MLBS: Transparent Data Caching in Hierarchical Storage for Out-of-Core HPC Applications

Tariq Alturkestani*, Thierry Tonellot**, Hatem Ltaief*,
Rached Abdelkhalak*, Vincent Etienne**, and David Keyes*
*Extreme Computing Research Center, King Abdullah University of Science and Technology, Thuwal, Saudi Arabia
**Exploration and Petroleum Engineering Advanced Research Center, Saudi Aramco, Dhahran, Saudi Arabia
Email: *{firstname.lastname}@kaust.edu.sa, **{thierrylaurent.tonellot,vincent.etienne}@aramco.com

***Abstract*—Out-of-core simulation systems produce and/or consume a massive amount of data that cannot fit on a single compute node memory and that usually needs to be read and/or written back and forth during computation. I/O data movement may thus represent a bottleneck in large-scale simulations. To increase I/O bandwidth, high-end supercomputers are equipped with hierarchical storage subsystems such as node-local and remote-shared NVMe and SSD-based Burst Buffers. Advanced caching systems have recently been developed to efficiently utilize the multi-layered nature of the new storage hierarchy. Utilization of software components results in more efficient data accesses, at the cost of reduced computation kernel performance and limited numbers of simultaneous applications that can utilize the additional storage layers. We introduce MultiLayered Buffer Storage (MLBS), a data object container that provides novel methods for caching and prefetching data in out-of-core scientific applications to perform asynchronously expensive I/O operations on systems equipped with hierarchical storage. The main idea consists in decoupling I/O operations from computational phases using dedicated hardware resources to perform expensive context switches. MLBS monitors I/O traffic in each storage layer allowing fair utilization of shared resources while controlling the impact on kernels' performance. By continually prefetching up and down across all hardware layers of the memory/storage subsystems, MLBS transforms the original I/O-bound behavior of evaluated applications and shifts it closer to a memory-bound regime. Our evaluation on a Cray XC40 system for a representative I/O-bound application, seismic inversion, shows that MLBS outperforms state-of-the-art filesystems, i.e., Lustre, Data Elevator and DataWarp by 6.06X, 2.23X, and 1.90X, respectively.**

***Index Terms*—Multilayered Buffer System, Asynchronous I/O, Caching and Prefetching, Lustre, Burst Buffer, DataWarp, Data Elevator, Parallel File System**

I. Introduction

Memory and storage systems have not maintained the same rate of technology scaling as processing for many years, creating an I/O performance gap between the subsystems [1], [2]. Out-of-core simulations pay the highest penalty. These systems typically produce and/or consume a massive amount of data that cannot fit on a single compute node memory, and hence may require to read and write back and forth this data many times for computation. I/O data movement can thus be a bottleneck in large-scale simulations. In fact, many modern scientific applications running on HPC systems typically induce high latency blocking for I/O [3], [4], [5].

As we approach exascale, the limited on-node memory capacity makes I/O one of the main bottlenecks for many critical simulations, hindering scientific productivity [6]. To enhance the I/O performance on HPC systems, the storage subsystem is being expanded in a hierarchical manner, adding high-throughput intermediate layers in between Dynamic Random Access Memory (DRAM) and traditional Hard Disk Drive (HDD) storage. Advances in memory architecture have made it feasible and affordable to include multiple heterogeneous storage media in such systems [7]. Several layers of hardware storage, such as on node-local Non-Volatile Memory (NVMe) and Solid State Drive (SSD), and remote-shared SSD-based Burst Buffers are used to leverage the high I/O bandwidth and improve the I/O performance. Due to the importance of closing the gap between I/O and compute, several production supercomputers are currently equipped with Burst Buffers [8]. For example, the Aurora supercomputer at the Argonne National Laboratory, Summit at the Oak Ridge National Laboratory, and Sierra at the Lawrence Livermore National Laboratory are or will be equipped with node-local Burst Buffers. Meanwhile, Trinity supercomputer at the Los Alamos National Laboratory, Cori at the Lawrence Berkeley National Laboratory and Shaheen-2 at the King Abdullah University of Science and Technology have adopted remote-shared SSD-based Burst Buffers [9].

These intermediate layers help to reduce the amount of time applications spend blocking for I/O. However, traditional Parallel File Systems (PFS) such as Lustre [10], GPFS [11] and PVFS [12] are not equipped to manage multi-layer storage subsystems. They support only a single layer of storage devices [13]. Since each layer is an independent subsystem, users are forced to study each layer independently and make necessary adjustments in their applications [14].

Several solutions have emerged in the recent decade to ease the burden on users. DDN IME [15], DataWarp [16] and Data Elevator [17] are parallel software systems that focus predominantly on bursty write operations. They redirect I/O calls from PFS to Burst Buffer and then asynchronously flush data to PFS. On read, they need to populate the Burst Buffer through manual stage-in operations. Since prefetching data to an intermediate storage is also necessary for many post-processing applications, systems that support one-way read

2640-0316/19/$31.00 ©2019 IEEE
DOI 10.1109/HiPC.2019.00046

operation, such as ARCHIE, have also been proposed [18]. Hermes and UniviStor are new systems that offer multi-tiered buffering solutions [19], [20]. Hermes utilizes users knowledge of data accesses for better data placement of files. UniviStor provides an independent parallel program that allows sharing of resources for multiple applications.

However, there are two main drawbacks with these solutions: 1) they require large fixed reservations on remote-shared Burst Buffers denying other applications from utilizing them, and 2) they use additional multi-threaded processes on compute nodes, thereby, damaging cache locality in multi-treaded computation kernels.

In this paper, we propose the design and implementation of a **M**ulti**L**ayered **B**uffer **S**torage (MLBS) that supports two-way caching of data for both read and write I/O operations to maximize the bandwidth utilization of multiple hardware storage layers. MLBS continuously prefetches data up and down across all hardware layers of the memory/storage media stack while ensuring that other applications can fairly use intermediate storage hardware. This non-intrusive framework provides simple APIs that enable applications to decouple the I/O operations from their critical paths, while eagerly pursuing their computation. MLBS demonstrates a versatile and asynchronous method for overlapping I/O with computations that decreases the strain on processing units and minimizes the performance impact on the computational kernels. Our three-layer implementation is designed to control the size used by each compute node in the Burst Buffer so that more nodes can be used with minimal impact on performance. To our knowledge, MLBS is the first framework to show the effectiveness of an asynchronous multilayered buffer system with real scientific applications. MLBS controls the performance impact on the computational kernels, while leveraging the Burst Buffer technology.

The main contributions of this paper are as follows.

- We deploy novel methods for caching and prefetching data across hierarchical parallel storage systems (§III).
- We implement a set of APIs that mimics C++ vector semantics and allows for collective MPI operations (§III-A).
- We design a parallel caching layer that minimizes the performance impact on multi-threaded computation kernels while increasing I/O throughput (§III-B).
- We present a deterministic HPC applications trace tool for online performance modeling (§IV-B, §V-A).
- We integrate MLBS into micro benchmark and three real scientific applications, Vector Particle-In-Cell (VPIC), Hardware Accelerated Cosmology Code (HACC), and Reverse Time Migration (RTM) (§IV-D).
- We evaluate MLBS on the distributed-memory Cray XC40 *Shaheen-2* system and show that MultiLayered Buffer Storage (MLBS) outperforms state-of-the-art file systems such as Lustre, Data Elevator and DataWarp by 6.06X, 2.23X and 1.90X, respectively (§V).

II. Background and Motivation

A. What are the current hardware/software solutions for out-of-core applications?

Out-of-Core applications perform computations on data that does not fit in main memory. Alternately, blocks of data are written and/or read from file systems depending on when each block is needed for computation. Current I/O libraries, such as POSIX, MPI-IO, HDF5, are responsible for data movement through applications I/O requests, while communicating independently with the file systems. Typically, they use variants of `memcpy()` as a transport method to copy data between applications memory space and file systems, e.g., Lustre, GPFS, PVFS, etc. This creates opportunities for the applications to overlap computation and I/O phases. Fig. 2 highlights these overlapping regions during a snapshot of a typical out-of-core application execution.

B. Is asynchronous I/O sufficient enough to overlap I/O and compute phases in iterative and bursty I/O applications with fast compute programming model?

I/O library calls such as `fwrite()`, `fread()` and `aio_write()`, `aio_read()` are asynchronous analog of blocking `write()` and `read()` system calls. They complete I/O requests in parallel alongside computation, thus, removing the strain of blocking I/O from applications critical path.

`fwrite()` and `fread()` calls pause the computation to copy the targeted data between application provided buffers and I/O library buffers using `memcpy()` and finally resume the flow of the application. Internally, in a second step, `fwrite()` and `fread()` copy the data between Operating System (OS)'s disk buffers and I/O library buffers. Applications can, therefore, use their buffers immediately after calls to `fwrite()` or `fread()` return while the OS kernel threads writes and read data directly form disk.

`aio_write()` and `aio_fread()` calls enqueue I/O request into a Pthread-based helper engine and return immediately to the application space. After the return, applications cannot use the pointer to targeted data for computation until the request is completed but free to do anything else. `aio_suspend()` is then used to wait until the pointer to targeted data is safe to use.

Without the loss of generality, the wall time (i.e. entire execution time) of an application that uses blocking I/O is the summation of computation time, Eq. (1), and I/O time, Eq. (2) times the number of iterations. On the other hand, using asynchronous I/O reduces the wall time to the maximum of either Eq. (1) or Eq. (2), as these operations are done in parallel.

$$\overbrace{\frac{\text{Data size (GiB)}}{\text{Processing speed (GiB/s)}}}^{\text{Compute Time}} \quad (1) \qquad \overbrace{\frac{\text{Data size (GiB)}}{\text{I/O bandwidth (Gib/s)}}}^{\text{I/O Time}} \quad (2)$$

However, when the processing throughput of a byte is greater than the throughput of asynchronous I/O library, which is limited by its internal memory buffers, I/O calls become

blocking. Therefore, asynchronous I/O functions, `fwrite()`, `fread()` and `aio_write()`, `aio_read()`, fail to overlap I/O and compute. In fact, due to Little's Law [22], when the utilization of internal I/O buffers exceeds 80%, I/O libraries may encounter infinite queue waiting time. Such scenario impedes current asynchronous I/O libraries to work as expected.

C. What is a Burst Buffer?

Due to the importance of closing the gap between I/O and compute throughput, several production supercomputers are currently equipped with high bandwidth intermediate storage subsystems knowing as Burst Buffers [8]. While most of the parallel storage systems in supercomputers are built with cost effective, low-bandwidth, high-capacity spinning disks HDDs, Burst Buffers are built with more expensive but high-bandwidth, low-capacity SSDs or NVMes [23].

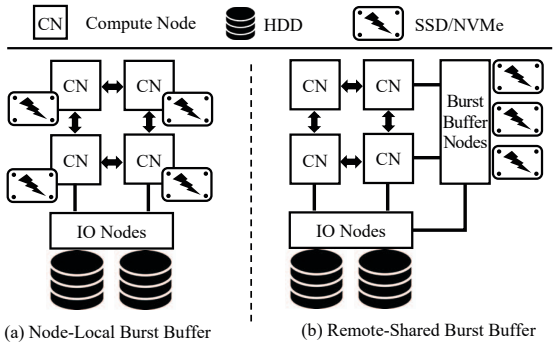

Fig. 1. Designs of (a) Remote-Shared and (b) Node-Local Burst Buffer

The two most common types of Burst Buffers are node-local and remote-shared. In the former design, Fig. 1(a), each node is equipped with a local SSDs or NVMe with limited capacity. For instance, on Summit supercomputer, each compute node has a local 1.6 TiB NVMe. In the latter design, Fig. 1(b), arrays of SSDs or NVMe are grouped in separate Burst Buffer nodes. These Burst Buffer nodes are located on the interconnect fabric and compute nodes access them through the network. For instance, on *Shaheen-2* supercomputer, all compute nodes share 1.5 PiT of high bandwidth SSDs located on the Burst Buffer nodes.

The choice between the two designs is beyond the scope of this paper. However, our proposed solution, MLBS, intends to hide the differences between the two designs from the applications' perspective.

D. What are the runtime opportunities for compute and I/O optimizations moving forward? A perspective from the Reverse Time Migration (RTM) application

Applications may deal with multiple heterogeneous file systems. For example, *Shaheen-2* supercomputer offers to its users an SSD-based DataWarp file system and a disk-based Lustre file system. I/O libraries must select or mix appropriate file systems in order to achieve better I/O and compute overlap [21].

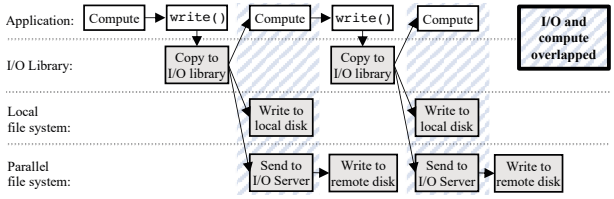

Fig. 2. I/O and compute phases overlap. The application writes to local or remote disks using appropriate file systems. The arrows show dependencies.

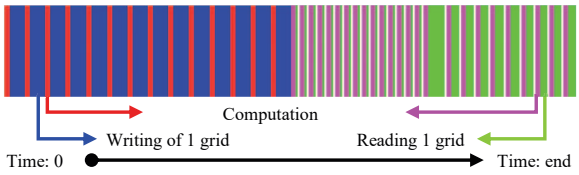

Fig. 3. Observed compute and I/O traces for a master iteration of RTM using *Shaheen-2* supercomputer with Lustre Parallel file system. Red and pink are forward and backward computation, respectively. Blue and green are the time RTM blocks for write and read operations, respectively.

In Fig. 3, we use our own tracer to understand how much time is spent in each compute and I/O phase in a run of RTM application. In the forward phase, the application computes 10 iterations on a 2 GiB grid before dumping it to the I/O library as a snapshot. The library then conducts the actual write to the PFS, i.e., Lustre. Once the forward phase is over, in the backward simulation and imaging phase, the application reads back previously written grids in Last In, First Out (LIFO) order and computes the so-called imaging condition every 10 iterations. Red and pink color bars represent compute times for forward and backward phases, and blue and green represent `write` and `read`. We can clearly observe how the backward phase read is faster in the beginning and slower later. This happens because the requested files in the beginning are still stored in DRAM and have not been evicted by the operation systems memory manager yet. Files at the end are brought back from Lustre and, hence, they take longer time. We can clearly identify runtime opportunities to overlap I/O read and/or write function calls with computational kernels from the application's critical path.

III. Design and Architecture of MLBS

Out-of-core scientific applications, such as Reverse Time Migration (RTM), write-read or read-write a large number of intermediate snapshots in a form of checkpoints on disks [5]. Applications typically compute on data arrays or vectors and issue I/O write/read request on them. We designed MLBS in a way that fulfills the aforementioned semantics. Our objective in designing MLBS is to hide I/O overhead from computation, while maintaining the following goals: 1) lightweight Application Program Interface (API) with minimal code and application design changes, 2) overlapping I/O and compute phases using asynchronous executions, 3) prefetching and caching across all multi storage layers, and finally 4) controlled impact on the computation kernels.

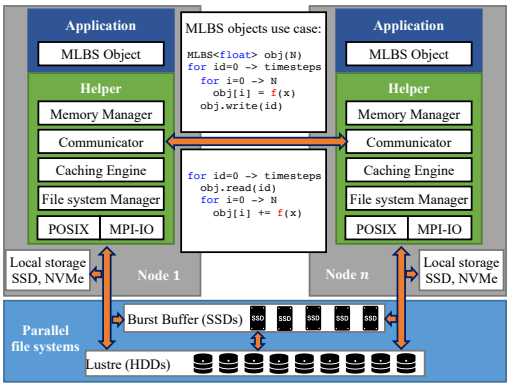

Fig. 4. Architecture and software stack of MLBS

```
1  template<class T> class MLBStore{
2  public:
3  // Constructors
4  MLBS();
5  MLBS(size_t numOfElements);
6  //Write/Read
7  void write(size_t snapID);
8  void read(size_t snapID);
9  // Rank to rank communications
10 void setComm(MPI_Comm comm);
11 void reduce(MPI_Op op, size_t root);
12 void allReduce(MPI_Op op);
13 // Dynamic allocation
14 void resize(size_t numOfElements);
15 void push_back(T& element);
16 //operator override
17 T& operator[] (const size_t key);
18 };
```

Fig. 5. MLBS C++ Class template and public APIs

In Fig. 4 we show the architecture and software stack of MLBS and its two building blocks, MLBS objects and MLBS helper engine. Users include MLBS objects in their parallel source code and use them as they would with standard arrays or vectors. One MLBS helper is launched per compute node which takes over write/read request and tries to achieve the design objective described above.

A. MultiLayered Buffer Storage (MLBS)

An MLBS object is a lightweight contiguous data storage container written as a C++ class template. It follows the basic semantics of the *vector* Standard Template Library (STL) container [24], i.e., indexing, dynamic memory allocation, etc. Additionally, it includes multiple APIs to enable I/O-compute overlap and parallel computation on its data. In Fig. 5 we list public MLBS APIs. The main objectives of the APIs are ease-of-use, non-intrusiveness and high throughput copy from application space to the DRAM allocation in MultiLayered Buffer Storage Helper (MLBS Helper) space. Therefore, with minimal code changes, application developers can easily integrate MLBS into their codes.

Constructors of MLBS, lines 4 and 5, reserve an initial block of contiguous DRAM memory. The allocated memory is used to hold `numOfElements` elements of type `T`, e.g. a million *float*, *double* or *int*.

Function `resize()`, line 14, enables dynamic memory reallocation. It resets the number of elements that MLBS can hold to `numOfElements`, `push_back(T & element)`, line 15, is used to append `element` to the end of MLBS objects after allocating necessary memory.

Function `write(id)` is used to copy all data elements to the MLBS Helper's local memory buffer. The data is stored as a record with ID `id`.

Since the copy operation is the first bottleneck of overlapping I/O and compute, we studied the performance of `memcpy()` and compared it to an alternative parallel copy method which uses vectorized Advanced Vector Extensions (AVX) instructions and parallel for loop, we elaborate with extensive details in Sec. §IV-A. Therefore, copy is done through AVX instructions and parallel loop.

Function `read(id)`, is used to load back data record `id` from MLBS Helper using memory swaps. Details about caching and prefetching strategies are provided in §III-B.

Functions `setComm()`, `reduce()` and `allReduce()` are used to make collective MPI operations within MLBS space. In domain decomposition type of work, for example, where multiple MPI ranks work on the same data, collective MPI-IO is used to allow each rank to write and/or read its own blocks of data. These collectives are also used to aggregate I/O request and assign one rank to conduct I/O requests. Supporting other MPI collective operations is straightforward.

B. MultiLayered Buffer Storage Helper (MLBSHelper)

Upon instantiation of an MLBS object, MLBS Helper is launched and pinned, along with application. The Pthread-based helper threads main objective is to handle all I/O operations: 1) moving the data between the application space and MLBS helper space, 2) conducting the actual write and read operations from main memory to subsequent storage layers, 3) continuously pushing data up and down across hierarchical storage systems such as Burst Buffer and Lustre, and 4) caching data in and out based on data access patterns.

We built MLBS Helper with the following components:

Memory Manager. When an MLBS object is initialized, MLBS reserves a small block of available DRAM and assigns it to the MLBS Helper. The exclusive segment of DRAM is used to cache some of the intermediate records. DRAM caching layers allow to reduce the latency of I/O requests. We have implemented a First In, First Out (FIFO) cache evict policy on this layer. On evict, records are pushed down to the next buffering layer and metadata table gets updated. In addition, ownership of the record gets transferred to the file system manger.

File System Manager. Once a record gets evicted from DRAM, the file system manager is then responsible to move the record from DRAM to the next fastest storage subsystem. On construction phase, MLBS attaches all available filesystems to MLBS. For example, in supercomputers such as *Shaheen-2* that host two PFS such as SSD-based DataWarp and

HDD-based Lustre, MLBS attaches DataWarp system as an intermediate layer and Lustre as a persistent layer. This natural order implies that data objects start their life cycle in DRAM and then get pushed to the Burst Buffer when DRAM is full. They then get further pushed to Lustre when the Burst Buffer is full. Depending on the prefetching strategy, the same data file may then be pulled back from Lustre to Burst Buffer to MLBS DRAM space, and finally, transferred to the application space using the `read()` method, as shown in Algorithm 2 and §III-B. The actual `read` and `write` is done using MPI-IO with the option to use POSIX as well. When the Burst Buffers are used, DataWarp's API are used to asynchronously stage-in and stage-out the records [16].

Communication Layers. Any exchange between the application space and MLBS Helper engines, including inter-helpers communications, should be carefully handled. Therefore, multiple reliable communication layers are built into MLBS Helper to guarantee smooth and fast data traffic between the application and the various storage layers. By default, MPI one sided communicators are used to exchange messages between helpers on different nodes. While in a scenario in which helpers are in one node, sharing the same physical DRAM, POSIX shared memory APIs are used.

Caching Engine Algorithm 1 details how MLBS Helper overlaps compute and I/O and continuously caches and prefetches intermediate record for applications with either LIFO or FIFO data access patterns, with default set to FIFO. MLBS Helper manages the data traffic in the background, while the application carries on its computations. The helper's caching engine consists of two phases. In the first phase, the engine loops over the 1st layer of buffer zone, i.e., DRAM, and evicts oldest data to the 2nd layer, e.g., Burst Buffer (line 5). It may further push down the oldest files from 2nd layer to the 3rd layer, e.g., Lustre. In the 2rd phase, i.e., the read phase, the helpers wait for the application to consume the data that are still in the MLBS 1st layer. As soon as a buffer slot is empty in the 1st layer, the helper fills up the buffer slot with the data file from the 2nd layer (line 12). It will also pop up another data file from the 3rd layer to the 2nd layer. The selection of the file depends on whether the application processes data in LIFO or FIFO. MLBS Helper maintains a full pipeline across storage layers, while maximizing I/O bandwidth and occupancy. Thanks to the helper threads, its API permits the application developers to instrument the I/O accesses with flexible memory interface, while customizing the helper engine to match the I/O patterns.

Metadata Manager. A lookup table is embedded in the MLBS Helper space. The table keeps track of the linage history of each record from the time is it created to the time it is removed. It also keeps track of the performance of each layer of storage and feeds necessary data to our I/O tracer (§V-A) and performance modeler (§IV-B). MLBS Helper engine uses the information on this table to reroute I/O requests in order to reduce the latency that is caused by shared resources internal and external interference [21], [25]

Algorithm 1: MLBS Helper LIFO and FIFO Buffering and Prefetching Method

```
1  begin
2      while Application is in the writing phase do
3          foreach MLBS Helper DRAM slot do
4              if First Disk is full then
5                  select first file (LIFO) or newest file (FIFO) in first disk
                     and push to second disk  // Selection of LIFO
                     or FIFO depends on the applications
                     data access patterns.
6              push file to first disk
7              Update the location of the file in the MLBS Helper metadata
                 table
8              clear slot and add to free pool
       // Switching to read mode
9      while There are files in 2nd and 3rd layer do
10         while MLBS Helper DRAM buffers not full            // Read
             operation in the application space frees up
             MLBS Helper DRAM buffers
11         do
12             Load most recent file(LIFO) or oldest (FIFO) from first disk
                 to an empty slot in MLBS Helper DRAM
13             Update the location of the file in the MLBS Helper metadata
                 table
14             Move most recent file(LIFO) or oldest (FIFO) file in second
                 disk to first disk
```

Algorithm 2: MLBS API functions: Write and Read

```
1  Function Write(id):
2      dest ← FindEmptymlbshelperSlot()      // blocking call
3      #pragma omp parallel   // parallel memory to memory copy
4      for i = 0 → datasize do
5          dest[i] = data[i]
6      Update file location in MLBS Helper metadata table
7  Function Read(id):
8      while file (id) is not yet loaded in MLBS DRAM  // blocking busy
         loop
9      do
10         WAIT( )
11     src ← GetFilePointer(filename)
12     SWAP(src, data)
13     Update file location in MLBS Helper metadata table
```

IV. Implementation Details

A. Memcpy vs. Parallel and Vectorized AVX

In order to measure the impact of memory copy, we study the throughput of copying arrays of different sizes using `memcpy()` and compare it against a technique, which uses vectorized AVX instructions and parallel OpenMP for loop.

Vectorized AVX `vmov{a,u,nt}p{s,d}` instructions copy data that are, aligned (a), unaligned (u) or non-temporal (nt) and packed as single (ps) or double (pd) floating points between main memory and 256-bit or 512-bit wide special registers. In its aligned (a) and unaligned (u) form, the instruction modifies cache entry on update while in the non-temporal (nt) form it skips cache and sends data directly to main memory. Compilers try to select the best one based on compile-time analysis. Alternatively, MLBS may force one or the other at run time depending on record size.

Fig. 6 shows that `memcpy()` can at best perform at 16 GiB/s even when both the source and destination fit in CPU cache, because it uses one hardware core only. However, with

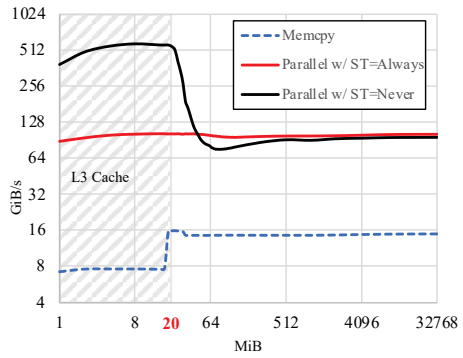

Fig. 6. Maximum copy throughput achieved when using `memcpy` or vectorized `parallel omp for` under different data sizes. The parallel code is compiled with streaming stores compiler flag set to `always` or `never`. The shaded gray area indicates that copied data fit in L3 cache.

parallel and temporal vectorized AVX instruction `vmovups`, the throughput reaches up to 512 GiB/s for data that always fit in cache and and 90 GiB/s when data is larger using non-temporal vectorized AVX instruction, `vmovntps`.

We used the Performance Application Programming Interface (PAPI) hardware event counter [26] to analyze the L3 cache hit ratio when copying data with vectorized instructions. Table I shows that using non-temporal `vmovups` (streaming stores) on data that fit within L3 cache results in lowering cache hit ratio, thereby, lowering copy throughput. When a cache miss occurs, copy operation cannot proceed until data are brought back to L3 cache from DRAM. For data that fit in cache it is always better to use temporal instructions.

Array Size	Streaming Stores=Always	Streaming Stores=Never
1 MiB - 20 MiB	0.77	0.99
22 MiB - 30 MiB	0.77	0.94
32 MiB - 62 MiB	0.77	0.74
64 MiB - 32 GiB	0.76	0.70

TABLE I
L3 CACHE HIT RATIOS WHEN COPYING DATA USING PARALLEL OMP FOR LOOP WITH AND WITHOUT STREAMING STORES.

B. Lighting the I/O Path

We have developed an easy to use lightweight event-tracer to visualize the different stages in any given application. The tracer can be used by users to observe the performance of each component in the code and to make effective code or system modification. Before and after each compute and I/O phase, users call our tracing library to start and end a tracing event. The library records the start and end wall times and draws the events on an image file as seen in Fig. 3 and 11. We use the tracer to shows how computations and I/Os are overlapped.

C. MLBS Hints

At the initialization step of MLBS, users can change the default parameters of MLBS and declare the number of helpers and the communication transport between helpers to be used. They can also specify the available capacity of each layer. This is especially important when a scarce and shared layer such as the Burst Buffer is used. Users also can decide whether each helper should be writing their data as independent files or group all their tasks in one file per helper. While the former option is easy and neat, it is known that PFS such as Lustre poorly perform when the number of small files is large. The latter option on the other hand is preferred especially, when the access for the data in the reading phase is sequential. While not required, these hints can bring additional performance gain for applications with deterministic I/O behavior.

D. MLBS Integration.

Algorithm 3 presents the pseudo-code of an application with MLBS API integration. This pseudo-code represents a class of HPC applications that write then read snapshots in either *FIFO* or *LIFO* order. We have integrated MLBS with

Algorithm 3: Integrating MLBS into applications

```
   Data: optional Number of helpers, size of grids, number of I/O ops, DRAM
         allocation and access pattern
   Result: Compute and I/O overlapped
1  Function main()
2      MLBSINIT()
3      MLBSHINTS(numHelpers, datasize, ioOps, mem, patters)
       // optional: setup all related parameters
       // Start forward phase
4      for i = 0 → numOfIntrs do
5          MLBS[i] ← COMPUTE(data)
6          if i mod snapRatio == 0 then
7              MLBS.WRITE(i)
       // Start processing phase (FIFO or LIFO order)
8      for i = numOfIntrs → 0 do
           // Alternatively i = 0 → numOfIntrs
9          MLBS[i] ← COMPUTE(data)
10         if i mod snapRatio == 0 then
11             MLBS.READ(i)
12             finalOutput ← CORRELATE(finalOutput, MLBS)
```

micro-benchmark, two I/O kernel from scientific applications Vector Particle-In-Cell (VPIC) and Hardware Accelerated Cosmology Code (HACC), and a production RTM. While the compute functions of the two I/O kernels are not considered in the evaluation, the micro-benchmark and the RTM application includes it. The RTM application relies on OpenMP programming models for the parallel computational kernel implementation (i.e., the stencil computation kernel in lines 5 and 9 from Algorithm 3), which fully takes advantage of instruction vectorization (i.e., AVX-2) on the underlying hardware. MLBS uses a C++ and Pthread-based implementation for flexibility purposes. To properly ensure the application OpenMP threads and MLBS Pthreads helper coexist, the total number of threads should match the number of physical cores, to prevent oversubscription overheads. Only single-threaded MLBS implementation is demonstrated in the paper to avoid performance slow down on the compute kernel side, since it is part of the critical path. However, the MLBS Pthread-based implementation can support more threads, for instance, in case the memory-to-memory copies (i.e., lines 7 and 11 from Algorithm 3) become a bottleneck. One can clearly notice the possible overlapping between lines 5 and 7 as well as between lines 9 and 11 during the forward and the backward

integration, respectively. The final image is then generated thanks to the image condition, as shown in line 12.

Ultimately, the makespan of the applications VPIC, HACC or RTM should be minimized by balancing compute and I/O workloads accordingly through MLBS.

V. Evaluation

To evaluate MLBS, we integrated our proposed storage object into our own micro-benchmark workflow that writes and reads and computes on large *3D* array §V-B. Then we evaluated MLBS on real scientific application I/O workloads from physics and cosmology §V-C. Finally we evaluated MLBS on real production workflow from geoscience §V-D. In our evaluation we compared MLBS against writing and reading directly to state-of-the-art storage management systems (i.e., Lustre, DataWarp and Data Elevator).

A. Experimentation Setup

We have used the Cray XC40 supercomputer *Shaheen-2* as an experimental platform. *Shaheen-2* is a 6174-node system, each with a dual-socket 16-core Intel Haswell E5-2698 V3 and 128 GiB RAM. *Shaheen-2* operates two parallel file systems, a 1.5 PiB SSD-based DataWarp and a 17.6 PiB HDD-based Lustre. The sustained bandwidth of SRAM based on the Stream benchmark [27] is 105.9 GiB/s. The I/O bandwidth measured with the IOR benchmark [28] is 1.5 TiB/s and 500 GiB/s, for Burst Buffer and Lustre, respectively.

I/O workload. We have integrated MLBS into our own micro-benchmark, two I/O kernels form scientific application VPIC and HACC and a production RTM code used in geoscience/oil and gas industry.

In our MPI+OMP based micro-benchmark, in a given time-step loop, each MPI process computes independently on a large *3D* array using all OMP threads. Depending of the checkpoint frequency that we set, i.e., how many time-steps before each checkpoint, each MPI process, writes to disk the entire array. Once the last time step is reached, the time-step loops restarts and the checkpoints are read from disk before they get computed on.

The two I/O kernels we used are VPIC and HACC. Vector Particle-In-Cell (VPIC) is a general purpose simulation code for modeling kinetic plasmas in spatial multi-dimensions. We selected the I/O kernel from the ExaHDF5 Parallel I/O Kernel (PIOK) Suite [29], [30]. Hardware Accelerated Cosmology Code (HACC) is an N-body method GPU implementation that simulates the formation of structure in collisionless fluids under the influence of gravity in an expanding universe. The simulation helps to understand the evolution of the universe from inception to today [31]. We used the I/O kernel from the CORAL Benchmark Codes [32], [33].

To showcase MLBS in production environment, we integrated MLBS into an RTM code. Reverse Time Migration (RTM) is a wave equation based depth seismic migration technique used in the oil and gas industry, particularly in complex geological environments such as subsalt exploration. The RTM image is formed by cross-correlating a forward simulated source wavefield with an adjoint (i.e., reverse time) simulation from receivers. In a first phase, the source wavefield is reversed in time by storing at predetermined imaging time steps (snapshots) the propagation history. Due to the huge volume represented by the snapshots, they are usually compressed and offloaded to disks. In a second phase, when the backward receiver propagation reaches one of the imaging time steps, the corresponding source snapshots are read back from disk, decompressed, and correlated with the receiver snapshots to incrementally calculate the image condition, until the final image is eventually obtained [34], [35].

Comparisons. In this evaluation, we compare our proposed transparent storage object solution with Lustre [10], DataWarp [16] and Data Elevator [17]. Lustre is the most common PFS on supercomputers. It does not provide any caching support and applications write and read directly to and from it. DataWarp is a the file system that manages the Burst Buffer on the Cray XC40 supercomputer. With its API, applications can use the Burst Buffer to cache, stage-in and stage-out files. This API enables the Burst Buffer as an intermediate stage between DRAM and Lustre. Data Elevator is a software library that uses the Burst Buffer as a caching layer. It relies on DataWarp to maintain the Burst Buffer and Lustre resources. It moves files asynchronous between the Burst Buffer and Lustre. While all the systems above deal with files on `write` and `read` library calls, MLBS deals directly with data. Applications that use MLBS do not have to worry about I/O at all. In fact, they write to and read from MLBS as if they are accessing DRAM, e.g., arrays and vectors access. Internally, MLBS asynchronous uses caching zones in DRAM and Burst Buffer to smoothly move data from DRAM to Burst Buffer to Lustre and the other way around. MLBS unifies DRAM, Burst Buffer and Lustre and mitigate the I/O rates among these layers. Unlike Data Elevator, MLBS does not require additional programs to run with applications.

Performance metric. The goal of MLBS is to reduce the time that is spent by application blocking for I/O, by overlapping I/O and compute. Thus, we measure the time spent for accessing or releasing data with either `write()` and `read()` library calls using Lustre, DataWarp or Data Elevator or through MLBS direct data access. We also evaluate the performance of MLBS using the performance model described below. We run each application three times and report the best performing results.

Performance modeling. We have integrated a model to monitor I/O performance at runtime. In fact, we can statically determine the entire application's execution time from the time breakdown of the major deterministic phases, i.e., kernel time (modeling and processing), write time, and read time. However, since I/O operations are done on shared resources, such as Lustre or Burst Buffer, performance fluctuations may occur depending on the availability of PFS resources. Therefore, we have added an online performance modeling tool into MLBS to better assess its performance impact, at any given state of the parallel file system. The model is actually straightforward, since it leverages the fact that I/O and compute operations may

I/O Ratio	I/O traffic GiB	I/O time (s)			
		Lustre	Burst Buffer	Data Elevator	MLBS
50	560	864.8	323.2	335.9	6.1
40	700	1,088.9	390.6	403.5	7.4
30	934	1,483.3	531.3	542.9	9.9
20	1,400	2,275.7	815.0	846.4	15.0
15	1,867	3,041.5	1,082.2	1,115.5	19.9
10	2,801	4,716.8	1,624.9	1,655.7	189.1
5	5,602	9,488.8	3,143.3	3,223.3	1,242.2
3	9,336	16,066.4	5,235.6	5,293.4	2,999.4
2	14,004	24,185.8	7,894.1	8,083.3	5,307.4
1	28,008	48,910.0	15,491.2	15,803.1	12,077.5

TABLE II
I/O PERFORMANCE IMPACT OF THE REFERENCE MICRO-BENCHMARK APPLICATION WHEN USING **LUSTRE BURST BUFFER**, **DATA ELEVATOR** AND **MLBS** ON 4 GIB GRID SIZE AND VARIOUS I/O RATIOS.

I/O Ratio	Wall time speedup X					
	Burst Buffer		Data Elevator		MLBS	
	Expected	Actual	Expected	Actual	Expected	Actual
50	1.28	1.26	1.24	1.21	1.43	1.43
40	1.33	1.32	1.30	1.28	1.55	1.55
30	1.44	1.41	1.40	1.37	1.76	1.76
20	1.61	1.56	1.56	1.50	2.16	2.15
15	1.75	1.69	1.69	1.62	2.56	2.55
10	2.00	1.91	1.93	1.83	3.41	3.13
5	2.39	2.29	2.33	2.20	3.59	3.59
3	2.67	2.55	2.62	2.48	3.41	3.64
2	2.84	2.69	2.80	2.60	3.29	3.60
1	3.07	2.94	3.04	2.86	3.27	3.62

TABLE III
COMPARISON OF EXPECTED AND SUSTAINED WALL TIME SPEEDUP BASED ON OUR PERFORMANCE MODEL

be overlapped. Therefore, in-between I/O operations, the wall time speedup window is bounded by

$$Speedup = \frac{t_{Reference}}{max\{(t_{kernel} + t_{memcopy}), t_{I/O}\}} \quad (3)$$

In this evaluation $t_{Reference} = t_{kernel} + t_{I/O_{Lustre}}$ yields the slowest execution time. Using the size of the dataset and the time taken to transfer it during the time window, we can assess the actual sustained I/O throughput obtained during the time window. This realistic I/O bandwidth can then be used, instead of the sustained I/O bandwidth obtained with the IOR benchmark [28], to evaluate and predict MLBS performance impact.

B. Evaluation of our micro-benchmark

We wrote a C++ *MPI+OMP* micro-bench that follows the complete workflow of an out-of-core adjoint state simulation. This application consists of modeling and processing phases. In the two phases, a compute kernel uses all available threads and vectorized instructions to computes on a large *3D* array. The application computes on the array $timeSteps$ number of times in each phase. At run time, users can set the size of the floating points array and the number of $timeSteps$. The modeling process includes dumping to disk the entire array at every *I/O ratio* time step. The processing phase reads back these arrays and computes on them. In this evaluation, we set the number of time steps to 3500 and increased the *I/O ratio* from 1 to 50. We also set the size of the *3D* array to 4 GiB.

In Table II, we show the time spent by that our application blocking for I/O as a function of *I/O ratio* on Lustre, Burst Buffer, Data Elevator and MLBS. The benchmark's computation time of the array is not affected by the I/O ratio. In fact, it only depends in the number of time steps which we set to 3500 and it is measured to be $1800s$. When the *I/O ratio* is set to 1, the array is dumped to disk every iteration in the computation phase. In the processing phase these arrays are then read back from disk. When the ratio is 2, I/O happens at every 2 time steps, so on so forth.

We also measure the expected and actual wall time speedups, see Table III. At the start of the benchmark, a small IOR-like code is used to measure the systems combined write and read throughputs. It also measures the time spent for a handful of compute iterations and identify the expected time of computing all time steps. This information is fed into our model described in V-A to measure the expected wall time speedup. Actual speedup is calculated after the benchmark is done.

Thus, when we set the ratio to 1, the application ends up requesting I/O from Lustre PFS for about 95% of its wall time. When writing and reading directly to the Burst Buffer, however, the blocking time is reduced by 65% improving I/O block time by a $3.16X$ speedup. The overall speed up for the application is then measured to be $2.94X$ When using MLBS, the I/O blocking time is reduced by 72% and the wall time is improved by $3.62X$; see Table III.

The I/O times for the Burst Buffer and Data Elevator are almost identical. Data Elevator by design uses the Burst Buffer as a caching layer. It redirects all I/O requests from the Lustre to the Burst Buffer. Its independent parallel program then flushes data from the Burst Buffer to Lustre. However, unlike MLBS Helper engine that is carefully pinned at runtime, Data Elevator's threads are free to switch hardware threads which hinders the computation kernel of out micro-benchmark.

As also seen in Table II, an application instance on one compute node can generate up to 28 TiB of two-way I/O traffic, write and read. The aggregate size of all files per an application instance is 14 TiB of intermediate data. Since *Shaheen-2*'s Burst Buffer capacity is only 1.5 PiB, we would only be able to run 110 application instances at once. Such limitation not only delays scientific discoveries, it also denies other users who would like to use the Burst Buffer, as well. Therefore, MLBS restricts the size of the Burst Buffer allocation to 1 TiB. This forces files that are pushed from DRAM to Burst Buffer to be pushed then to Lustre in a FIFO order. This allows for more than 3500 instances of the application to run at the same time. It also gives room for other applications to use the Burst Buffer without disruptions.

C. Evaluation of I/O workloads

Vector Particle-In-Cell VPIC is an application that demonstrates write-only I/O access. At the end of each time step the application dumps to disk particle data in HDF5 format. We have integrated MLBS into VPIC and measured the time spent by 512 nodes to write a total of 100 TiB in 16 time steps on Lustre, DataWarp Data Elevator and MLBS. In Fig. 7 we see

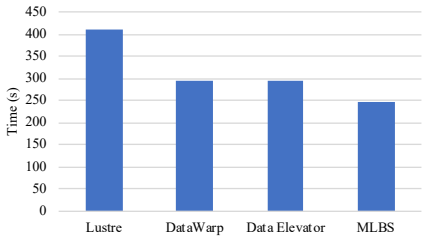

Fig. 7. The time spent by 512 nodes to write 100 TiB of data.

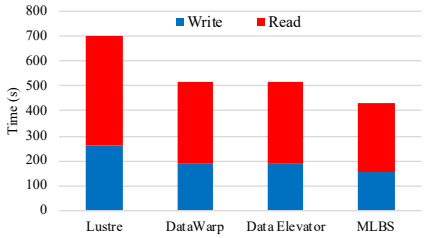

Fig. 8. The time spent by 512 nodes to read then write 64 TiB of data.

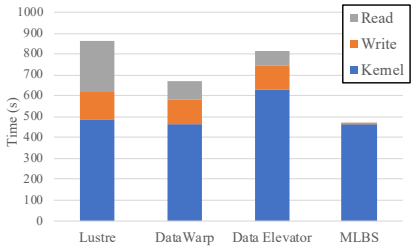

Fig. 9. Execution time breakdown for the RTM app on a 0.5 GiB grid.

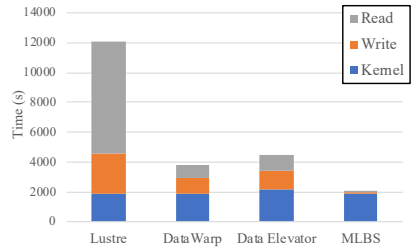

Fig. 10. Execution time breakdown for the RTM app on a 2.9 GiB grid.

that MLBS yields a 1.6X speedup, whereas DataWarp and Data Elevator achieve 1.3X I/O time speedup.

Hardware Accelerated Cosmology Code HACC is a read-after-write type to application. At each time step the application reads back data that is previously written using MPI-IO collective call. We have integrated MLBS into HACC and as seen in Fig. 8 we measured the time spent by 512 nodes to read then write a total of 64 TiB in 16 time steps using Lustre, DataWarp, Data Elevator, and MLBS. The buffering techniques in MLBS yield a 1.5X I/O speedup, which is not far away from 1.4X speedup on DataWarp and Data Elevator.

D. RTM

We evaluate the performance impact of MLBS on a real production code from the oil and gas industry. The code consists of forward modeling and backward propagation of *3D* wavefields. The RTM application processes a total of 338 shots on 338 compute nodes. Each shot computes for 5000 time steps and writes to disk 625 snapshots, i.e., an I/O ratio of 8. These snapshots are used in backward processing to produce the desired seismic image. In this paper the snapshots were not compressed. Compression, sometimes used in practice, swaps I/O time for additional processing time in both directions, at a cost in accuracy.

We evaluated the application with two different *3D* array sizes, a small one, $512 \times 512 \times 512$, and a larger one, $1400 \times 1400 \times 400$. With the small array the application generates a total of 48.7 TiB of intermediate data, with the larger one a total of 413.2 TiB is generated.

We also measured the performance of the RTM application on Lustre, Burst Buffer, Data Elevator, and MLBS. The reference RTM kernel used all 32 threads. All *3D* arrays are written or read either directly to Lustre or Burst Buffer, or through Data Elevator and MLBS.

Data Elevator uses separate I/O processes per node. While this process is important to redirect I/O request from Lustre to Burst Buffer, it cannot co-exist with RTMs computation threads. Therefore the computation time is increased.

We integrate our C++ MLBS implementation into the RTM application on *Shaheen-2* following Algorithm 3. We remove one thread from the RTM pool of OpenMP threads and reassign it to MLBS engine as a helper thread. This does not affect the performance of the computational kernel, since the stencil kernel is memory-bound, while fully utilizing the vector units for AVX. We test the performance impact when we allocate 70% of the physical DRAM capacity for MLBS, while we limit the capacity that is given to the Burst Buffer. Such a scenario is realistic when the total capacity of the Burst Buffer per run is limited. It also enables other users to utilize the Burst Buffer while our code is running.

In Fig. 9 and 10, for each I/O configuration, we break down the wall time of the execution to its kernel, `write()` and `read()` operations. On Lustre, we observe that 43% of the execution wall time is spent blocking for I/O on the small 0.5 GiB arrays and 84% on the larger 2.9 GiB arrays. Using the Burst Buffer and Data Elevator, I/O block time is reduced to 25% and 50%, respectively, on the small and large arrays. MLBS, with its careful thread affinity control and DRAM and Burst Buffer utilization reduces the I/O block time to less than %2, thus nearing a perfect I/O compute overlap.

In Table IV, we compare the expected and actual wall time speedups. We calculate the expected speedup at runtime as detailed in §V-A. Both Burst Buffer and Data Elevator reduce the time spend by application blocking for

Array Size	Burst Buffer		Data Elevator		MLBS	
	Expected	Actual	Expected	Actual	Expected	Actual
0.5 GiB	1.40	1.29	1.09	1.06	1.87	1.87
2.9 GiB	3.84	3.18	3.50	2.71	6.33	6.06

TABLE IV
COMPARISON OF EXPECTED AND SUSTAINED WALL TIME SPEEDUP OBSERVED ON THE RTM APPLICATION.

I/O. However, since Data Elevator misses with the affinity of the RTM computation threads, it causes an increase in the computation time; therefore, the benefits of the Data Elevator are reduced. This is clearly visible on expected and actual speedups reported on the table. MLBS carefully handles I/O requests while minimizing the impact on the computation kernel. The obtained results demonstrate the ability of MLBS in maximizing the throughput of DRAM and Burst Buffer, when Lustre is the last layer. Finally in Fig. 11 we show

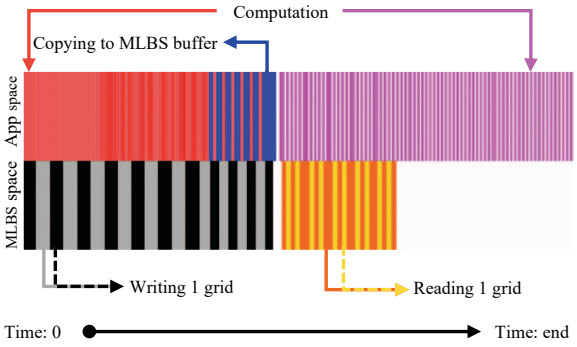

Fig. 11. Each red and pink bars represent iteration of 8 steps. Black and gray bars show overlapped write operation. Yellow and orange bars represent overlapped read operation.

an actual overlap between I/O and compute when integrating MLBS with RTM. The trace helps us to visualize how I/O is carried through in the MLBS space and shows the effectiveness of embedded helper engine. It also shows how the RTM stresses MLBS especially towards the end of the forward modeling drawing in red. Given these generic components, MLBS components may be further extended to support various applications, beyond the herein studied applications.

VI. Related Work

Many HPC centers have adopted multiple storage layers in their systems. Ranging from node-local to remote-shared NVMes and SSDs, these layers are helping to reduce the performance gap between I/O and compute operations. However, PFS are not equipped to manage the multi-layer nature of storage subsystems as they have traditionally managed each layer independently [13]. Lustre [10], GPFS [11] and PVFS [12] are used, for instance, for single layer disk-based storage systems. Therefore, many systems and libraries have been proposed to fully utilize all layers of storage subsystems. These systems can be classified into three categories: node-local, remote-shared or hierarchical caching systems.

Systems such as BurstFS and BurstMem [36], [37] were proposed to redirect I/O calls from PFS to local NVMe and SSDs. Similar work such as SSDalloc, NVMalloc and [38], [39] use memory mapping techniques to include local NVMe and SSDs in applications address space.

Burst Buffer caching systems such as DDN IME [15], DataWarp [16], and Data Elevator [17] accelerate applications writes by redirecting writes from PFS to remote-shared Burst Buffers. Helper processes of these systems then asynchronously flush data to the PFS. However, on read, users are forced to manually populate data into Burst Buffers.

Recent works such as Hermes [19], UniviStor [20] and ARCHIE [18] provide buffering solution on all stages of storage layers. Hermes offers three data placement policies to manage caching and prefetching of flat data files. UniviStor is implemented as an independent parallel program that runs along side application and manage their data movements. ARCHIE is a read-only system that provides multidimensional array-aware prefetching. Instead of prefetching entire flat files, it prefetches blocks of data.

In contrast to previous work, MLBS is a software-level library solution that consists in overlapping I/O and compute for out-of-core simulations. We provide a holistic solution to enhance the time-to-solution of real multi-threaded distributed-memory scientific applications using both write behind [40] and prefetching mechanisms. These optimizations permit to stream data across several stacked memory/storage layers (i.e., DRAM/Burst Buffer/Lustre), while maximizing the throughput at each encountered hardware layer. MLBS is able to properly balance the I/O and compute loads, with minimal and controlled impact on the application computation phases. Our caching engine ensures that all stacks of intermediate storage Are shared fairly among all other applications running without MLBS. The closest solution to MLBS is UniviStor. However, since its independent processes co-share compute nodes, it might decrease the computations performances of associated application and increases its time-to-solution.

VII. Conclusion

Out-of-core scientific simulations spend a considerable amount of time blocking for I/O. Advances in storage technology have made it possible to include heterogeneous multi-layered storage subsystems between main memory and PFS. Leading-edge large-scale production supercomputers are now typically equipped with node-local and remote-shared NVMes as well as SSD-based Burst Buffers. Existing solutions to manage storage subsystems 1) have unpredictable impact on the computation kernels by corrupting their cache locality and 2) deny other applications from fairly using remote-shared Burst Buffers. We have introduced a versatile software library, named MultiLayered Buffer Storage (MLBS), which overlaps expensive I/O operations with computations in out-of-core scientific simulations. MLBS asynchronously pushes and pulls data across hardware stacked storage layers, such as main memory/Burst Buffer/Lustre. Moreover, it ensures that computation kernels are not affected by helper cores,

and it also increases and optimize the occupancy on Burst Buffers. The resulting up and down pipelining of data between storage layers overlaps with the main computational kernel. We evaluated MLBS on the distributed-memory *Shaheen-2* Cray XC40 supercomputer. We show that MLBS permits the application to carry on as if it does not require out-of-core computations, while achieving up to 6.06X wall time speedup with Burst Buffer as an intermediate layer on large data files. In our future work we will consider hardware accelerators, which would require the support for an additional rather small memory layer (i.e., GPU's main memory) at the top of current hardware stacked storage layers.

ACKNOWLEDGMENT

For computer time, this research used the resources of the Supercomputing Laboratory at King Abdullah University of Science & Technology (KAUST) in Thuwal, Saudi Arabia.

REFERENCES

[1] B. Dong, X. Li, L. Xiao, and L. Ruan, "A new file-specific stripe size selection method for highly concurrent data access," in *Proceedings of the 2012 ACM/IEEE 13th International Conference on Grid Computing*. IEEE Computer Society, 2012, pp. 22–30.

[2] A. Shoshani and D. Rotem, *Scientific data management: challenges, technology, and deployment*. Chapman and Hall/CRC, 2009.

[3] A. Kumar, M. Boehm, and J. Yang, "Data management in machine learning: Challenges, techniques, and systems," in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 1717–1722.

[4] B. Xie, Y. Huang, J. S. Chase, J. Y. Choi, S. Klasky, J. Lofstead, and S. Oral, "Predicting output performance of a petascale supercomputer," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2017, pp. 181–192.

[5] W. W. Symes, "Reverse Time Migration with Optimal Checkpointing," *Geophysics*, vol. 72, no. 5, pp. SM213–SM221, 2007.

[6] D. A. Reed and J. Dongarra, "Exascale computing and big data," *Communications of the ACM*, vol. 58, no. 7, pp. 56–68, 2015.

[7] DC Report: SSD price premium over disk is falling. [Online]. Available: http://bit.ly/2LWJdKS

[8] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the role of burst buffers in leadership-class storage systems," in *IEEE 28th Symposium on Mass Storage Systems and Technologies (msst)*. IEEE, 2012, pp. 1–11.

[9] J. Bent, "Unraveling "Burst Buffer" tiers: A survey of the various instantiations," June, 2017, lustre User Group (LUG). [Online]. Available: https://bit.ly/2ZHv0Iu

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

[11] F. B. Schmuck and R. L. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters." in *FAST*, vol. 2, no. 19, 2002.

[12] R. B. Ross, R. Thakur *et al.*, "PVFS: A parallel file system for Linux clusters," in *Proceedings of the 4th annual Linux showcase and conference*, 2000, pp. 391–430.

[13] A. M. Caulfield, J. Coburn, T. Mollov, A. De, A. Akel, J. He, A. Jagatheesan, R. K. Gupta, A. Snavely, and S. Swanson, "Understanding the impact of emerging non-volatile memories on high-performance, io-intensive computing," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 2010, pp. 1–11.

[14] G. K. Lockwood, D. Hazen, Q. Koziol, R. Canon, K. Antypas, J. Balewski, N. Balthaser, W. Bhimji, J. Botts, J. Broughton *et al.*, "Storage 2020: A Vision for the Future of HPC Storage," 2017.

[15] DDN IME. [Online]. Available: https://www.ddn.com/products/ime-flash-native-data-cache/

[16] D. Henseler, B. Landsteiner, D. Petesch, C. Wright, and N. J. Wright, "Architecture and design of cray datawarp," *Cray User Group CUG*, 2016.

[17] B. Dong, S. Byna, K. Wu, H. Johansen, J. N. Johnson, N. Keen *et al.*, "Data elevator: Low-contention data movement in hierarchical storage system," in *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. IEEE, 2016, pp. 152–161.

[18] B. Dong, T. Wang, H. Tang, Q. Koziol, K. Wu, and S. Byna, "Archie: Data analysis acceleration with array caching in hierarchical storage," in *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 2018, pp. 211–220.

[19] A. Kougkas, H. Devarajan, and X.-H. Sun, "Hermes: a heterogeneous-aware multi-tiered distributed I/O buffering system," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2018, pp. 219–230.

[20] T. Wang, S. Byna, B. Dong, and H. Tang, "UniviStor: Integrated Hierarchical and Distributed Storage for HPC," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2018, pp. 134–144.

[21] A. Kougkas, H. Devarajan, X.-H. Sun, and J. Lofstead, "Harmonia: An Interference-Aware Dynamic I/O Scheduler for Shared Non-Volatile Burst Buffers," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2018, pp. 290–301.

[22] F. Vega. (2012, sep) Little's law: Isn't it a linear relationship? Accessed on Sept. 2019. [Online]. Available: http://www.vissinc.com/2012/09/07/littles-law-isnt-it-a-linear-relationship/

[23] High-performance storage list. Accessed on Sept. 2019. [Online]. Available: https://www.vi4io.org/

[24] D. R. Musser, G. J. Derge, and A. Saini, *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*, 3rd ed. Addison-Wesley Professional, 2009.

[25] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf, "Managing Variability in the IO Performance of Petascale Storage Systems," in *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2010, pp. 1–12.

[26] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with PAPI-C," in *Tools for High Performance Computing 2009*. Springer, 2010, pp. 157–173.

[27] J. McCalpin, "Memory Bandwidth and Machine Balance in High Performance Computers," *IEEE Technical Committee on Computer Architecture Newsletter*, pp. 19–25, December 1995.

[28] HPC IO Benchmark Repository. [Online]. Available: https://github.com/hpc/ior

[29] ExaHDF5 Parallel I/O Kernel (PIOK) Suite. [Online]. Available: https://sdm.lbl.gov/exahdf5/software.html

[30] Vector Particle-In-Cel. [Online]. Available: https://github.com/glennklockwood/vpic-io

[31] S. Habib, A. Pope, H. Finkel, N. Frontiere, K. Heitmann, D. Daniel, P. Fasel, V. Morozov, G. Zagaris, T. Peterka *et al.*, "HACC: Simulating sky surveys on state-of-the-art supercomputing architectures," *New Astronomy*, vol. 42, pp. 49–65, 2016.

[32] CORAL-benchmarks. [Online]. Available: https://asc.llnl.gov/CORAL-benchmarks/

[33] Hardware Accelerated Cosmology Code. [Online]. Available: https://github.com/glennklockwood/hacc-io

[34] P. Sava and S. Hill, "Overview and classification of wavefield seismic imaging methods," *The Leading Edge*, vol. 28, no. 2, pp. 170–183, 2009.

[35] E. Baysal, D. D. Kosloff, and J. W. Sherwood, "Reverse Time Migration," *Geophysics*, vol. 48, no. 11, pp. 1514–1524, 1983.

[36] T. Wang, K. Mohror, A. Moody, K. Sato, and W. Yu, "An ephemeral burst-buffer file system for scientific applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2016, p. 69.

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

[38] A. Badam and V. S. Pai, "SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011.

[39] C. Wang, S. S. Vazhkudai, X. Ma, F. Meng, Y. Kim, and C. Engelmann, "NVMalloc: Exposing an Aggregate SSD Store as a Memory Partition in Extreme-Scale Machines," in *Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 2012.

[40] J. Arulraj, M. Perron, and A. Pavlo, "Write-Behind Logging," vol. 10, no. 4. VLDB Endowment, Nov. 2016, pp. 337–348.