, and stripe count $\in \{4, 16\}$. We then choose the maximum I/O bandwidth values for all cases in our evaluation and hence, in the plots. Different stripe sizes and counts show similar results and thus we include a representative subset of the results in this section for brevity.

4.1 Write Benchmarks: Baseline IOR vs. LSMIO

We start our analysis by establishing a baseline of performance on Viking's Lustre File system using the standard IOR benckmark. We run our benchmarks on the cluster using concurrency up to 48 nodes. As seen in Figure 5, the write performance scales linearly for IOR as long as the number of nodes in the simulation are less than the Lustre stripe count. After that threshold, the write performance dramatically drops by as much as 6.2×. Evaluations with different block-sizes exhibits similar behavior though there is an improvement from 64K to 1M by as much as 4.9× as the concurrency increases. We then perform the same simulation including the same block-sizes and the same concurrency using our LSMIO

[6]https://wiki.lustre.org/IOR

Table 4: Technical specification for the University of York's Viking system

Viking	
Processor	Intel Xeon 6138
CPU speed	2.0 GHz
Cores per node	40
Nodes	137
Memory	192 GB
File System	Lustre
Lustre OSTs	45
OST h/w	10 × 8TB 7,200 RPM NLSAS
Lustre OSSs	2

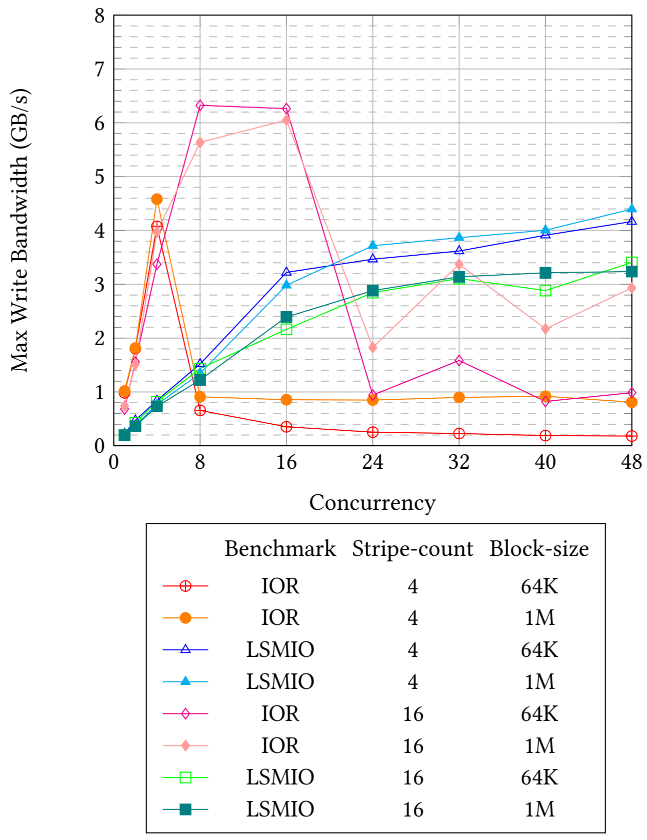

Figure 5: Comparison of IOR baseline to LSMIO, with Lustre stripe count 4 and sizes 64K and 1M

library. Even though our library does not perform as well as IOR at low levels of concurrency, it continues to scale and surpasses IOR baseline by as much as 23.1× when the concurrency (or the number of nodes) peaks at 48. This gives us sufficient confidence in our approach to begin comparing it to commonly used I/O libraries such as HDF5 and ADIOS2.

4.2 Write Benchmarks: LSMIO vs. HDF5 and ADIOS2

IOR provides a built-in mechanism to benchmark an HPC file system using an HDF5 formatted file. As seen in Figure 6, writing an HDF5 file is significantly slower than the IOR baseline by any where between 2.6× and 48.1×. Different block-sizes show similar behavior, though we observe up to 9.9× difference within HDF5 performance going from 64K to 1M after concurrency surpasses the Lustre stripe count.

The ADIOS2 library is increasingly being used by new HPC applications. To compare the performance of our library to ADIOS2, we configure ADIOS2 to use approximately the same parameters as LSMIO including the same in-memory buffer size of 32 MB. While the IOR baseline starts strongly as long as the concurrency is less than the Lustre stripe count, writing an ADIOS2 file continues to

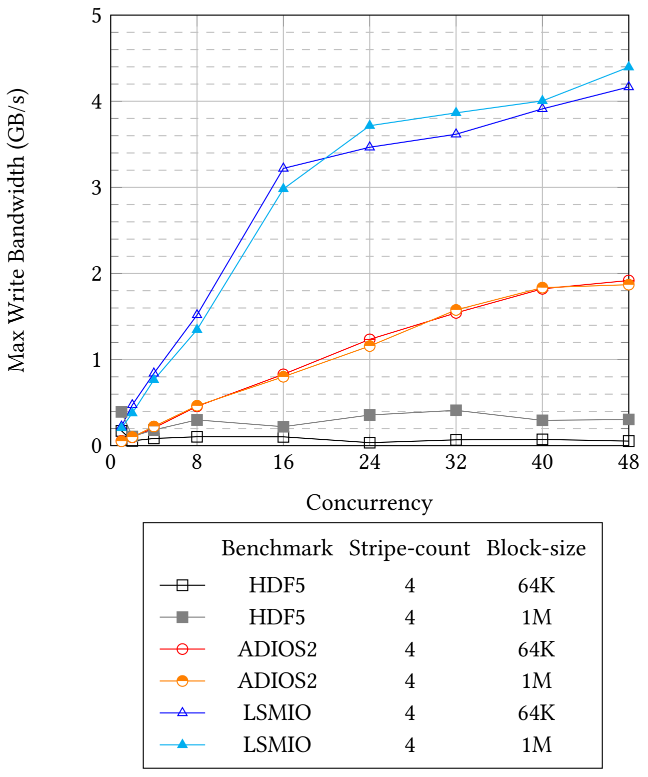

Figure 6: Comparison of HDF5 and ADIOS2 to LSMIO, with Lustre stripe count 4 and sizes 64K and 1M

scale and easily surpasses the IOR baseline by as much as 10.7× when the concurrency reaches 48 nodes.

When we compare both to LSMIO, there is a dramatic performance difference between LSMIO and ADIOS2 as well as between ADIOS2 and HDF5. Writing an ADIOS2 file easily outperforms HDF5 by as much as 35.3× when the concurrency reaches 48 nodes. Similarly, the LSMIO implementation continues to scale and surpasses HDF5 by more than 76.7× and ADIOS2 by more than 2.4× when the concurrency (or number of nodes) reaches 48.

4.3 Write Benchmarks: LSMIO Plugin vs. ADIOS2

ADIOS2 provides a plugin interface to augment its storage layer. The advantage of this is that an application built using ADIOS2 can use a plugin with no code changes, i.e., with just an XML configuration change, which is read by ADIOS2 during its start up sequence. With that we can repeat our experiments using our LSMIO-based plugin for ADIOS2, allowing users to make use of our library without any code changes.

We ensure that LSMIO and ADIOS2 use the same configuration parameters, such as the in-memory buffer in our benchmark experiments. As with our previous experiments, we repeat our runs with different block-sizes, but we note that the difference in performance is negligible. All LSMIO implementations, as well as

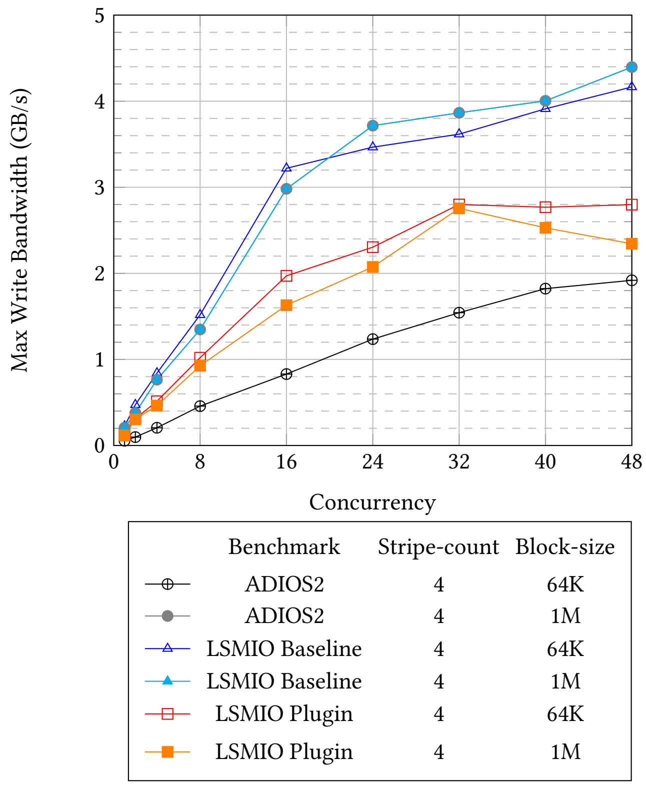

Figure 7: Comparison of ADIOS2 to LSMIO baseline and LSMIO Plugin, with Lustre stripe count 4, sizes 64K and 1M

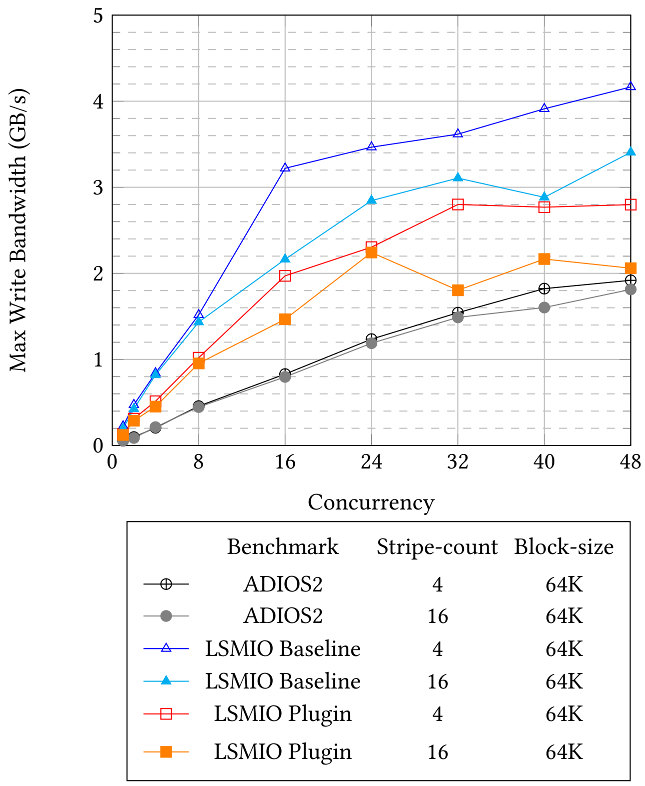

Figure 8: Comparison of ADIOS2 to LSMIO baseline and LSMIO Plugin, with Lustre stripe counts 4 and 16, size 64K

ADIOS2, continue to scale as the number of nodes in the simulation increases.

As seen in Figures 7 and 8, the performance of the LSMIO plugin for ADIOS2 lands approximately in the middle between ADIOS2 and our base LSMIO implementation. The performance gain from ADIOS2 to our LSMIO plugin is as high as 1.5× and from our LSMIO plugin to the LSMIO baseline is approximately 1.5× as the concurrency (or number of nodes) reaches 48. Namely, the performance gain by using the LSMIO directly is higher than using it through the ADIOS2 plugin due to the constraints that ADIOS2 brings in. We believe that this performance differential is caused by: (i) additional layers of abstraction introduced by ADIOS2, (ii) the strong typing in ADIOS2 compared to the byte-array representation used by LSMIO, and (iii) an inefficiency in memory management inside our plugin implementation compared to using LSMIO directly.

4.4 Write Benchmarks: Collective I/O

Another important part of the analysis is comparing the benchmark results when collective I/O is enabled. IOR has built-in capabilities to benchmark using collective I/O for its baseline tests as well as HDF5 tests. At the present time we do not have a collective I/O implementation of our LSMIO library. However, we are able to compare LSMIO baseline to IOR and HDF5 with collective I/O in Figure 9.

Collective I/O does provide significant performance improvement to IOR's default implementation, improving the baseline performance by as much as 12.1×. However, the collective I/O improvement for HDF5 is not particularly significant. When the concurrency is lower, collective I/O improves HDF5 performance by 2× on average; as the concurrency increases collective I/O reduces HDF5's performance by as much as 2.5×.

LSMIO baseline continues to outperform IOR's default implementation with or without collective I/O. In the latter case, LSMIO outperforms IOR with collective I/O by as much as 2.2× as the concurrency peaks.

4.5 Read Benchmarks

Our library focuses primarily on improving the performance of write operations. In doing so, the synchronous and point-lookup nature of the read requests negatively impacts the read performance of the LSM-Tree implementation. Hence, as seen in Figure 10, ADIOS2 achieves the highest read performance, and its performance scales well with increases in concurrency.

In the case of IOR, read performance also scales as concurrency increases. However, IOR's read performance is significantly worse than ADIOS2, and drops by as much as 18.6× when collective I/O is enabled.

LSMIO significantly outperforms the IOR baseline by about 5.5× when the concurrency peaks.

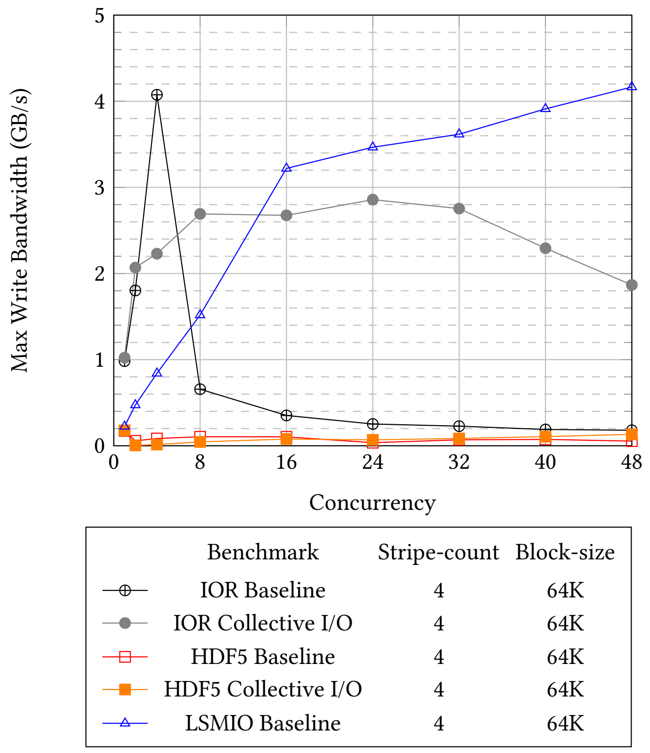

Figure 9: Comparison of IOR baseline and HDF5 using Collective I/O to LSMIO, with Lustre stripe count 4 and size 64K

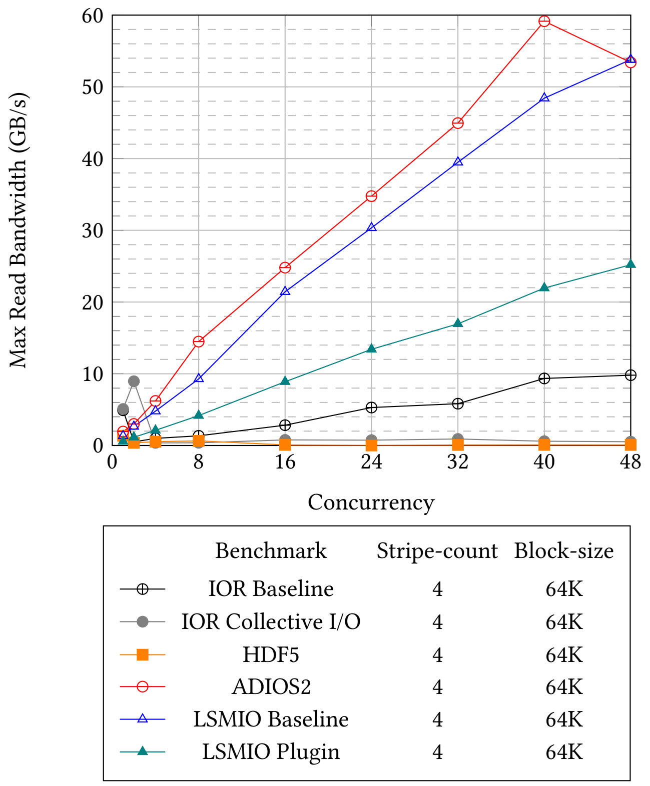

Figure 10: Comparison of IOR baseline with or without Collective I/O, HDF5, and ADIOS2 to LSMIO baseline and LSMIO plugin, with Lustre stripe count 4 and size 64K

When comparing different file formats, HDF5 continues to struggle against the other formats in performance on the read side as well. IOR baseline outperforms HDF5 by as much as 125.2× and LSMIO baseline outperforms HDF5 by as much as 687.2× as the concurrency peaks.

Maintaining the same pattern from the write benchmarks, the performance gain by using the LSMIO directly is higher than using it through the ADIOS2 plugin due to the constraints that ADIOS2 brings in.

5 CONCLUSION

As we enter the Exascale era of HPC, compute has outgrown I/O by orders of magnitude. Given that as much as 75-80% of HPC I/O is checkpoint related, improving the performance of checkpoint I/O is of utmost importance to ensure it does not become a bottleneck.

In this paper we have introduced an I/O library, called LSMIO, which is based on an LSM-Tree for the scientific data write-path. Our implementation is backed by the RocksDB LSM-Tree implementation, which is highly customizable and enables us to tailor it to our needs.

Our evaluation of LSMIO has demonstrated that we can improve performance – measured by utilisable write bandwidth – by as much as 23.1× from the baseline, as the number of compute nodes increases while available I/O capacity per node is decreasing. We subsequently compared our library to different I/O libraries targeted at the writing of checkpoint data from compute nodes to the storage layer, such as the Lustre file system.

We have demonstrated that our library's write performance is better than the leading HPC I/O library, ADIOS2, by around 2.4× and HDF5 by as much as 76.7× on our evaluation system. We have additionally developed an LSMIO plugin backend for ADIOS2 that allows users of ADIOS2 to start using LSMIO without any code changes – with a small performance penalty when compared to LSMIO alone, but still up to 1.5× the write performance of the default ADIOS2 implementation.

When we incorporate collective I/O into the IOR baseline and HDF5 benchmarks, it improves their performances by as much as 12.1× and 2× respectively. However, LSMIO continues to outperform IOR with collective I/O by as high as 2.2× as the concurrency increases.

Finally, we evaluated the read performance of our library and compared it to the other file formats: ADIOS2 and HDF5. Although the performance of LSMIO is on average 23.3% below that of ADIOS2, our library still performs significantly better than IOR with and without HDF5.

5.1 Future Work

This paper outlines our initial implementation and investigation into LSM-Trees for the scientific data write path. Currently this has only been evaluated on a single HPC cluster using the Lustre file system; wider evaluations are required on alternative HPC file systems, and larger Lustre installations. Testing on differently constructed and configured file systems might highlight additional opportunities to optimize the I/O performance of the write-path. In addition, we are considering adding a collective I/O capability using MPI to our library. We anticipate this boosting its performance further.

The work presented in this paper has been collected using the IOR benchmark only, likely representing a "best case" for I/O operations. The I/O operations in more representative scientific applications are likely to elicit different performance characteristics, and so there are ongoing efforts to embed our library in real applications.

In addition, to improve the read-performance in real applications, we will explore sequential or batch read of the variables from the LSM-Tree into memory instead of random reading of each key for the data. To improve the write-performance of the ADIOS2 plugin implementation for our LSMIO library, we will improve its memory management such that it behaves similarly to when the LSMIO library is called directly.

ACKNOWLEDGMENTS

This project was undertaken on the Viking Cluster, which is a high performance compute facility provided by the University of York. We are grateful for computational support from the University of York High Performance Computing service, Viking and the Research Computing team.

REFERENCES

[1] Saurabh Agarwal, Rahul Garg, Meeta S. Gupta, and Jose E. Moreira. 2004. Adaptive Incremental Checkpointing for Massively Parallel Systems. In *Proceedings of the 18th Annual International Conference on Supercomputing*. 277–286.

[2] Abhinav Agrawal, Gabriel H. Loh, and James Tuck. 2017. Leveraging Near Data Processing for High-Performance Checkpoint/Restart. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[3] Dulcardo Arteaga and Ming Zhao. 2011. Towards Scalable Application Checkpointing with Parallel File System Delegation. In *Proceedings of the 6th IEEE International Conference on Networking, Architecture, and Storage*. IEEE, 130–139.

[4] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. 2009. PLFS: A Checkpoint Filesystem for Parallel Applications. In *Proceedings of the IEEE/ACM Conference on High Performance Computing Networking, Storage and Analysis*. 1–12.

[5] Ramon Canal, Carles Hernandez, Rafa Tornero, Alessandro Cilardo, Giuseppe Massari, et al. 2020. Predictive Reliability and Fault Management in Exascale Systems: State of the Art and Perspectives. *ACM Computing Surveys (CSUR)* 53, 5 (2020), 1–32.

[6] Franck Cappello, Geist Al, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. 2014. Toward Exascale Resilience: 2014 Update. *Supercomputing Frontiers and Innovations: an International Journal* 1, 1 (2014), 5–28.

[7] Yun Seok Chang, Sun Young Cho, and Bo Yeon Kim. 2003. Performance Evaluation of the Striped Checkpointing Algorithm on the distributed RAID for Cluster Computer. *Lecture Notes in Computer Science (LNCS)* 2658 (June 2003), 955–962.

[8] Yong Chen, Xian-He Sun, Rajeev Thakur, Philip C Roth, and William D Gropp. 2011. LACIO: A new collective I/O strategy for parallel I/O systems. In *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 794–804.

[9] Yong Chen, Xian-He Sun, Rajeev Thakur, Huaiming Song, and Hui Jin. 2010. Improving Parallel I/O Performance with Data Layout Awareness. In *Proceedings of the IEEE International Conference on Cluster Computing*. IEEE, 302–311.

[10] Tzi-Cker Chiueh and Peitao Deng. 1996. Evaluation of Checkpoint Mechanisms for Massively Parallel Machines. In *Proceedings of the Annual Symposium on Fault Tolerant Computing*. IEEE, 370–379.

[11] Manuel Costa, Paulo Guedes, Manuel Sequeira, Nuno Neves, and Miguel Castro. 1996. Lightweight Logging for Lazy Release Consistent Distributed Shared Memory. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vol. 10. 59–73.

[12] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*. 3.

[13] Ifeanyi P. Egwutuoha, David Levy, Bran Selic, and Shiping Chen. 2013. A Survey of Fault Tolerance Mechanisms and Checkpoint/Restart Implementations for High Performance Computing Systems. *The Journal of Supercomputing* 65 (2013), 1302–1326.

[14] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. 2011. An Overview of the HDF5 Technology Suite and its Applications. In *Proceedings of the EDBT/ICDT Workshop on Array Databases*. 36–47.

[15] Sanjay Ghemawat and Jeff Dean. 2014. LevelDB, A Fast and Lightweight Key/Value Database Library by Google.

[16] Kai Hwang, Hai Jin, Roy Ho, and Wonwoo Ro. 2000. Reliable Cluster Computing with a New Checkpointing RAID-x Architecture. In *Proceedings of the 9th Heterogeneous Computing Workshop (HCW 2000)*. IEEE, 171–184.

[17] Quincey Koziol and R. Matzke. 1998. *HDF5 – A New Generation of HDF: Reference Manual and User Guide*. Champaign, IL.

[18] Michael Laufer and Erick Fredj. 2022. High Performance Parallel I/O and In-Situ Analysis in the WRF Model with ADIOS2. *arXiv preprint arXiv:2201.08228* (2022).

[19] Jianwei Li, Wei keng Liao, Alok N. Choudhary, Robert B. Ross, Rajeev Thakur, William Gropp, Robert Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. 2003. Parallel netCDF: A High-Performance Scientific I/O Interface. In *Proceedings of the 15th ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'03)*. ACM, Pheonix, AZ, 39–50.

[20] Kai Li, Jeffrey F. Naughton, and James S. Plank. 1994. Low-latency, Concurrent Checkpointing for Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems* 5, 8 (1994), 874–879.

[21] Qing Liu, Jeremy Logan, Yuan Tian, Hasan Abbasi, Norbert Podhorszki, Jong Youl Choi, Scott Klasky, Roselyne Tchoua, Jay Lofstead, Ron Oldfield, et al. 2014. Hello ADIOS: The Challenges and Lessons of Developing Leadership Class I/O Frameworks. *Concurrency and Computation: Practice and Experience* 26, 7 (2014), 1453–1473.

[22] Jay F. Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. 2008. Flexible IO and Integration for Scientific Codes Through the Adaptable IO System (ADIOS). In *Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments*. 15–24.

[23] Avinash Maurya, Bogdan Nicolae, M. Mustafa Rafique, Thierry Tonellot, and Franck Cappello. 2021. Towards Efficient I/O Scheduling for Collaborative Multi-level Checkpointing. In *Proceedings of the 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 1–8.

[24] Robert McLay, Doug James, Si Liu, John Cazes, and William Barth. 2014. A User-friendly Approach for Tuning Parallel File Operations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*. IEEE, 229–236.

[25] Kshitij Mehta, John Bent, Aaron Torres, Gary Grider, and Edgar Gabriel. 2012. A Plugin for HDF5 using PLFS for Improved I/O Performance and Semantic Analysis. In *SC 2012 Companion: High Performance Computing, Networking Storage and Analysis*. IEEE, 746–752.

[26] Phil Miller, Shen Li, and Chao Mei. 2011. Asynchronous Collective Output with Non-dedicated Cores. In *Proceedings of the IEEE International Conference on Cluster Computing*. IEEE, 494–502.

[27] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R. De Supinski. 2010. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*. IEEE, 1–11.

[28] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-Structured Merge-Tree (LSM-tree). *Acta Informatica* 33 (1996), 351–385.

[29] Xiangyong Ouyang, Karthik Gopalakrishnan, Tejus Gangadharappa, and Dhabaleswar K. Panda. 2009. Fast Checkpointing by Write Aggregation with Dynamic Buffer and Interleaving on Multicore Architecture. In *Proceedings of the International Conference on High Performance Computing (HiPC)*. IEEE, 99–108.

[30] James S. Plank. 1993. *Efficient Checkpointing on MIMD Architectures*. Princeton University.

[31] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. 1995. Libckpt: Transparent Checkpointing Under Unix. In *Usenix Winter Technical Conference*. 213–223.

[32] James S. Plank, Jian Xu, and Robert H. B. Netzer. 1995. *Compressed Differences: An Algorithm for Fast Incremental Checkpointing*. Technical Report. Citeseer.

[33] Raghunath Rajachandrasekar, Adam Moody, Kathryn Mohror, and Dhabaleswar K. Panda. 2013. 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*. 143–154.

[34] Mendel Rosenblum and John K. Ousterhout. 1992. The Design and Implementation of a Log-structured File System. *ACM Transactions on Computer Systems (TOCS)* 10, 1 (1992), 26–52.

[35] Kento Sato, Adam Moody, Kathryn Mohror, Todd Gamblin, Bronis R. De Supinski, Naoya Maruyama, and Satoshi Matsuoka. 2011. Towards an Asynchronous Checkpointing System. *IPSJ SIG Technical Report* 2011, 18 (2011), 1–8.

[36] Kento Sato, Adam Moody, Kathryn Mohror, Todd Gamblin, Bronis R. de Supinski, Naoya Maruyama, and Satoshi Matsuoka. 2014. Fmi: Fault Tolerant Messaging Interface for Fast and Transparent Recovery. In *Proceedings of the 28th International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1225–1234.
[37] Faisal Shahzad, Markus Wittmann, Thomas Zeiser, Georg Hager, and Gerhard Wellein. 2013. An Evaluation of Different I/O Techniques for Checkpoint/Restart. In *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing, Workshops and PhD Forum*. IEEE, 1708–1716.
[38] Luís Moura Silva and João Gabriel Silva. 1998. An Experimental Study About Diskless Checkpointing. In *Proceedings of the 24th EUROMICRO Conference*. IEEE, 395–402.
[39] M. Moura Silva, Bart Veer, and J. Gabriel Silva. 1994. Checkpointing SPMD Applications on Transputer Networks. In *Proceedings of the IEEE Scalable High Performance Computing Conference*. IEEE, 694–701.
[40] Rajagopal Subramaniyan, Eric Grobelny, Scott Studham, and Alan D. George. 2008. Optimization of Checkpointing-related I/O for High-performance Parallel and Distributed Computing. *The Journal of Supercomputing* 46 (2008), 150–180.
[41] Rajeev Thakur, William Gropp, and Ewing Lusk. 1999. Data Sieving and Collective I/O in ROMIO. In *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation (Frontiers'99)*. IEEE, 182–189.
[42] Nitin H. Vaidya. 1999. Staggered Consistent Checkpointing. *IEEE Transactions on Parallel and Distributed Systems* 10, 7 (1999), 694–702.
[43] Lipeng Wan, Qing Cao, Feiyi Wang, and Sarp Oral. 2017. Optimizing Checkpoint Data Placement with Guaranteed Burst Buffer Endurance in Large-scale Hierarchical Storage Systems. *J. Parallel and Distrib. Comput.* 100 (2017), 16–29.
[44] Steven A. Wright, Simon D. Hammond, S. J. Pennycook, Iain Miller, J. A. Herdman, and Stephen A. Jarvis. 2012. LDPLFS: Improving I/O Performance without Application Modification. In *Proceedings of the 26th IEEE International Parallel & Distributed Processing Symposium Workshops & PhD Forum (IPDPSW'12)*. IEEE, Shanghai, China, 1352–1359.
[45] Steven A. Wright and Stephen A. Jarvis. 2015. Quantifying the Effects of Contention on Parallel File Systems. In *Proceedings of the 29th IEEE International Parallel & Distributed Processing Symposium Workshops & PhD Forum (IPDPSW'15)*. IEEE, Hyderabad, India, 932–940.
[46] Weixia Xu, Yutong Lu, Qiong Li, Enqiang Zhou, Zhenlong Song, Yong Dong, Wei Zhang, Dengping Wei, Xiaoming Zhang, Haitao Chen, et al. 2014. Hybrid Hierarchy Storage System in MilkyWay-2 Supercomputer. *Frontiers of Computer Science* 8 (2014), 367–377.
[47] John W. Young. 1974. A First Order Approximation to the Optimum Checkpoint Interval. *Commun. ACM* 17, 9 (1974), 530–531.
[48] Gengbin Zheng, Lixia Shi, and Laxmikant V. Kalé. 2004. FTC-Charm++: An In-memory Checkpoint-based Fault Tolerant Runtime for Charm++ and MPI. In *Proceedings of the IEEE International Conference on Cluster Computing*. IEEE, 93–103.

A APPENDIX

A.1 Artifact Description

A.1.1 *HPC: OS.*

- Linux: Flight Direct 2018.3 (Based on CentOS Linux 7 (Core))

A.1.2 *HPC: Modules Loaded.*

- data/HDF5/1.10.7-gompi-2020b
- compiler/GCC/11.3.0
- devel/CMake/3.24.3-GCCcore-11.3.0
- mpi/OpenMPI/4.1.4-GCC-11.3.0
- lib/zlib/1.2.12-GCCcore-11.3.0
- lib/lz4/1.9.3-GCCcore-11.3.0
- lib/libunwind/1.6.2-GCCcore-11.3.0
- devel/Boost/1.79.0-GCC-11.3.0

A.1.3 *Programming Languages.*

- LSMIO: C++ 17 (required)

A.1.4 *Library Dependencies: Manually Added.*

- git clone https://github.com/gflags/gflags.git
- branch master
- commit a738fdf9338412f83ab3f26f31ac11ed3f3ec4bd

- git clone https://github.com/google/googletest.git
- branch main
- commit e9fb5c7bacc4a25b030569c92ff9f6925288f1c3

- git clone https://github.com/google/glog.git
- branch master
- commit 674283420118bb919f83ceb3d9dee31ef43ff3aa

- git clone https://github.com/fmtlib/fmt.git
- branch master
- commit f6276a2c2b76c54c3a659d5fee5557c7bec95a0c

- git clone https://github.com/ornladios/ADIOS2.git
- branch release_29
- commit 03552904256f9430ecff31033c1c6bb05a364d45

- git clone https://github.com/google/leveldb.git
- branch main
- commit fb644cb44539925a7f444b1b0314f402a456c5f4

- git clone https://github.com/facebook/rocksdb.git
- branch 8.1.fb
- commit 7d2d9518fbcc72d21cb0a4a7397de0b5ab576a17

A.1.5 *Binary Dependencies: Manually Added.*

- git clone https://github.com/hpc/ior.git
- branch main
- commit c2386ed7b85559030ef20dc68cd7ed75afdc244e

A.1.6 Benchmarking Parameters. For all benchmarks all the block-sizes mentioned in Section 4 are used.

- IOR/baseline and IOR/HDF5 benchmarks: We set transfer-size equal to the block-size.
- ADIOS2 benchmarks used the same parameters as IOR. Additionally, ADIOS2 specific parameters were:
  - BufferChunkSize = 32MB
  - StripeSize = MinDeferredSize (Transfer size values used from IOR)
  - Asynchronous writes enabled
- LSMIO/baseline as well as the LSMIO/plugin in ADIOS2 benchmarks: We used the same parameters and in some cases translated from ADIOS2 such as BufferChunkSize.

For all benchmarks we configured our HPC environment to run each task on a separate physical node:

- `--ntasks=<concurrency: number of tasks>`
- `--nodes=<concurrency: number of tasks>`
- `--ntasks-per-node=1`
- `--ntasks-per-socket=1`
- `--cpus-per-task=1`

A.1.7 Measurements. IOR is the benchmark referenced by the Lustre file system. We used it as the baseline and for the HDF5 measurements.

For LSMIO measurements we started measuring right after the first MPI barrier and before the first I/O operation until after the last I/O operation and a second MPI barrier. We used a monotonic clock and relied on the steady_clock implementation by GCC C++. In addition,

- For the ADIOS2 library – including when LSMIO plugin is used – we called `PerformPuts()` and then `close()`
- For LSMIO baseline, we sent the last `DB::Put()` call which triggers an automatic flush.