NVStream: Accelerating HPC Workflows with NVRAM-based Transport for Streaming Objects

Pradeep Fernando ‡ Ada Gavrilovska ‡ Sudarsun Kannan † Greg Eisenhauer ‡
‡ Georgia Tech, † University of Wisconsin-Madison

ABSTRACT

Nonvolatile memory technologies (NVRAM) with larger capacity relative to DRAM and faster persistence relative to block-based storage technologies are expected to play a crucial role in accelerating I/O performance for HPC scientific workflows. Typically, a scientific workflow includes a simulation process (producer of data) and an analytics application process (consumer of data) that stream, share, and exchange data supported by an underlying OS-level file system. However, using an OS-level file system for data sharing adds substantial software overheads due to frequent system calls, journaling (for crash-consistency) cost, and file-system metadata update cost. To overcome these challenges, we design NVStream– a lightweight user-level data management system that exploits NVRAMs byte addressability and fast persistence to support streaming I/O in scientific workflows. First, NVStream reduces I/O-related software overheads by designing a memory-based persistent object store and log-structured heap manager that exploit NVRAM's large capacity. Second, NVStream incorporates a hardware-assisted *non-temporal stores* for crash-consistent updates at near hardware data copy (memory copy) speeds. Finally, NVStream reduces data written to NVRAM with a delta compression, which further reduces I/O cost for workflows with higher write locality. The evaluation of NVStream using I/O benchmarks and scientific applications demonstrates 10× reduction in I/O compared to NVRAM-optimized file systems and also guaranteeing crash-consistent data movement.

HPDC '18, June 11–15, 2018, Tempe, AZ, USA

ACM ISBN 978-1-4503-5785-2/18/06…$15.00
https://doi.org/10.1145/3208040.3208061

CCS CONCEPTS

• **Information systems → Storage class memory**; • **Computer systems organization → Reliability**;

KEYWORDS

HPC I/O, NVM, streaming data, crash-consistent updates

ACM Reference Format:
Pradeep Fernando ‡ Ada Gavrilovska ‡ Sudarsun Kannan † Greg Eisenhauer ‡ ‡ Georgia Tech, † University of Wisconsin-Madison . 2018. NVStream: Accelerating HPC Workflows with NVRAM-based Transport for Streaming Objects. In *HPDC '18: The 27th International Symposium on High-Performance Parallel and Distributed Computing, June 11–15, 2018, Tempe, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3208040.3208061

1 INTRODUCTION

Long-running scientific computations, such as material combustion, fusion and climate modeling, periodically produce program outputs of the simulation state. These periodic program outputs serve multiple purposes. First, they are used as application checkpoints, which are used for recovery in the event of simulation or system failure. Second, they are increasingly being used to provide online insights into the simulation state, and are directly consumed by co-running coupled analytics programs, performing output visualization, verification, uncertainty analysis, or other data analysis tasks [22, 27] This producer-consumer relationship among application components establishes a streaming workflow, and HPC technologies supporting streaming workflows are becoming an integral part of HPC systems [9, 17, 18].

Although at a high-level, streaming workflows provide an opportunity for analytics application to quickly consume simulation data, the performance of the streaming infrastructure is highly dependent on the performance and capacity of the available memory and storage resources. This is because, in streaming workflows, in-memory buffers are used to temporarily stage simulation outputs that must be written to storage (e.g., for checkpointing), or for a co-running workflow component to consume the data. However, due to significantly large data volumes and the limited amount of DRAM capacity, the volume and frequency of data that can be generated and passed through the workflow components

is restricted. To overcome these challenges, production-level systems, such as ADIOS [18] use persistent storage (flash, hard-disk) devices and OS-level file systems for temporarily staging data, thereby overcoming the DRAM capacity limitations. Because, the data movement across workflows is used for both reliability (i.e., checkpointing) and for coupled analytics, it must be carried out without sacrificing reliability or consistency properties. As a result, these systems typically rely on the underlying file system crash-consistency and durability mechanisms such as journaling and logging.

However, using slower storage devices such as flash or hard disk to stage data as a replacement for DRAM can significantly increase data exchange cost in streaming workflow due to significantly higher latency and lower bandwidth compared to using DRAMs. Emerging non-volatile memory technologies such as 3D-Xpoint DIMMs [1] provide 100× faster read/writes and up to 10× higher bandwidth compared to flash memory, and can accelerate the data exchange performance. Unfortunately, naive use of NVRAM underneath file system stacks can substantially limit NVRAMs hardware benefits provided to streaming workflows. This is because traditional file system based solutions incur overheads in several ways, including high serialization (memory to block conversion) overheads, de-serialization overheads, and POSIX-based system call and journaling/logging costs in the data movement path, which are not necessary given the byte-addressable interface exposed by NVRAM. Even file systems specifically designed for NVRAM [10] continue to impose these undue software overheads. Furthermore, adherence to full file system semantics prevents software to take advantage of new NVRAM-centric architectural features, such as support for streaming, non-temporal writes [13], which are particularly useful for streaming workflows. Note that using memory copy-based operations with NVRAM [20] is also inadequate: First, staged data poses requirements for temporal durability (i.e., to provide guaranteed delivery). Second, for correctness, the persistent nature of NVRAM requires transactional mechanisms so that partial updates are not persisted and delivered to downstream consumers.

Based on these observations, we design and implement **NVStream**, a user-level transport for workflow coupling and high-performance data streaming via NVRAM. NVStream provides benefits by leveraging the memory-based nature of NVRAM, the streaming semantics and temporal durability requirements of scientific workflow systems, and the new architectural capabilities in modern processor architectures. Its design combines a streaming, versioned object-store, with efficient log-based memory management, and hardware-accelerated persistence for consumer-producer patterns. Additional optimizations in the data movement path for applications exhibiting temporal locality in their data access behavior are realized through use of delta encoding [5].

The outcome is a novel ***NVRAM-specific data transport*** for scientific workflows based on ***streaming persistence***. Using NVStream, HPC workflows on NVRAM-based platforms achieve benefits due to reduced software overheads in the I/O stacks in their data producer *and* consumer components, and due to the more efficient use of the NVRAM-based resource. This leads to opportunities for higher-volume data exchanges – i.e., more frequent, or with larger outputs – with lower data movement overheads, enabling a faster data path for coupled analytics pipelines, and faster time to scientific discovery.

In summary, the paper makes the following contributions:

- We provide an in-depth analysis of the characteristics of using NVRAM-based transport channels in streaming workflows. (§7)
- We design a novel solution for streaming persistence which accelerates NVRAM-based producer-consumer data exchanges. (§4)
- We design and develop NVStream, a new NVRAM-specific systems for coupling in-situ analytics in HPC systems based on streaming persistence. (§5)

The current NVStream implementation is focused on a single-node, multi-core, shared memory platform and this implementation is evaluated in an environment emulated NVRAM running several data-intensive benchmarks and HPC applications. The experimental results show that NVStream cuts down the object snapshot size by, ∼50% for GTC and as much as ∼99% for miniAMR. Furthermore the accelerated I/O data movement improves performance by $7 - 10x$ for applications such as CM1 and GTC.

2 BACKGROUND AND MOTIVATION

Continued growth of data requirements in HPC. Going towards exascale we expect orders of magnitude increase in HPC I/O, both in terms of output frequency and size. Next generation HPC engines, with order of magnitude increase in core counts, enable high resolution science simulations with larger data sizes [15]. In addition, the increased core counts will bring down the system-wide mean time between failures (MTBF), necessitating more frequent outputs of the application's checkpoint state [15].

Given the expected significant mismatch between the increase in computational density vs. I/O capability in future systems [15], one way to reduce the cross-node data movement demand is to enable in-situ processing of simulation outputs by services and analytics components of the end-to-end scientific workflow [27]. Purely DRAM-based data exchanges among workflow components are limiting. 1) DRAM

continues to be a critical resource in HPC systems, and is expected to become more critical as we move towards exascale. 2) DRAM is a volatile device and thus vulnerable to data loss during workflow/node restarts.

NVRAM as a data transport. The emergence of NVRAM technologies such as 3D-Xpoint [1], changes the design point for future exascale systems and their stacks. This new class of memory is: 1) fast, with read/write latencies that are 100x faster than SSDs and more comparable to DRAM (within 2-5x factor), providing for fast data movement in and out of the device; 2) dense, with an order of magnitude more memory capacity than DRAM, providing for a holding area of large amounts of I/O data produced by the applications; 3) byte addressable, i.e., accessible via a load/store interface that enables fine-grain data movement and flexible system software stacks; and 4) persistent, i.e., can reliably store critical HPC I/O data such as checkpoints over node restarts. Thus this new storage class memory is ideal for low-latency, persistent data buffering between workflow components. However, a mere hardware replacement won't yield the best performance of the device as the existing software stacks lack "NVRAM awareness" in their designs.

Low-latency NVRAM I/O. Recent research on storage stacks has explored various design points in implementing NVRAM aware system software. On one end of the spectrum, we have software stacks with file system APIs that treat the NVRAM as a block device [6, 8, 10, 24, 25]. While these optimized file-system implementations remove redundant system components such as the page-cache from the I/O path and fit well into the existing persistent storage I/O model, they still introduce unnecessary systems software overheads. This is because the use of the file system stack leads to long I/O data movement paths due to multiple software abstraction layers (e.g., in the VFS layer), and requires switching back and forth from user-space to the kernel during data movements. Alternatively, one can treat NVRAM as fast memory and access the device using familiar 'malloc' like APIs. Memory APIs avoid kernel interactions for the most part and expose the device's load/store interfaces directly to the userspace. However current 'malloc' APIs (e.g., `malloc`, `free`), are not sufficient for NVRAM data interactions as they lack the awareness for device persistence.

Costly NVRAM updates. Unlike the DRAM memory, the NVRAM memory retains stored data across application/node restarts. Thus, a node crash may catch an update to an NVRAM-resident multiword data-structure in an intermediate state, resulting in neither the original state of the data nor the final state. Hence, to realize a crash-consistent persistent state, atomic update semantics are used that guarantees all-or-nothing update guarantees. However, the modern processors only support word sized (64 bit on x86) atomic updates that deny hardware supported atomic updates for multiword

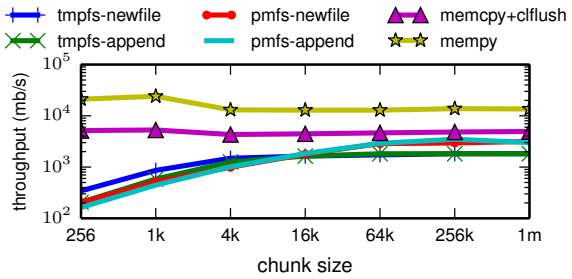

Figure 1: File systems designed specifically for memory devices such as tmpfs (volatile memory), PMFS (persistence memory) still yield lower throughput compares to crash-consistent memory copies across all data copy granularities. y-axis uses log-scale

data. System software techniques such as 1) write-ahead logging (WAL), 2) copy-on-write and 3) append-logging provide software-based atomic update primitives for such use-cases.

NVRAM-aware general purpose I/O storage stacks (file systems) such as `pmfs` optimize their crash-consistency routines around the byte-addressability of NVRAM devices. However general purpose I/O stacks miss the opportunity to incorporate application-specific optimizations for the same. For example, `pmfs` uses write-back mapped/processor-cached memory mappings for both data and meta-data and uses undo-based WAL for persistent meta-data updates; this is because, the former accelerates temporal data access to the file-system and the latter supports in-place updates of the meta-data structures – design decisions around two common usage patterns of file-system I/O. In contrast, in the context of HPC workflow I/O, data producers do not read back their output data (unless it is a restart) and they produce immutable output data in the form of application checkpoints. Thus crash-consistent semantics provided by existing NVRAM storage stacks introduce unnecessary I/O overheads due to lack of their ability to optimize application-specific I/O use cases.

Application awareness. Generic NVRAM storage I/O stacks, being loosely integrated into HPC applications, do not exploit optimization opportunities specific to an application. For example, some HPC applications do not modify every data grid point in each of their iterations. This can happen due to (but not limited to) application phases where data is sparsely modified (e.g., simulating a meteoroid traveling through space before hitting the ground) and program flow changes (e.g., in the GTC application checkpointing variables are selected based on its input configuration). A naive I/O stack, without application programmer's intervention, would end up moving unnecessary and redundant data to NVRAM, consuming both bandwidth and storage space of the device. A suitable HPC workflow transport can cut down the redundant data exchanges by leveraging its application awareness.

	tmpfs	PMFS	mmap	NVStream
No kernel crossings	X	X	✓	✓
Consistent updates	X	✓	X	✓
Synchronization	X	X	X	✓
Indexing	✓	✓	X	✓

Table 1: NVStream features, compared against some of the viable state-of-the-art system software stacks. Competing storage stacks lacks in features or has high software-overheads

Streaming writes. HPC program output I/O readily qualifies for a non-temporal store (streaming store) based output data copy. Streaming stores are a recently introduced architectural feature that bypasses the processor caches and sends writes directly to the memory device. This maximizes the data copy bandwidth by relaxing memory orderings enforced by the hardware. HPC I/O, often in the form of program snapshots, benefits from streaming stores. This is because 1) HPC program snapshots are rarely read back by the data producer and thus do not require temporal stores, 2) HPC program snapshots move large amounts of data on to the NVRAM devices, thereby benefiting from the high bandwidth data movement.

Opportunities for performance improvements. We compare three alternative NVRAM data movement techniques against streaming store-based I/O (streaming I/O) to understand the performance and feature gaps between them. The stock Linux `memcpy` implementation does not provide crash-consistency semantics but moves data using non-temporal store instructions whenever possible. We encode I/O data chunks as a new file (`tmpfs/pmfs-newfile`) or an append (`tmpfs/pmfs-append`) to an existing file. We use `memcpy` based NVRAM I/O to approximate the performance benefit of streaming I/O. We defer more detailed analysis to §7. `pmfs` [10] is a NVRAM optimized journaling file-system that provides both metadata and data crash-consistency using undo-logging and copy-on-write, respectively. `tmpfs` is a DRAM memory file system, that does not support any crash-consistency semantics – thus loosely corresponds to a best case file-system performance. The `memcpy+clflush` numbers refer to a memory based naive crash-consistent I/O mechanism that uses the `memcpy` APIs coupled with volatile cacheline flushes, while the vanilla `memcpy` approximates NVRAM I/O with streaming stores, albeit without crash-consistency.

Figure 1 shows that the streaming I/O (`memcpy`) outperforms `pmfs` across all data transfer granularities, and by as much as ∼ 100× at smaller chunk sizes. We attribute the lower data transfer throughput of both `tmpfs` and `pmfs` under small chunk-sizes to increased kernel crossings during data copying. While the streaming I/O outperforms all other I/O techniques, the mere introduction of crash-consistency in the form of cacheline flushes (`memcpy+clflush`) results in significant performance loss and cuts down the `memcpy` based I/O throughput by as much as ∼ 75% (still better than file-system throughput). Therefore, streaming I/O coupled with a light-weight crash-consistency mechanism would provide both high bandwidth and consistent data movement for persistent memory.

Furthermore, none of the above I/O stacks provide all the features required by an NVRAM-based HPC workflow transport. Table 1 evaluates each of the data copy techniques against a set of features required by a workflow transport. NVStream (NVS), presented in this paper, addresses all requirements and provides workflow components with fast, persistent memory-aware I/O transport.

In summary, *existing file system-based persistent storage stacks incur high overheads during persistent HPC I/O, both due to kernel intervention and generic crash-consistency semantics. While memory-based I/O APIs eliminate kernel overheads, supporting crash-consistency with memory-based I/O still is expensive. Therefore, careful adoption of modern hardware features and domain-specific optimizations creates opportunities (and is required) for creating a high performant NVRAM HPC I/O stack with minimal crash-consistency overhead.*

3 NVSTREAM- A NVRAM-BASED STREAMING OBJECT STORE

We design NVStream, a NVRAM-aware data transport for HPC workflow I/O. The main goals of the NVStream are:

- HPC data transport for node-local and cross-node workflow interactions;
- persistent and consistent simulation output state maintenance;
- NVRAM device-aware data transport design; and
- exploiting HPC application I/O awareness for high performant transport design.

We first briefly describe a high-level usage of the proposed system in the context of HPC workflow I/O, followed by a deep dive into each of the components of the system.

NVStream Overview. HPC workflow components, i.e., data producers and consumers, interact with the NVStream runtime using a set of I/O APIs. These APIs are modeled after familiar memory like API calls. We determined that the effort of porting a traditional file I/O-based application to NVStream is minimal. For an example, we ported the GTC and CM1 applications to NVStream with ∼ 15 and ∼ 85 number of source code line updates.

Figure 2 shows the control and data path interactions between workflow producers and consumers with the NVStream runtime. The application that generates data, such as the simulation code, first declares the output variables by allocating them using NVStream API call `nvs_create_obj`, as shown in Figure 3. The call registers the allocation metadata with the

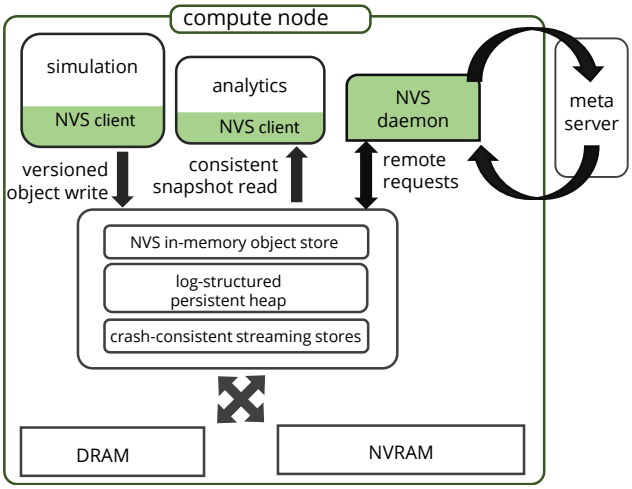

Figure 2: The high level component design of NVStream.

```
1  nvs_init(store_id); /* runtime init */
2  /* object allocation with the key = "zeta" */
3  void *obj = nvs_create_obj("zeta", sizeof(int)*1000);
4  int *array = (int *)obj;/* casting to our own type */
5  /*
6   * perform computation steps on this object
7   */
8  nvs_put("zeta"); /* output write */
9  nvs_free_obj{"zeta"};
10 nvs_finalize(); /* runtime shutdown */
```

Figure 3: Science simulation producers producing program snapshots using NVStream API

NVStream runtime and returns a contiguous volatile memory block similar to `malloc`. Later, the producer outputs a versioned snapshot of the object using `nvs_put`, which stores the snapshot durably on the underlying NVRAM-based shared memory layer. The consumer application in the workflow uses similar calls to retrieve consistent snapshot versions from the NVStream transport channel.

The major components in the NVStream runtime which support these interactions are:

Log-structured heap is a NVRAM-aware shared heap implementation, responsible for allocating space on the shared NVRAM memory and providing for crash-consistent data writes and reads. §4

Object-store is the application-facing component of NVStream. It exposes a set of APIs to both producers and consumers for workflow interaction. §5

NVStream daemon is responsible for background services such as garbage collection of old NVStream objects and support for remote/cross-node data transfers. §6

4 LOG-STRUCTURED MEMORY HEAP

4.1 Persistent Smart Pointers

NVStream I/O transport uses persistent shared memory to move data between producers and consumers in a coupled workflow. We manage persistent shared memory regions as named mappings (`mmap` call) supported by the OS. Each mapping is uniquely identified as a file name – we identify it as `map_id`. A process independent unique shared memory address is constructed using `<map_id :offset>` where `offset` denotes the number of byte offset value from the beginning of mapped segment's start address. Given a shared memory address, a process accessing persistent shared memory data, first maps the memory segment identified by `map_id` into its own address space and then records the starting virtual memory address of the mapped segment – `map_start`. Next, the process local virtual address is obtained by {`map_start` + `offset`}.

We need a similar addressing mechanism for structures placed on the volatile shared memory as well. We use boost [3] interprocess primitives for the same.

4.2 Shared Persistent Heap

NVStream stores the producer generated data and metadata on NVRAM before they are streamed to the workflow consumer/s. For efficient data updates/lookup, the data writes on persistent devices follows a pre-determined data storage layout/format. The system software stack operating on the persistence device often determines the specifics of the data layout. For example, `pmfs` formats the NVRAM device space with a B-tree index structure for efficient file-data traversal. While complex data layouts such as inodes and B-tree indexes improve on-device data organization and data traversal speeds, they often come at a cost in the form of excessive metadata overheads, costly crash-consistency routines, etc.

We learn that the simulation program snapshots, placed on the NVStream persistent memory regions are versioned/immutable objects. These immutable objects are consumed by one or more downstream analytics applications – often in a monotonically increasing object version order. The immutable objects exchanged between producers and consumers form a streaming data pattern and is well-supported by the FIFO data structure.

The above observation motivates us to format (see Figure 4) the NVStream persistent shared memory as a log-structured append-only storage heap (heap-log). The new data gets appended to the tail of the log (marked by `log_tail`) while old data gets taken out from the log-head (marked by `log_head`. The heap-log is generic enough that it does not enforce any layout on its append entries. The high-level software abstractions, such as the NVStream object-store,

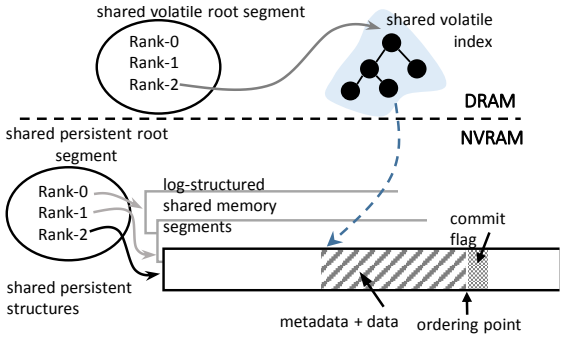

Figure 4: NVStream stores data in a shared memory log-structured heap formatted on NVRAM device and maintains volatile state of the runtime, in DRAM. The indexing structures are placed in shared volatile memory for multi-process accessibility

defines the specific entry format that describes the stored data.

In addition, we acknowledge the fact that almost all HPC applications follow single-program multiple-data compute pattern influenced by MPI parallelization framework and thus perform parallel I/O. We optimize the NVStream persistent heap for the same by maintaining per-process heap-log – supporting efficient parallel data writes. A root persistent object act as the entry point to the NVStream's storage hierarchy and is uniquely identified by a `store_name`. It keeps track of the heap-logs – identified by their `rank` id.

Our design choice of formatting NVRAM device as a log-structured heap greatly simplifies the crash-consistent data updates on NVRAM and more importantly, enables high throughput persistent data writes on to the NVRAM device.

4.3 Crash-Consistent Data Streaming

Although, the data stored on NVRAM device survives application and node restarts, maintaining metadata and data consistency of the stored data in the wake of unplanned crashes is a challenging problem. Current crash-consistent update techniques solve the problem by either atomically updating data or encoding enough information so that during recovery, one can reason about the consistent state. For designing crash consistent updates for NVStream's persistent heap-log, we make the following observations:

Observation-1 - the versioned objects and the append-only log-structured heap do not mutate existing data/metadata during output writes.

Observation-2 - data producing HPC applications do not read-back the written out data in the fast path – happens only during a restart.

Observation-1 leads us to incorporate a logging-based crash-consistent updates – a natural design choice for our log-structured heap. To that end, crash-consistent and durable log appends to the heap-log includes following steps: ① acquire the write lock of the heap-log, ② issue writes that comprise of the appending data, ③ ensure all the writes have been committed to NVRAM, ④ issue `commit` flag append to the heap-log, ⑤ ensure that `commit` flag write has also been committed to the NVRAM device.

The regular `temporal-store` instructions provided by x86 processors operate on write-back(WB) mapped memory segments. They are optimized for temporal data access patterns and are buffered in the processor cache hierarchy before being evicted to the backing memory (write-back operation). Therefore, temporal-stores, when used for heap-log appends, requires explicit cacheline flushing (using `clflush`, `clflushopt`) to move cached temporal stores into the backing memory device. Cacheline flushing is expensive and significantly reduces the memory parallelism and bandwidth usage due to the ordered execution of flush instructions. In contrast, x86 streaming stores operate on write-through(WT) mapped memory segments, and these writes bypass the processor cache altogether. In addition, the streaming stores enjoy a relaxed memory consistency semantics in contrast to total-store-ordering(TSO) memory consistency enforced on temporal-stores by the processor. Further, the absence of TSO and write-combining buffer support of CPU (batching) allows streaming stores to move data fast into the backing memory device. Interestingly enough, **Observation-2** suggests that temporal stores are of little use to HPC applications, and importantly, temporal store-based I/O may negatively contribute to the producers compute performance as they are likely to evict the actual application's (simulation's) working-set from the cache.

Based on these observations, we use streaming stores for NVStream's heap-log appends. We use store (`sfence`) instruction to enforce explicit ordering between streaming stores (e.g., steps ③ and ⑤ in the heap log append).

5 NVSTREAM OBJECT STORE

5.1 Object Allocation

Application code allocates objects using `nvs_create_obj` API call. The API call is similar to `malloc` memory allocation call, but accepts additional `object_key` parameter. The `object_key` serves two purposes, 1) allow subsequent NVStream APIs to use it as a volatile handle for the allocated object (e.g., `nvs_put` and `nvs_free`), 2) provide analytics applications a persistent object handle to retrieve NVRAM stored objects (e.g., `nvs_get`). The NVStream runtime in its internal representation maintains a fully qualified object handle in the form of [`store_id : object_key`] using the `store_id` given in `nvs_init`. Our prototype currently supports hierarchical `store_id`'s up to two levels– usually

in the form of [store_name/rank]. For an example, an object with key, 'zeta' that belongs to the MPI rank 1 of a coupled simulation named gtc-analyze is represented as [gtc-analyze/1:zeta].

NVStream object allocation involves allocating a memory chunk from the DRAM main memory (Linux glibc allocator) and updating the volatile index structure that maps the object_key to a memory address along with other metadata that includes object's size and the current version. The compute kernel uses the returned object for regular/volatile reads and writes. During program snapshots, the NVStream runtime generates a versioned copy of this object on the persistent memory. We describe this operation next.

5.2 Object Write

nvs_put call creates a versioned persistent snapshot of the volatile object identified by the object_key. The call implements the following control sequence. First, NVStream runtime scans its internal structures to find the object address for a given key which was mapped and recorded during the object allocation request. Next, it makes a snapshot of the DRAM resident object on persistent memory by appending/copying object data into the process local heap-log (see Figure 4) along with name, version, and other metadata information of the object. Finally, we update the DRAM resident indexing structures with the newly written object information. **Batched appends.** Most HPC applications perform I/O in batches; after a certain number of iterations, an application writes out all the intermediate objects/variables in a batch. We exploit this characteristic of HPC applications and introduce a batched object write API – nvs_snapshot. The nvs_snapshot call converts multi-object writes into one single append operation on the NVRAM based heap-log. Batching the writes eliminates multiple crash-consistent write sequences that involve costly store ordering, and increases the memory bus bandwidth usage during data movement.

Delta-compression. Matrix-related arithmetic is a norm in HPC applications, and checkpoint/analytics output state mostly comprise of these matrix variable states. However, we learn that some HPC applications do not modify all the data points in their application matrices/grids across iterations. This observation opens up an opportunity to selectively store the data that has been changed since the previous I/O step – storing only the delta from the previous state. This optimization is called **delta-compression** and the NVStream flavor that integrates the optimization as NVSD.

The proposed NVSD uses memory page-based write protection support available in the memory management unit (MMU) and the OS exposed system call interfaces to create lightweight deltas in the userspace. The NVSD approach calculates deltas at object granularity and we page-align the object allocations. During each compute iteration after nvs_put operation, the NVSD runtime write-protects all the pages of one or more DRAM resident objects. As a consequence, any subsequent write to the memory protected object raises a page protection signal (SIGSEG_FAULT) leading to the following control sequences: 1) NVSD registered signal handler gets invoked, 2) the runtime determines the memory page and the object that belongs to the faulting memory address, and finally, 3) the modified/written pages are recorded in a bit vector and the write protection on the faulted memory page is removed. We use the modified page bit vector to calculate the delta of that particular object and only store the modified memory pages in heap-log. We store additional metadata information such as the relative position of the modified pages within an object; this information is used for reconstructing the exact object sate at the analytics application (the consumer) as if there were no delta-compression on the stored data.

5.3 Object Read

The nvs_get call retrieves a NVRAM-resident persistent object and is internally implemented as follows. We first find the heap-log corresponding to the given [store_id:object_key] using the rank value in the store_id; if the rank belongs to local NVStream store, the requested object version is available on one of the heap-logs available on the local node/persistent device, else it has to be retrieved from a remote node. We look at each of those scenarios next.

Scenario1: *object is local.* We walk the rest of the volatile index structure in-search for object_key:version mapping. A successful search returns a persistent shared memory address (map_id : offset) of the heap-log stored object. The address uniquely identifies a persistent memory location in which the object resides using which we map the corresponding log-heap into our own address space and read the object data/metadata at offset.

Scenario2: *object is remote.* The NVStream daemon process is responsible for retrieving remote objects over the network. We place a remote object retrieval request on the NVStream daemon's §6 shared memory request queue and synchronously wait for the response.

6 NVSTREAM DAEMON

6.1 Remote Data

The NVStream daemon component is responsible for 1) serving requests for local objects from remote nodes across the network, 2) fetching the objects located in remote nodes for local analytics processes, and finally, 3) for services such as garbage collection of stale objects.

Remote requests. The NVStream daemon receives object requests from remote peer nodes through a TCP server socket. Similarly, it looks-up the requested object by scanning the shared memory volatile index structure similar to that of the `nvs_get`. After an object is located, the daemon process serves it to the remote peer.

Remote objects. The NVStream daemon receives the remote object retrieval requests from local analytics processes through a shared memory queue structure. First, it finds the remote peer node that hosts the requested object; note that the object-to-remote host mapping is done via a metadata server hosted at a known address and the metadata server lookup maps the `store_id:rank` portion of a `object_key` to a hosting server address. Next, we synchronously request the object/s from the remote host. Finally, we serve the retrieved object to the local analytics process. It is important to note that the 'remoteness' of the requested object is transparent to the analytics application and is completely handled by the NVStream library/runtime. We cache the `store_name:rank` to `host-address` mappings for future interactions.

6.2 Garbage Collection

NVStream garbage-collection (GC) service cleans up the old versions of the objects from the heap-log. NVStream implements a simple GC algorithm called `max_version_gc`; the `max_version_gc` garbage collector purges all the stale object versions except most recent version (object with `MAX` version number). The GC service monitors the head and tail values of the heap-log and triggers garbage collection routine if the log space is filled up to a threshold value, which is a configurable parameter.

The NVStream's garbage collection routine steps include 1) calculating the new head offset of persistent log heap that constitutes the most recent `MAX` versions of objects, and 2) truncating the persistent log up to the new log offset.

The integration of delta-compressed objects into NVStream increases the complexity of log truncation operation; this is because the recent delta-based object versions may depend on the older versions of the same object for valid reconstruction of their complete object state. Thus simply truncating heap-log may result in losing some of the data parts which are critical for reconstructing more recent object versions out from their deltas. During NVSD heap-log truncation, we create a full program checkpoint that includes the objects of version `MAX`, thereby effectively truncating the dependency list of object deltas at the `MAX` object version. We maintain the `MAX` versioned program checkpoint separate from the heap-log. We only have to maintain one program snapshot version (the most recent one) per heap-log for guaranteeing correct reconstruction of full object state.

6.3 Failure Recovery

Recovering from failure includes restoring consistent runtime/volatile and persistent state of the system. During recovery, the NVStream daemon §6 re-initializes the shared volatile memory structures that include synchronization structures and metadata indexes. Populating volatile indexes for persistent objects involves walking the heap-log from `log_head` to `log_tail` and updating volatile indexes corresponding to objects details found on the heap-log. Because of log-structured persistent heap design, the recovery steps of NVStream is almost trivial in contrast to WAL based crash-consistency designs, that involves replaying the undo/redo log for restoring the consistent persistent state.

7 EVALUATION

We evaluate NVStream against several real-world and proxy HPC applications, 1) GTC, 2) CM1, and 3) miniAMR. Our experiments are designed to answer the following questions:

- How NVStream compares to state-of-the-art NVRAM storage stacks for streaming data writes and reads?
- What is the effectiveness of NVStream's delta compression technique and its contribution to fast data movement?
- Does NVStream reduces overall application execution time in the context of workflows with analytics or checkpoint I/O?

7.1 Methodology and Benchmarks

Interconnect	Mellanox Infiniband
CPU core	Intel XeonE5 1.8GHz
CPU cores per node	80 cores over 4 dies
Main memory per node	500GB over 4 NUMA nodes

Table 2: Node configuration of the in-house Aries cluster

Because memory controller-based NVRAM is not commercially available, we emulate persistent NVRAM using memory regions mapped over a DAX-enabled file system (pmfs) similar to [14]. DAX-enabled file systems allow direct load/-store access to the underlying mapped memory (DRAM in this case), thus emulating the load/store interface of NVRAM. Next, write-through memory mappings are not available in userspace as the Linux OS always maps the memory segments as write-back memory; therefore we use streaming stores over WB memory similar to [23]. Further, NVRAM devices are expected to have low device bandwidth compared to DRAM main memory; thus the NVStream data movement time depends on the NVRAM device properties [11]. Our current NVRAM emulation platform assumes the device to have latency and bandwidth parameters similar to that of DRAM, and differentiate them only based on their data persistence support. We leave detailed sensitivity analysis of

NVStream against different NVRAM latency and bandwidth parameters for future work. Finally, all our experiments are performed on a local machine, Aries (see Table 2); a local testbed provides us with greater flexibility of software stack configurations (e.g., installing pmfs). Next, we briefly describe each of the main simulation applications – GTC, CM1, and miniAMR.

Gyrokinetic Toroidal Code (GTC) is a three-dimensional particle-in-cell application [12] used in micro-turbulence fusion device studies. The checkpoint data constitutes of 2D/3D arrays. We change the original input parameters 'npartdom', 'micell' and 'mecell' in constant factors to weak scale the workload size of the benchmark.

CM1 is a three-dimensional, non-hydrostatic, non-linear, time-dependent numerical model designed for idealized studies of atmospheric phenomena [2]. It is designed for studies of relatively small-scale processes in the Earth's atmosphere, such as thunderstorms. We change the input parameters `nx`, `ny` and `nz` to vary the workload size of the benchmark.

miniAMR applies a seven-point stencil calculation on a unit cube computational domain, which is divided into blocks. The blocks all have the same number of cells in each direction and communicate ghost values with neighboring blocks. With adaptive mesh refinement, the blocks can represent different levels of refinement in the larger mesh.

We use several alternative I/O mechanisms as comparison points for NVStream. We describe each of those I/O techniques next:

nvs: is the proposed NVStream implementation that uses persistent streaming writes for data movement.

nvs+delta: is the NVSD implementation that incorporates the delta compression technique into NVStream.

memcpy: is a best case user-space data movement mechanism for NVRAM which does not support crash-consistent data movements. We use the default `memcpy` routines supported by glibc.

tmpfs: is a volatile main memory-based file system which lacks crash-consistent data storage. The file system uses a ram-disk – a pseudo disk that uses main memory, as its backing device. tmpfs serves as a metric for the overheads involved in traditional file-system-based data movements, even in the absence of costly crash-consistency semantics. We store each of the variable states in a new file at the program checkpoint where the versioning information is encoded into the file name.

pmfs: is NVRAM aware file system that supports crash-consistent data storage. `pmfs` [10] internally treats NVRAM as a memory device underneath its POSIX file-system interface, thereby eliminating the page-cache and device driver layers. We emulate an NVRAM device for PMFS by reserving a portion of DRAM at the OS start-up [4]. We use the PMFS

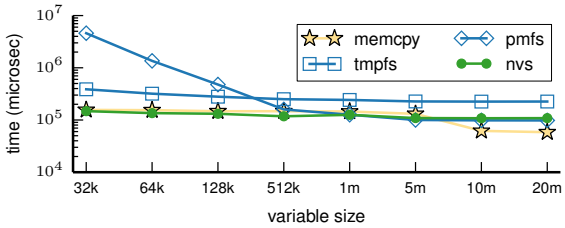

Figure 5: Checkpoint/analytics data write time against different storage stack transports, namely tmpfs, memcpy, pmfs and nvs. y-axis uses log-scale

implementation [19] ported for a newer Linux 4.x based kernel. Program snapshots are encoded into files similarly to `tmpfs`.

nocheckpoint: represents the application execution time without any checkpoint-related data movements representing the best case execution time.

7.2 Data Movement Latency

First, we evaluate NVStream against a home-brewed microbenchmark named NVSB. NVSB emulates an I/O dominant MPI-based compute kernel that periodically checkpoints its matrices after every configured number of iterations. We use NVSB to evaluate the sensitivity of each I/O strategy towards data movement granularity. To this end, we vary the variable sizes used in NVSB in each run, while keeping the total checkpoint data size constant. We run a single-threaded (one rank) benchmark instance with 400MB iteration checkpoint data size and plot the results in Figure 5.

NVStream outperforms file system-based I/O for all data granularities. The NVStream I/O is 2.6× faster compared to `tmpfs` and as much as 31× faster than `pmfs` I/O at 32KB variable size. It is important to note that `pmfs` performs as well as NVStream for large variable sizes (5M, 10M). However the I/O performance decreases exponentially as the variable sizes get smaller. We attribute this behavior to the crash-consistency overhead of `pmfs`. Smaller variable sizes increases the number of new file creation, therefore increase the file-system metadata manipulations. Metadata updates involve executing costly undo-logging based crash-consistency routines thus lowering the I/O throughput of `pmfs` at small I/O granularity. NVStream I/O performs equally well at every I/O data granularity. This is because NVStream uses data batching during checkpoints, effectively converting multiple I/O calls into one large I/O operation. NVStream performs as well as, or even slightly better than `memcpy` based I/O. This is because a `memcpy` call for every small variable prevents memcpy from utilizing the full memory bus bandwidth due to serialization. However for larger I/O sizes (e.g., 10MB), `memcpy` outperforms NVStream I/O by as much as 2×.

Next, we evaluate NVStream with the three HPC applications, GTC, CM1, and miniAMR. We configure the GTC test setup such that each rank outputs $\sim 210MB$ of I/O data

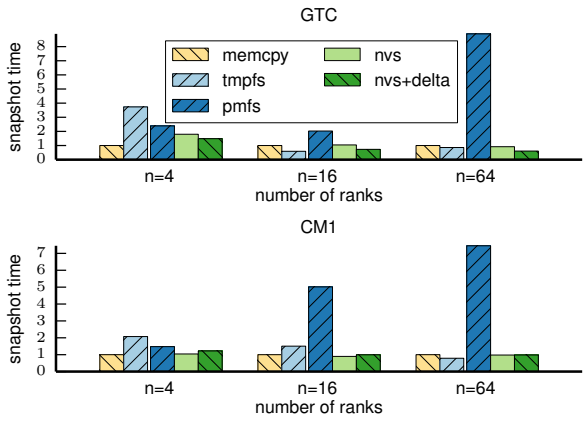

Figure 6: GTC and CM1 snapshot time for each of the I/O techniques. We normalize the times to best case data movement time – memcpy. We run each benchmark with increasing number of MPI ranks.

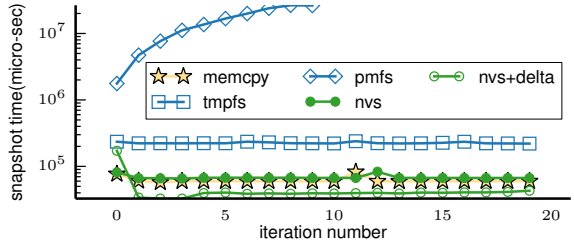

Figure 7: Program snapshot time of miniAMR plotted against checkpoint iteration number. y-axis uses log-scale

per iteration. We run the benchmark for increasing number of MPI ranks(N value) while keeping the per rank output data size more or less the same (weak scaling). We report in Figure 6 the average data I/O time for each I/O mechanism, normalized to `memcpy` I/O time. NVStream I/O is 24% faster than `pmfs` when 'N=4', whereas `pmfs` I/O performance drops drastically with the number of MPI ranks. As a result at 'N=64' NVStream I/O is 10× faster than `pmfs`. NVSD moves ~ 50% less I/O data compared to other I/O techniques and is 33% faster compared to NVStream I/O and `memcpy`.

Similarly, we configure the CM1 application to output ~ 45MB of I/O data per-iteration/per-rank. Unlike GTC, CM1 is a compute heavy HPC kernel, thus increasing the I/O data size leading to more time spent in the compute portions of the benchmark. We report in Figure 6 the average data I/O time for each I/O mechanism normalized to `memcpy` I/O time. NVStream I/O is 7× faster than `pmfs` and NVSD I/O performs the same. It is important to note that the performance of NVStream I/O is as fast as `memcpy` based I/O in most occasions, which confirms the lightweight crash-consistency semantics of NVStream.

miniAMR [21] does not have an inbuilt checkpoint routine, and therefore, we implement it by saving the current

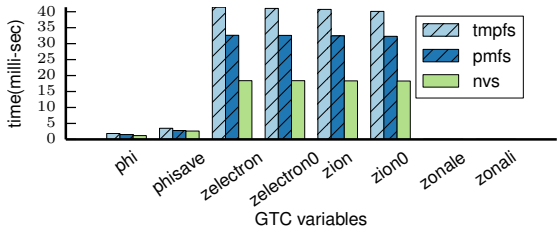

Figure 8: Data read time of the analytics kernel with different data transports

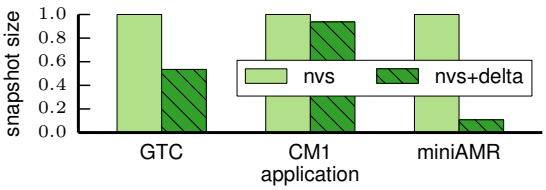

Figure 9: I/O data size reduction using delta-compression. We normalize the data size to nvs

mesh/blocks after each compute iteration. We use the application supplied sample workload *Two moving spheres* as our benchmark run. However the default max block size of 4000 results in a large number of variables and drives our `pmfs` setup to a non-responding state. We use 400 as the max number of blocks parameter and run the application with 4 ranks (see Figure 7). The `pmfs` based file I/O is 500× slower compared to `memcpy` at its worst-case data point. NVStream I/O performs within 11% margin of `memcpy` – the best case I/O mechanism, and NVSD outperforms the best case by 27%.

Next, we couple the GTC simulation output with a data compression analytics kernel [16] and evaluate the NVStream data read performance. Each rank of the analytics kernel reads the GTC variables in a monotonically increasing version order – the most common data read pattern found in the coupling workflows. We record the time spent on data reads in the analytics kernel under each of the I/O mechanisms and report the results (see Figure 8) normalized to `memcpy` based I/O reads. The analytics kernel runs with 16 MPI ranks where each rank consumes a GTC application output belonging to a single rank, thereby creating a one-to-one coupling testbed. We observe that NVStream reads are 43% faster compared to `pmfs`.

7.3 Delta-compression

Next we evaluate the ability of NVSD– the delta compression enabled NVStream– to reduce the total program snapshot size over simulation runs. The data size reduction will directly contribute to reduction in system interconnect bandwidth usage during data movements, data read time at the analytics application endpoints, and data movement across the compute nodes. In these experiments, we record the total program checkpointing size of each application with and without delta compression, shown in Figure 9.

In both GTC and CM1, we observe that the data access pattern across the iterations stays the same, keeping the output data amount at a fairly constant level across iterations and ranks. We report the average I/O reduction across compute ranks. NVSD reduces GTC program snapshots by as much as 47%. The biggest data reduction contributions are from variables that get conditionally activated during execution. The original GTC code selectively checkpoints objects based on the static configuration options, thus the checkpoint routine has complex conditional structures. NVStream eliminates the need for this application specific logic from the checkpoint routines, and delivers the same or better level of checkpoint performance without developer intervention. In contrast, CM1 allocates around ~80 variables per MPI rank, with a high modification factor among most of the variables. Therefore, NVSD yields only a marginal data reduction of 7% over NVStream.

miniAMR [21] differs from both GTC and CM1 as it refines the initial mesh during simulation (e.g., sphere moving through 3-dimensional mesh). The visual inspection of the application code reveals that it allocates its global mesh data structures during the application initialization, similarly to most other HPC applications. Our initial memory profiling on the application shows that the modification factor among large variables is very low. Our experiments further confirms our hypothesis. As a result, NVSD yields in data size reduction as much as 99% in some ranks.

7.4 Impact on Simulation Time

The ability of NVSD to compress output data comes at the cost of page-protecting and signal handling, which introduces additional compute overheads. To quantify this, we measure the effects of NVSD on application execution time. We highlight NVSD in this section because it alters the compute kernel's running time and tracks the data updates during the compute operation. The remaining I/O mechanisms, including NVStream, only get activated during the data write/read routines of the application. We measure the iteration time of each of the application and report the execution overhead against `nocheckpoint` I/O representing the best case execution time of the application.

The bandwidth of our emulated NVRAM device is same as the DRAM bandwidth of our testbed as we do not throttle the bandwidth of the emulated device. In addition, the compute requirements of GTC and CM1 are such that for smaller evaluation scale of the experiments, the data movement overhead of all NVRAM-based I/O mechanisms remains small in comparison to their compute phase. The `pmfs` based program snapshots incur 2% of execution overhead for GTC at 'N=64'. NVStream I/O contributes to similar overhead and NVSD increases the iteration time of GTC and CM1 by 2% and 0.5% respectively.

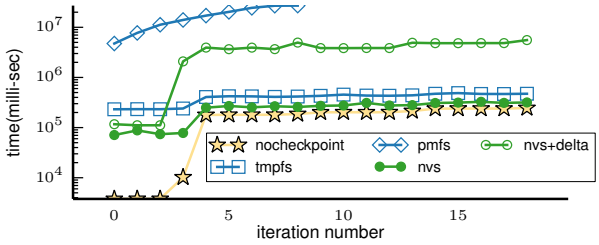

Figure 10: Iteration time cross section of the miniAMR against different snapshot schemes. Please note that y-axis uses log-scale

In contrast, the checkpoint routines of miniAMR are executed frequently enough to make the I/O overhead significant enough at the program scale used in our evaluation. With miniAMR, the `pmfs` based file I/O costs increase drastically (see Figure 10), incurring iteration overheads as much as 66× over `nocheckpoint` time. This is because, as we highlighted in §7.2, `pmfs` suffers high performance degradation due to frequent meta-data updates. However, NVStream checkpoints only incurs 47% overhead over `nocheckpoint` iteration time. Finally, NVSD increases the iteration miniAMR iteration time by as much as 20× at the benefit of significant reduction in checkpoint size as shown in Figure 9.

8 RELATED WORK

HPC I/O. The main simulation scientific applications are often coupled with analytics applications forming data workflows [18, 20]. The high data movement costs and increasing node-local processing power encourage in-situ analytics where simulation and analytics applications are co-run on the same physical machine. ADIOS [18] couples transport across simulation and analytics by using files or volatile memory buffers. However, it lacks support for crash-consistent updates. Next, MPI-IO supports parallel I/O API when a parallel file-system backs destination files. While NVStream does not offer a file-system API, the thread local log-structured memory in NVStream maximizes parallel I/O performance. Finally, TCASM [20] is a kernel based system software component that supports streaming workflow data management based on memory-mapped (mmap) interface. It is designed for volatile memory and lacks versioning support apart from immediate-next semantic.

NVRAM heaps. Mnemosyne [23] was one of the first systems to design NVRAM-based persistent memory programming. Mnemosyne implements persistent heap, log API for application logging, and persistent transactional memory. Mnemosyne heap does not support multi-process reads/writes and supports a generic word-based logging with additional torn write detection bits for every word for crash consistency. While it works well for frequent log updates, the detection bit adds significant overhead during large I/O

data writes. Another memory-based persistent library for NVRAM, NV-Heaps [7], supports object-based transactional memory as opposed to Mnemosyne's word-based support.

NVRAM file-systems. PMFS [10] is a direct-access file system (DAX) that removes the page-cache layer between the persistent memory device and the file-system API. It uses undo logging for maintaining metadata consistency and copy-on-write for data consistency. BPFS [8], another file-system optimized for NVRAM, uses copy-on-write to maintain its B+-tree based metadata structures for crash-consistency. NOVA [26], another NVRAM-based file system, provides better performance over PMFS using a log-structured design. However, NOVA also suffers from system call, metadata update, and garbage collection cost. However, we show in §2 that these systems perform poorly compared to a pure memory API based system.

9 CONCLUSION

The gap between compute and I/O speed is expected to widen in future HPC machines. In-situ analytics will enable data processing at the point of data origin, thus reducing the data movements across the network. Compute nodes with node-local NVRAM provides an opportunity to accelerate local HPC I/O; however the important question of how best to use NVRAM in the context of HPC I/O remains unclear.

In this paper, we propose NVStream, an NVRAM-based object store that provides streaming data transport to couple science simulations and analytic workflows. The NVStream system provides a lightweight and crash-consistent data sharing layer for streaming HPC I/O data. Furthermore, NVStream applies delta compression techniques before storing the data in the NVRAM streaming buffers, resulting in both fast data movement and efficient usage of NVRAM resource. Our evaluation of NVStream on several types of I/O and scientific benchmarks show up to 10X reduction in the I/O time.

Acknowledgment. We thank the anonymous reviewers for their helpful feedback. This work was supported by the Department of Energy, through the SSIO Unity project.

REFERENCES

[1] 2017. 3D XPoint. https://www.micron.com/products/advanced-solutions/3d-xpoint-technology. (2017).

[2] 2018. CM1. (2018). http://www2.mmm.ucar.edu/people/bryan/cm1/.

[3] 2018. Offset smart pointer for shared memory. https://www.boost.org. (2018).

[4] 2018. PMDK. https://pmem.io/2016/02/22/pm-emulation.html. (2018).

[5] Cristiana Amza and et al. 1996. Treadmarks: Shared memory computing on networks of workstations. *Computer* (1996).

[6] Adrian M. Caulfield and et al. 2010. Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories. In *MICRO*.

[7] Joel Coburn and et al. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *ASPLOS*.

[8] Jeremy Condit and et al. 2009. Better I/O Through Byte-addressable, Persistent Memory. In *SOSP*.

[9] Ciprian Docan and et al. 2012. Dataspaces: an interaction and coordination framework for coupled simulation workflows. *Cluster Computing* (2012).

[10] Subramanya R Dulloor and et al. 2014. System software for persistent memory. In *Eurosys*.

[11] Pradeep Fernando, Sudarsun Kannan, Ada Gavrilovska, and Karsten Schwan. 2016. Phoenix: Memory Speed HPC I/O with NVM. In *HiPC*.

[12] gtc 2017. GTC. http://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/gtc/. (2017).

[13] R Intel. 2007. and IA-32 architectures optimization reference manual. *Intel Corporation, May* (2007).

[14] Sudarsun Kannan, Ada Gavrilovska, Karsten Schwan, and Dejan Milojicic. 2013. Optimizing checkpoints using NVM as virtual memory. In *IPDPS*.

[15] Peter Kogge, Keren Bergman, Shekhar Borkar, et al. 2008. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems Peter Kogge, Editor. (2008).

[16] Sriram Lakshminarasimhan and et al. 2011. Compressing the incompressible with ISABELA: In-situ reduction of spatio-temporal data. In *European Conference on Parallel Processing*.

[17] Rob Latham. 2012. Parallel I/O in practice. http://www.nersc.gov/assets/Training/pio-in-practice-sc12.pdf. (2012).

[18] Jay F Lofstead and et al. 2008. Flexible io and integration for scientific codes through the adaptable io system (adios). In *Challenges of large applications in distributed environments*. ACM, 15–24.

[19] Sanketh Nalli and et al. 2017. An Analysis of Persistent Memory Use with WHISPER. In *ASPLOS*.

[20] Douglas Otstott and et al. 2014. Enabling composite applications through an asynchronous shared memory interface. In *Big Data*.

[21] Aparna Sasidharan and Marc Snir. 2016. *MiniAMR-A miniapp for Adaptive Mesh Refinement*. Technical Report.

[22] Christopher Sewell and et al. 2015. Large-scale compute-intensive analysis via a combined in-situ and co-scheduling workflow approach. In *SC*.

[23] Haris Volos and et al. 2011. Mnemosyne: Lightweight Persistent Memory. In *ASPLOS*.

[24] Haris Volos and et al. 2014. Aerie: Flexible File-system Interfaces to Storage-class Memory. In *Eurosys*.

[25] Xiaojian Wu and A. L. Narasimha Reddy. 2011. SCMFS: A File System for Storage Class Memory. In *SC*.

[26] Jian Xu and Steven Swanson. 2016. NOVA: a log-structured file system for hybrid volatile/non-volatile main memories. In *FAST*.

[27] Fang Zheng and et al. 2013. GoldRush: resource efficient in situ scientific data analytics using fine-grained interference aware execution. In *SC*.