Transitioning Scientific Applications to using Non-Volatile Memory for Resilience

Brandon Nesterenko
bnestere@uccs.edu
University of Colorado Colorado Springs
Colorado Springs, Colorado

Xiao Liu
xiszishu@ucsd.edu
University of California, San Diego
La Jolla, California

Qing Yi
qyi@uccs.edu
University of Colorado Colorado Springs
Colorado Springs, Colorado

Jishen Zhao
jzhao@eng.ucsd.edu
University of California, San Diego
La Jolla, California

Jiange Zhang
jzhang3@uccs.edu
University of Colorado Colorado Springs
Colorado Springs, Colorado

ABSTRACT

Scientific applications often run for long periods of time, and as a result, frequently save their internal states to storage media in cases of unexpected interruptions (e.g., hardware failures). Emerging non-volatile memory (NVRAM) can write up to 40× faster than traditional mechanical storage devices, providing an attractive medium for this purpose. This paper investigates the implications of transitioning a scientific application, Fluidanimate, to use NVRAM for fault tolerance. In particular, we evaluate the performance implications and ease-of-use of four fault-tolerance approaches: 1) logging through transactions, 2) multi-versioning through copy-on-write operations, and 3) checkpointing through IO operations (e.g., fwrite) on a direct access (DAX) filesystem and 4) checkpointing with a DRAM cache. Our study results in three key findings. First, additional changes to the application are required to take advantage of the increase in IO speed provided by NVRAM. Second, the performance scalability of the approaches lack when considering a single process. Third, NVRAM can increase reliability in a distributed computing environment by allowing individual nodes to error and automatically recover before the rest of the system notices.

CCS CONCEPTS

• **Software and its engineering → Software fault tolerance**; *Software performance*; • **Computing methodologies** → *Modeling and simulation*; • **Computer systems organization** → *Processors and memory architectures.*

KEYWORDS

fault tolerance, software performance, scientific applications

ACM Reference Format:
Brandon Nesterenko, Xiao Liu, Qing Yi, Jishen Zhao, and Jiange Zhang. 2019. Transitioning Scientific Applications to using Non-Volatile Memory for Resilience. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '19), September 30-October 3, 2019, Washington, DC, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3357526.3357563

ACKNOWLEDGMENTS

This paper is supported in part by NSF grants CCF-1421443, 1829524, 1829525, 1817077, and SRC/DARPA Center for Research on Intelligent Storage and Processing-in-memory.

1 INTRODUCTION

Scientific applications are often compute and data intensive, and as a result, frequently run for long periods of time. For example, simulations in computational fluid dynamics (CFD), a branch of study heavily used to model fluid flows, e.g, the air flow within various structures[19], can often take months to complete. On the other hand, as modern architectures evolve to include an increasingly larger number of cores to support larger scales of computing, hardware failures can frequently occur to disrupt the normal execution of long-running applications, requiring these applications to periodically save their internal states in persistent memories so that they can more readily recover from unexpected interruptions (e.g., hardware failures).

One of the biggest challenges faced by persisting scientific applications is the large amount of data to save, specifically to copy from DRAM to a more persistent storage such as hard drives or network-based storage, as the run-time cost of such data copying is often too steep for applications to use lightly, except when saving results from long important durations of computation. Emerging non-volatile memory (NVRAM) technology offers a persistent storage with a high capacity for storing large amounts of data and with access speeds comparable to that of DRAM (up to 40× faster than traditional mechanical storage devices), providing an attractive alternative to allow scientific applications to transparently survive the various hardware errors (e.g., a power failure).

Figure 1 compares these two approaches to further illustrate the potential benefit of replacing traditional hard drive with NVRAM for more efficient software persistence. Traditionally, illustrated in

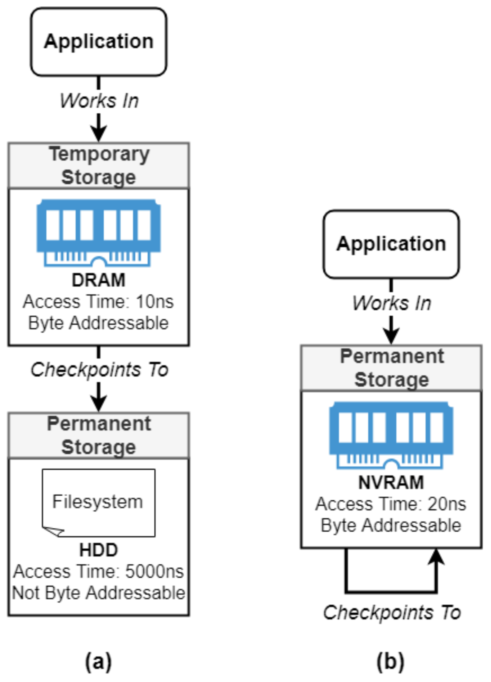

Figure 1: The comparison of saving an application state using (a) a traditional hard disk drive approach, and (b) a non-volatile RAM based approach.

Figure 1(a), software applications operate in DRAM as the computation requires byte-addressability and low access latency — typically at 10ns [13], where the access speed of RAM is the length of time it takes for data to be transferred to/from RAM to the CPU [6]. To persist the application's internal state, it must copy data from the DRAM to a hard disk drive (HDD), which has a typical access latency of 5,000ns [6]. The access speed of an HDD is the average length of time it takes to transfer data to/from the disk drive [6]. Alternatively, as shown in Figure 1(b), an application can operate directly within NVRAM because it is byte-addressable and can work similarly to DRAM. Data persistence requires less runtime overhead because NVRAM has a typical access latency of only 20ns [22].

This paper investigates using NVRAM to support transitioning scientific applications for transparent fault tolerance of hardware failures. Using a scientific benchmark, Fluidanimate, from PARSEC [16], we perform a case study to evaluate the performance implications of using four different approaches: 1) logging through transactions, 2) multi-versioning through copy-on-write operations 3) checkpointing through IO operations (e.g., fwrite) on a direct access (DAX) filesystem and 4) checkpointing with a DRAM cache, to periodically save the internal data of the application so that it can transparently recover from any hardware failures.

Our main technical contributions include:

- We study the performance implications of using NVM to support instant fault tolerance and recovery for a scientific application, Fluidanimate from the PARSEC 3.0 [16] benchmark suite. We observe that a rudimentary transition to using NVM to support the fault tolerance of applications does not necessarily lead to any significant increase in IO throughput (2.25×). We show that additional data structure changes are necessary to make full use of the NVRAM's performance advantages.
- We compare the four different NVRAM fault tolerance techniques: 1) logging using transactions, 2) multi-versioning through copy-on-write operations, 3) checkpointing through IO operations on a direct access (DAX) filesystem, and 4) checkpointing with a DRAM cache. We show an inverse relationship between the complexity to implement an approach and the overhead incurred to perform fault tolerance related tasks. Additionally, we show that when using only a single process, the performance scalability of the application can significantly decrease when periodically saving the application internal data.
- We study the effectiveness of using the various NVM persistence techniques to support transparent error recovery in a distributed multi-process setting that uses a heartbeat mechanism [54] to support error detection. Our results show that NVRAM based fault tolerance can increase reliability in a distributed computing environment by allowing individual nodes to error and automatically restore such that no other nodes in the system notice the error.

The remainder of this paper is organized as follows. Section 2 introduces background of using NVRAM to support fault tolerance in scientific applications. Section 3 describes how we have integrated the use of NVRAM into a scientific application. Section 3 presents our experimental study and discusses the implications of using NVRAM to support fault tolerant applications. Section 5 discusses related works. Finally Section 6 concludes the paper.

2 BACKGROUND

A large class of scientific applications seek to model or simulate real world phenomenon as a function of time. This process typically consists of three steps. First, a physical problem is discretized into an initial state, which typically represents a set of particles or bodies as a mesh. Second, the state is progressed through various circumstances, e.g. forces or temperatures, via iterating through an outer time loop. Lastly, the application ends when the state converges within a desired threshold[57].

From a storage perspective, scientific applications often require immense amounts of data to be stored in memory. The high memory demand also makes these applications good candidates for distributed computing. In particular, a memory grid can be created that spans across multiple compute nodes, where each node stores a subset of the discretized mesh. Additionally, each node is responsible for operating on its subset of the mesh, typically through performing multi-threaded calculations. It is therefore important to maintain scalability both at the distributed multi-process level, as well as on a single node at the multi-thread level. In this paper, multi-threading refers to the practice of mapping a computation onto multiple threads, the smallest unit of execution on a shared-memory computer, owned by a single process [1]. Distributed computing refers to the practice of leveraging multiple different computer nodes on a network, without any sharing of memory [14].

As modern architectures evolve and distributed computing capabilities increase, hardware failures are expected to happen more frequently [29]. This places fault tolerance as an important aspect of scientific applications, as it will allow them to recover more quickly from these failures. Non-volatile memory is an emerging technology with the potential to improve the time to recover these applications after hardware errors occur. The remainder of this section provides background for our study of using NVRAM to support fault tolerance in scientific applications. We first summarize existing approaches to support fault tolerance. We then elaborate on the various aspects of incorporating NVRAM into a scientific application.

2.1 Supporting Fault Tolerance

Traditionally, fault tolerance is supported by directly saving the internal state of an application to a disk drive through system support for checkpointing, logging, or copy-on-write operations. These methods, however, present large difficulties when scaling to the demands of large scientific applications due to the immense overhead induced.

Checkpointing is one of the most prominent fault tolerance techniques for scientific workloads [53]. This method designates certain areas of code as checkpoints to save the application's data into permanent storage; the program can reload this data anytime later to resume from this checkpoint after a crash. This method typically imposes blocking [25], where the normal execution of the process is stopped by both saving or loading all data into or from permanent storage, respectively.

Logging maintains a collection of idempotent operations that, in order to revert to a previous program state, need to be un-done, in the case of an undo log, or re-done, in the case of a redo log. The logging process is performed via four states: *steady*, *active*, *ready*, and *abort*. A process initially starts in the *steady* state, which indicates that a process has not logged any operations in the log. When an operation has been added to the log, the process goes into the *active* state, indicating it is currently modifying data and logging all updates. While in the *active* state, the process can change into either the *abort* state to indicate it will reject the current changes in the log file, or the *ready* state to indicate it will accept the changes. To reject the changes, the operations in the log file are applied in backwards order, the log file is deleted, and the state is set back to *steady*. To accept the changes, the log file is deleted and the state is set back to *steady* [66]. When considering blocking time, operations are added individually to the log, reducing an up-front block time to save an applications state; however, when restoring data, the application blocks to apply all operations.

Copy-on-write creates a copy of the data, or a *shadow copy*, to be modified on writes, and replaces all references of the original copy to the shadow copy once all modifications are complete [67]. This mechanism allows for fast recovery, as the versions of data are typically maintained via links via *fopen* [9] and *unlink* [4], where an application state can be quickly restored by linking to a previous version, or a snapshot. Copy-on-write allows for the ability to snapshot data at multiple levels of granularity, where further optimization can create shadow copies at the page level to reduce overhead [46, 67, 72].

2.2 Using NVRAM In Scientific Computing

Non-volatile RAM (NVRAM) is an emerging storage technology that presents a promising opportunity to mitigate fault tolerant induced overhead. With the requirement of guaranteeing its data being retrievable during a power outage, NVRAM can be supported via a number of technologies, including Phase Change Memory (PCM), Spin-Transfer Torque magnetic Memory (STT-RAM), and Resistive Memory (ReRAM). For example, PCM uses the two states (crystalline and amorphous) of the chalcogenide material to represent 0 and 1. Similar in structure to DRAM, all types of NVRAM are byte-addressable and can be accessed through the memory bus as a DIMM (dual in-line memory module) [42]. NVRAM technologies can not only offer performance comparable in latency and bandwidth to that of the DRAM, it can also achieve high storage density, which makes terabyte size memory feasible for a single computational node [58]. As Intel and Micron have released 3DXpoint as a commercial product, NVRAM technology is ready to fill the gap between DRAM and SSD [49].

Despite data on NVRAM surviving power failures, systems that uses NVRAM as main memory do not naturally guarantee data persistence, which requires that all the data must be recoverable to a point in time before a power outage. For example, the system may use volatile memories such as SRAM as a transparent cache in the memory hierarchy, and as the SRAM cache may reorder write and read operations as performance optimizations, at the time of a power outage, part of a program's data may potentially still reside in the SRAM cache while the rest have been written to the NVRAM, resulting in inconsistency. To provide better data persistence support, the x86 instruction set architecture (ISA) offers hardware-level instructions, e.g., (*mfence* and *sfence*), and write back instructions, e.g., (*wbinvd* and *clflush*), to allow applications to force ordering of stores to specific memory locations and thus guarantee the stores reach NVRAM after these instructions are successfully evaluated. Intel also expanded the ISA and added *clwb* to allow cache line write-back without invalidation [2]. Existing research also proposed multiple architectural designs to optimize the ordering and write-back process to better support persistent memory [41, 45, 51], including various designs based on logging [12, 35, 47, 55], checkpointing [59] and shadow paging [24, 60]. To support more efficient use of NVRAM, the Linux operating system has added direct access (DAX) features to allow applications bypass all the system buffers for the file system, specifically ext4, xfs, and brtfs[10]. When NVRAM is mounted with DAX file system, using NVRAM as a memory mapped device can fully explore its performance.

NVRAM brings opportunities to the scientific application from several perspectives, including its high storage density, which allows larger memory space, and its memory bus accessibility, which provides more flexibility than disk's I/O connection. In contrast to traditional DRAM memory, which supports as large as 128GB per DIMM [11], NVRAM is able to support 512GB per DIMM, making terabyte level memory feasible. In contrast to saving an application on an SSD or HDD, which requires using file-system commands to access and save files, NVRAM offers the ability for applications to leverage byte-addressability. For example, applications can retain pointers in NVRAM when saving/restoring application state,

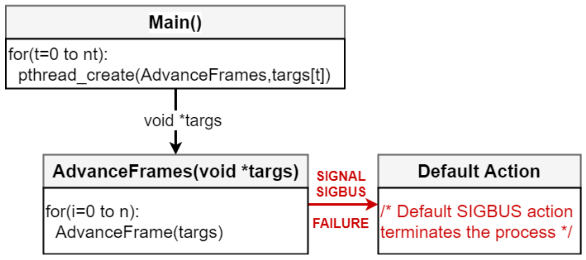
Main()
for(t=0 to nt):
pthread_create(AdvanceFrames,targs[t])
void *targs
AdvanceFrames(void *targs)
for(i=0 to n):
AdvanceFrame(targs)
SIGNAL SIGBUS
FAILURE
Default Action
/* Default SIGBUS action terminates the process */

Figure 2: Original workflow of Fluidanimate.

This allows applications to more readily persist large complex data structures by storing the entire working set directly into NVRAM.

3 MODIFYING SCIENTIFIC APPLICATIONS TO USE NVRAM

A number of different methods can be used to incorporate NVRAM-based fault tolerance into an application. To investigate the pros and cons of each, we have manually modified Fluidanimate, a computational fluid dynamics (CFD) benchmark from the PARSEC 3.0 [16], to use four of these methods: logging through transactions, multi-versioning through copy-on-write operations, checkpointing through file IO operations, and checkpointing by using DRAM as a cache. Because the Fluidanimate benchmark is data-intensive, the main challenge in making it fault tolerant is how to efficiently save and restore the large amount of data [33]. The computation structure of the application is illustrated in Figure 2, which is typical of many scientific applications (e.g. finite difference method and finite element method based applications [57]). Therefore, we expect our observations in making Fluidanimate fault tolerant can similarly apply to a large number of other scientific applications.

The following first describes how we modified the Fluidanimate application to support fault tolerance. We then compares four different approaches we have adopted to use NVRAM to save and restore data for the application. Finally, we discuss how to extend the capabilities enabled by our modifications to support instant recovery of individual processes in a distributed memory environment.

3.1 Modifying Fluidanimate

Shown in Figure 2, the original code structure of Fluidanimate consists of two levels of control-flow. The first level uses a loop to spawn a reconfigurable number (*nt*) of threads in the *main* function. The second level uses a loop to perform the actual CFD computation for each thread. The *AdvanceFrames* function serves as the entry point for each thread. The original code does not guarantee correctness against any failure, and therefore the process terminates itself as default action upon detecting a SIGBUS signal.

To support fault tolerance, the following pieces of information from the original code of Fluidanimate need to be saved periodically to NVRAM to allow instant restart on memory errors.

(1) *thread_args* stands for arguments specific to each thread. The arguments are stored as an array, where each thread is allocated an element in the array to store its thread-specific information, represented by the *targs* parameter in *AdvanceFrames(void *targs)*.

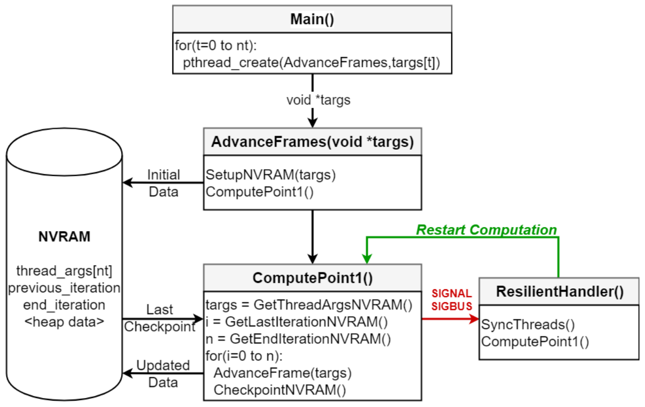

Figure 3: Restructured workflow of Fluidanimate.

(2) *previous_iteration* represents the previous iteration completed in the computation loop. (*i* in the for loop of *AdvanceFrames*)

(3) *final_iteration* stands for the end condition to quit the computation loop (*n* in the for loop of *AdvanceFrames*).

(4) *heap_data* represents the data allocated in the heap used in the computation of *AdvanceFrame*().

Figure 3 shows our modified code structure of Fluidanimate. First, we add *NVRAM*, a new shared memory region, to save the aforementioned pieces of information periodically as the application progresses. Second, we modify the *AdvanceFrames* function to store its thread specific arguments into NVRAM before invoking the actual computation. Third, we assign a new function – *ComputePoint*1 – for the computation. We make the *ComputePoint*1 function parameterless so it can automatically restore the application to the latest version of data in NVRAM before continuing. More specifically, this function first fetches all the needed data from the NVRAM region and then adapts the computation defined in the original code; this procedure ensures that the internal results after each iteration of the outermost loop are periodically saved into NVRAM before the next iteration. Finally, we introduce a SIGBUS signal handler – *ResilientHandler* – to deal with any detected memory errors. The handler synchronizes all threads within the process to ensure all of them are aware of the error, and each thread will then restart itself by invoking the *ComputePoint*1() function.

3.2 Using NVRAM

```
int vals[32];
vals[3]=5;
```

Listing 1: Unmodified code version to be extended with other NVRAM implementation strategies

We have explored four different APIs to support writing data into the NVM abstraction in Figure 3 as well as correctly extracting the previously written data. The following discusses each API in more depth, while using a simplistic example in listing 1 to illustrate the technical details. The pros and cons of each API are then discussed.

3.2.1 *Transaction-based Logging.*

The Intel Persistent Memory Development Kit (PMDK) [12] provides a C++ API that supports an application to group its instructions that modify NVRAM into *transactions*, where if the application successfully reached beyond the end of a transaction, all instructions inside the transaction are guaranteed to have been carried out correctly; on the other hand, if the application execution breaks down before reaching the end of the transaction, all modifications to NVRAM will be aborted, and NVRAM is guaranteed to return to its previous state before the beginning of the transaction. Internally, PMDK uses an object pool data structure to record the location and type of each variable in the NVRAM. The object pool is structured as a tree, where the root node is used as an initial reference point that all other NVRAM variables must attach. Inside a transaction, PMDK uses logging to record which variables are modified so that all modifications can be cohesively applied together.

```
 1: struct _root_data_t { int vals[32]; };
 2: PMEMobjpool *pool = pmemobj_open(
 3:       "/mnt/nvm/pool", ...);
 4: TOID(struct _root_data_t) root = POBJ_ROOT(pool,
 5:       struct _root_data_t);
 6: TX_BEGIN(pool) {
 7:       pmemobj_tx_add_range(root.oid, 0,
 8:        sizeof(int * 32));
 9:       int *my_vals = D_RW(root)->vals;
10:       my_vals[3] = 5;
11: } TX_END
```

Listing 2: Extending listing 1 with PMDK transaction API.

To illustrate, Listing 2 shows the result of extending the code from listing 1 to use Intel PMDK transactions. Here line 1 defines the layout for the root node in the object pool, located in a file-system directory that is mapped to a chunk of NVRAM address space. The memory pool is then initialized at lines 2-5, with the root object initialized at lines 4-5. The transaction starts at line 6 and ends at line 11. Inside the transaction, Line 7 invokes a function *pememobj_tx_add_range* to notify the library which area of memory will be modified during this transaction, before lines 9-10 proceeding with the actual read and write operations.

3.2.2 Multi-versioning. PMDK[12] also provides another interface to support atomic operations on ranges of memory. In this interface, the persisted data is stored in the same object pool structure as the transaction-based approach. We use this interface to create a multi-versioning approach to extend Fluidanimate with fault tolerance by associating these memory ranges with snapshots of the application's state. In particular, we create two versions: one labeled as the working version, which the application uses during computation, and the other labeled as the shadow version, which is used as the snapshot. Certain points in the code are designated to back up the data from the working version into the shadow version. To recover state, the application needs only to switch the labels on the two versions, where it can immediately start using the snapshot as the working set, as the byte-addressability of NVRAM allows pointers to stay consistent across the two versions.

```
 1: struct _version_t { int[32] vals; };
 2: struct _root_data_t {
 3:       TOID(struct _version_t) v1;
 4:       TOID(struct _version_t) v2;
 5: };
 6: PMEMobjpool *pool = pmemobj_open(
 7:       "/mnt/nvm/pool", ...);
 8: TOID(struct _root_data_t) root = POBJ_ROOT(pool,
 9:       struct _root_data_t);
10: TOID(struct _version_t) cur_v =
11:       D_RW(root)->v1;
12: int *my_vals = D_RW(cur_v)->vals;
13: my_vals[3] = 5;
14: pmemobj_memcpy_persist(pool,
15:       D_RW(D_RW(root)->v2)->vals,
16:       D_RW(D_RW(root)->v1)->vals,
17:        sizeof(int)*32);
```

Listing 3: Extending listing 1 with PMDK Atomic API.

To illustrate, Listing 3 shows the result of extending the code in listing 1 to use the PMDK atomic API. Here Line 1 defines a new data structure to hold all data that will be persisted in a single version. Lines 2-5 define the root data structure that holds the two versions. The persistent memory object pool and its root data are initialized at Lines 6-9. Lines 10-13 then perform the variable modification to the current version ($v1$), which is then propagated into the next version ($v2$) at Lines 14-17 by invoking *pmemobj_memcpy_persist*, which guarantees that either all of the data are copied correctly, or none is carried out, so that if an error occurred, the application data are simply collectively reverted to the previous version.

3.2.3 File System API. Here application checkpointing is supported through IO operations (e.g. fwrite/fread) of a Direct Access (DAX) file system. DAX bypasses the operating system's block layer to access the underlying NVRAM directly. By mapping NVRAM sections onto a DAX filesystem, an application can use fwrite/fread operations to directly store and retrieve data periodically into NVRAM. This API allows the application to operate within DRAM, while checkpointing data to files backed by NVRAM. This strategy, however, requires the developer to organize application data into pre-defined continuous region of checkpoint data, without taking advantage of the byte-addressability property of NVRAM.

```
1: int vals[32];
2: vals[3]=5;
3: fwrite(&vals, sizeof(int), FILE_SIZE_INT, file);
```

Listing 4: Code sample extending code from listing 1 with Filesystem checkpoint call.

To illustrate, Listing 4 extends the code from listing 1 to use the filesystem API. Here only a single line is appended at the end of the original code, so that the $fwrite()$ operation is invoked to write the data directly into a file.

3.2.4 Using DRAM as Cache[39]. Here the idea is to provide a hybrid DRAM/NVRAM approach, where DRAM acts as a cache in front of NVRAM. NVRAM then acts as a fast disk, where the checkpoint from DRAM to NVRAM is done in a similar manner as the file system API. We extend the interface from [39], which provides a similar hybrid DRAM/NVRAM checkpointing library, to support the storage and restoration of pointer data structures, where our API is shown in table 1.

```
1: int vals[32];
2: ckpt_context *ctx = init_ckpt_context();
3: reg_region(ctx, &vals[0], sizeof(int) * 32);
4: vals[3] = 5;
5: do_checkpoint(ctx);
```

Listing 5: Extending listing 1 with our DRAM cache API

To illustrate, listing 5 extends the code from listing 1 to use DRAM as cache for NVRAM via our API. Here line 2 is inserted into the original code to create a checkpoint context, ctx, for the

Function	Description
ckpt_ctx* init_backup_context()	Allocates a new checkpoint context, which maintains the locations of memory regions and pointers to be checkpointed
void register_region(ctx, dram_base, size)	Register a new memory region from within DRAM at *dram_base* with *size* to store into the checkpoint context *ctx*
void register_pointer(ctx, from, to)	Register a pointer into the checkpoint context *ctx*, where both *from* and *to* are addresses that exist in DRAM
void do_backup(ctx)	Performs the backup of the data in DRAM maintained by the checkpoint context *ctx* into NVRAM
ckpt_ctx* do_restore()	Restores the persisted data from NVRAM into DRAM and returns a checkpoint context for future checkpointing demands

Table 1: Our DRAM cache API extending the interface from [39].

application, which will keep track of memory regions and pointers to checkpoint. Line 3 registers the *vals* array as a new memory region to be checkpointed in the context of *ctx*. Line 5 performs the checkpoint of the data in DRAM maintained by *ctx*, the *vals* array, into NVRAM.

3.2.5 Comparing The APIs. Table 2 compares varying aspects of these APIs, including 1) the location of the working set, 2) which information is saved on a checkpoint, 3) which information is restored on a restart, and 4) summary comments to highlight the advantages or disadvantages of a particular strategy.

Working Set. A working set in NVRAM induces more overhead than DRAM, as the interface has slightly more overhead to access this data, and the access time is slightly larger. The logging and multi-versioning approaches have their working set in NVRAM and the filesystem and DRAM cache version have their working set in DRAM.

Checkpointed Info. The checkpointed information consists of all information saved when saving an application state. All approaches must to store the data required for computation in the checkpoint file; however, depending on how the checkpoint is created, pointers do not have to be persisted. The filesystem approach read/writes data structures in a particular order and does not need to persist pointers, whereas all other approaches do. This provides the filesystem approach with the ability to incur less checkpointing overhead as it saves less data.

Restored Info. The restored info consists of all information copied back into the working set of an application in order to get it to load the state at the last taken checkpoint. The logging and DRAM cache approaches copy back all data and pointers and induce the most overhead. The multi-versioning approach only reverts pointers to link to the previous snapshot version, and incurs the least overhead. The filesystem approach only reverts the data, and has overhead in-between the other approaches.

Summary Comments. To summarize each NVRAM approach, the logging and DRAM cache approaches are the easiest to incorporate, but incur the most data transfer overhead when saving and restoring data. The multi-versioning approach has the fastest restoration time at the cost of it being harder to implement into an application. The filesystem approach has the least amount of data storage overhead incurred at the cost of it being the hardest to implement.

3.3 Using NVRAM In Distributed Computing

Most scientific applications need to use multiple processes that are distributed across different nodes connected by networks of varying speeds. In this distributed setting, when any of the processes encounters an irrecoverable error, the other processes will be affected and need to recover collectively. Therefore, if a process can self-recover from a disruptive error, it is imperative that the recovery is sufficiently fast so that the other processes will not notice the lapse and enter into recovery mode unnecessarily. Therefore, we investigate whether using NVRAM can support the instant recovery of each process so that the overall application execution is not disrupted by minor soft errors encountered by individual processes.

A common method for multi-node systems to track errors is to employ the heartbeat mechanism [54]; it is a designated process (the heartbeat monitor) that tests the well-being of the application by periodically sending a communication to all live processes, and specifying a timeout for each process to respond before the whole application is plumped into error mode.

Figure 4 shows two additional modifications to our fault-tolerant version of Fluidanimate in Figure 3 to use NVRAM in the distributed computing setting. First, we add a new standalone process, the *heartbeat monitor*, which uses MPI to indefinitely wait to receive messages from nodes with a preconfigured timeout. Second, we extend the computation loop in *ComputePoint*1() to signal a heartbeat message via MPI after every iteration. If the heartbeat monitor does not receive a message before its timeout expires, the node is considered dead. The goal of this modification is to study whether our integration of NVRAM into the Fluidanimate application can allow the computation to be distributed into multiple nodes, with each node offering instant recovery capability such that minor soft errors (e.g., memory corruption) will not disrupt the overall progression of the distributed application.

4 EXPERIMENTAL STUDY

By modifying Fluidanimate to use NVRAM to support fault tolerance, the purpose of this study is to answer the following questions.

Approach	Working Set Stored In	Checkpointed Info	Restored Info	Summary Comments
Logging	NVRAM	Data + Pointers	Data + Pointers	High Overhead, Easy to Incorporate
Multi-versioning	NVRAM	Data + Pointers	Pointers Only	Fast Restore, Harder to Incorporate
Filesystem	DRAM	Data Only	Data Only	Customizable, Hardest to Incorporate
DRAM Cache	DRAM	Data + Pointers	Data + Pointers	High Overhead, Easy to Incorporate

Table 2: Comparing different APIs to use NVRAM.

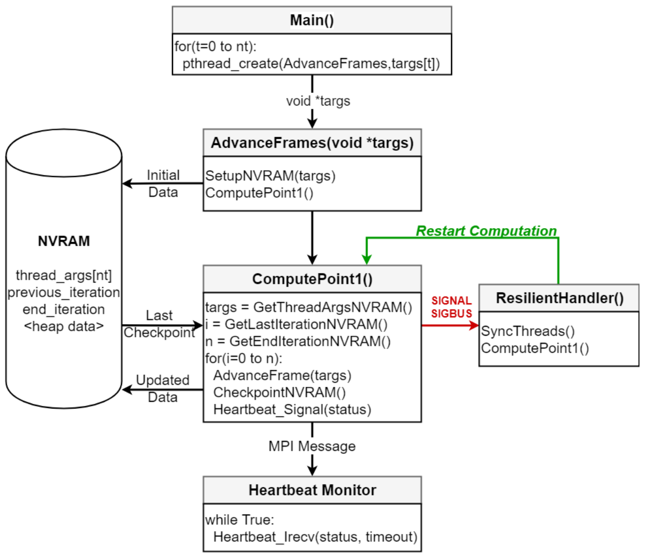

Figure 4: Our heartbeat monitor extension to the fault-tolerant version of Fluidanimate in Figure 3.

- **What is the complexity of modifying an existing scientific application to use NVRAM to support resiliency against hardware failures?** Our experience shows that modifying Fluidanimate is relatively straightforward, as long as a relatively high level API is provided to consistently store and extract application data into NVRAM.
- **What is the runtime overhead incurred by the different APIs of using NVRAM?** We have experimented with four of these APIs, summarized in Section 3.2. Our experimental results show that the overhead incurred by using these APIs vary significantly, and generally higher-level interfaces also are associated with higher overhead.
- **Does the speed advantage of NVRAM over traditional hard drive translate immediately to the shorter time required when saving and extracting application data?** Our answer to this question is no. In particular, we found that the memory layout of the data being saved and extracted play a significant role, to the extent that unless the data layout is re-organized, sometimes no performance benefit may be gained by using NVRAM over the traditional hard drive.

CPU		Intel(R) Xeon(R) CPU E5-2420 1.90 GHz, 12 Cores
Cache	L1-Data	32 KBytes
	L1-Instruction	32 KBytes
	L2-Private	256 KBytes
	L3-Shared	15360 KBytes
Main Memory		16 GiB
Operating System		CentOS 6.6

Table 3: Machine configuration.

- **Is NVRAM based technology sufficient to support resilient scientific applications, in the sense that if an application process encounters any hardware-level failure, the process can self-recover fast enough so that other dependent processes won't notice the failure?** To this end, we have studied the performance scalability of using the varying NVRAM technologies in the scientific application in both multi-thread and multi-process settings. We show that when using a heartbeat mechanism to monitor distributed processes, existing NVRAM technologies are sufficient to serve this purpose.

We performed our experiments on a machine with an Intel E5-2420 processor with 12 cores running CentOS 6.6. Table 3 shows the full machine configuration. The use of NVRAM on this machine was supported by DRAM emulation, by extending the Linux kernel 4.6.0 in CentOS 6.6 with ext4 file system with DAX enabled, configured by following the pmem guide [30]. The modified Fluidanimate application is evaluated by using its given native input (the input with the largest size). The application was compiled using g++ with the -O3 compiler optimization flag. The timings were recorded by inserting calls to the C++ chrono utilities library [7] inside the source code. The measurements are the average of a total of four runs, with an average standard deviation of 2.3%.

4.1 Runtime Overhead of NVRAM APIs

To compare the runtime overhead incurred by the various APIs of using NVRAM, specifically logging, multi-versioning, filesystem, and DRAM cache; we measure the time it takes to perform three actions: 1) store the application state once, *checkpoint* time; 2) restore the application state after a memory error occurs, *restore* time, and 3) compute a single iteration in the fluid dynamics main loop, *computation* time. To measure these actions, we place timing calls before and after the appropriate code section in the source code. The baseline computation time of a single iteration in Fluidanimate without any fault tolerance support was 770ms.

Table 4 reports the runtime overhead of using each NVRAM API to restart, compute, and save the result of a single iteration of the top-level computation in Fluidanimate. Here the transactional API,

Approach	Complexity	Checkpoint Time (ms)	Restore Time (ms)	Computation Time (ms)
Logging	Low	896	1472	887
Multi-versioning	High	860	1	792
Filesystem	High	241	192	803
DRAM Cache	Low	1230	643	775

Table 4: Comparing the NVRAM-based resiliency techniques by complexity of implementation vs performance (baseline computation time = 770ms)

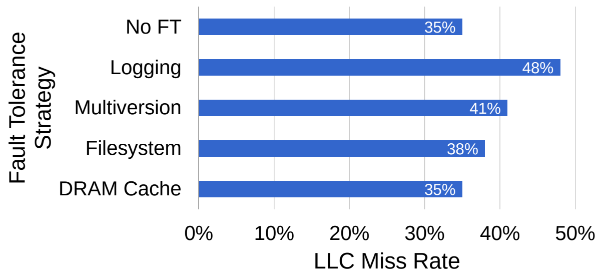

Figure 5: Cache miss rate of Fluidanimate when run under each fault tolerant implementation compared to the original version with no fault tolerance.

one of the least complicated to use, resulted in the high checkpoint and recovery times, at 896ms and 1.5s, respectively. On the other hand, the multi-versioning API, which is more complicated to use, although had a high checkpoint time, at 860ms, had the fastest recovery time, at 1ms. The file-system API, which also has a high implementation complexity, offers low checkpoint and restore time at 241ms and 192ms, respectively. Finally, the DRAM cache API, one of the easiest to use, also resulted in low performance, with 1.2s and 643ms checkpoint and recovery times, respectively. With the exception of the logging API, the internal compute time of the application did not change significantly.

To explain the variation in computation time using different strategies, we use the last level cache (LLC) miss ratio, as shown in Figure 5, where these values were measured using Linux Perf [3]. The miss ratio for the original version with no fault tolerance ($NoFT$) was 35%, where the miss ratios were 48%, 41%, 38%, and 35% for the logging, multi-versioning, filesystem, and DRAM cache fault tolerance strategies. The logging API, which added a 15% overhead into the application computation time, also added 13% additional cache misses into the application run-time.

4.2 Performance Advantage of Using NVRAM

In this study, we investigate whether the advantages of NVRAM over traditional magnetic storage systems are directly translated to higher performance (lower runtime overhead) in supporting fault tolerance in real applications. For this purpose, we compared the performance of saving and recovering data for Fluidanimate using our filesystem API for NVRAM, vs. using the same API but saving and recovering data using three different storage systems: an Ext4 hard disk drive (HDD), an emulated NVRAM disk file system, and a DRAM *tmp* file system. Figure 6 compares the maximum

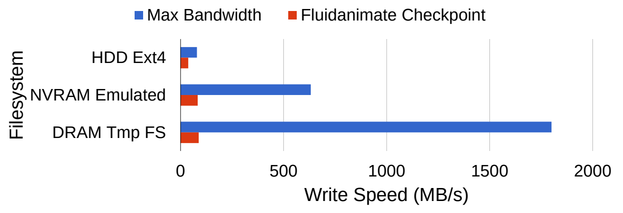

Figure 6: The bandwidth observed when saving/recovering data for Fluidanimate compared to the maximum bandwidth attainable for each storage system.

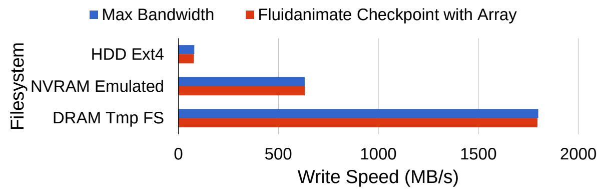

Figure 7: The bandwidth observed when checkpointing the modified version of Fluidanimate that uses arrays when compared against the maximum bandwidth attainable for a filesystem.

bandwidth attained by using each storage system, benchmarked via *dd* [8]. When using the HDD, the attained throughput is 37.2 MB/s, 46% of its maximum throughput of 79.7 MB/s. However, when using our emulated NVRAM, the attained throughput is 82.8MB/s, only 13% of its maximum bandwidth of 632 MB/s. Taking the matter further, using the DRAM *tmp* filesystem only attained 88 MB/s, 4.8% of its maximum bandwidth of 1800 MB/s. Overall, using NVRAM or even DRAM to replace a magnetic storage system resulted in only 2.3x speedup in the time required to save/recover data. in spite of a maximum speedup of 8x. So in conclusion, the speed advantage of NVRAM and its byte addressability does not necessarily translate to performance savings when used in real-world applications.

When investigating why Fluidanimate cannot fully take advantage of NVRAM, we discovered that the low NVRAM usage efficiency was because the use of linked data structures to store data in the application, so that when saving and recovering data, each individual element of a linked list must be saved and recovered separately. This incurs significant overhead within the CPU, which has to issue and wait for each write and read operation. To address the problem, we replaced the linked list data structure with an array-based implementation. The resulting application is then

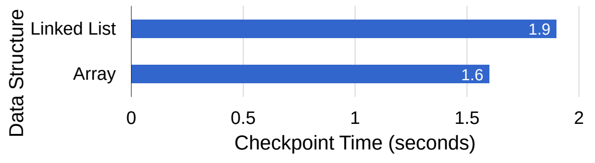

Figure 8: The duration of time spent in saving/recovering data in Fluidanimate when using a linked list vs an array.

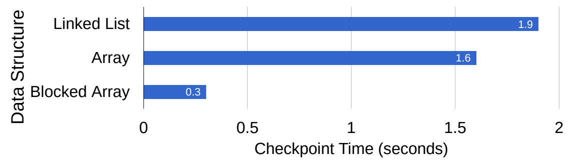

Figure 9: Extending Figure 8 to show the duration of time taken to checkpoint Fluidanimate with a blocked array.

able to achieve the maximum bandwidth when using each of the storage systems, shown in Figure 7. However, when comparing the total run time of the two versions of the application, one using an array to save its internal data while the other using a linked list, the throughput increases do not translate necessarily to application speed. Shown in Figure 8, when using arrays, the overall time spent in the application in saving and recovery data decreased only by 19%, in spite of the throughput increase of 7.6x. This discrepancy is due to a much larger amount of data required to be managed by the array-based version of the application, due to the simplistic data structure transformation from the linked list to the array-based implementation. In particular, the size of data saved by the linked list was 225 MB, whereas the size of data from the array was 4.8 GB.

In summary, we observe that the layout of data in the original application plays an important role in the overall efficiency of the application in using fast storage systems such as NVRAM (or DRAM). To further validate this conclusion, we created a third data structure, which organizes the elements of an array into smaller chunks to reduce waste of space while still providing high bandwidth. The new result of application performance is shown in Figure 9, where the chunk size was 2MB. Here the application using the blocked array to save data only took 0.3 seconds in saving and recovering data from NVRAM, providing a 6.3x reduction in time.

4.3 Performance Scalability

Scientific applications place a high priority on performance scalability. However, when supporting resilience against hardware failures, a performance overhead must be paid to periodically save their data to allow instant recovery. Figure 10 compares the performance of the original Fluidanimate application when using 1, 2, 4, and 8 threads with that after the application is made resilient by using each of the NVRAM APIs we selected. Here the original implementation ran at 2.06, 2.76, 4.79, and 6.98 GFLOPS when running with 1, 2, 4, and 8 threads, respectively, showing close-to-linear scalability. However, when using the NVRAM transactional API, the performance dropped to 1.18, 1.82, 2.17, and 2.47 GFLOPS. The

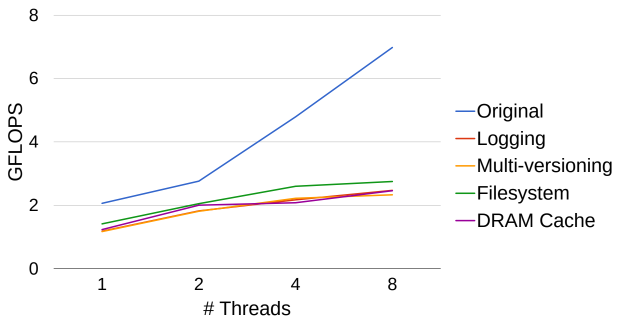

Figure 10: Performance scalability of the application when using each NVRAM API to support resiliency.

performance of the application when the other APIs are similar (multi-versioning at 1.17, 1.81, 2.22, and 2.33 GFLOPS; file-system at 1.41, 2.05, 2.6, and 2.75 GFLOPS; DRAM API at 1.23, 2, 2.08, and 2.46 GFLOPS). This significant performance degradation is because the application is only allowed to use a single thread when saving its data to NVRAM. So while the computation phase of Fluidanimate allows all threads to work concurrently, once the computation phase ends and the checkpointing phase begins, only one thread is active. and the other threads have to wait. Therefore, existing APIs need to be extended to supporting concurrent read/write operations to be used in large-scale scientific applications without significantly compromising their scalability.

4.4 Supporting Instant Recovery

As large scientific applications often need to be distributed among multiple nodes, a resilient application needs to be comprised of processes that can quickly self-recover from hardware failures, so that failures in an individual process does not translate to the whole application entering into error-recovery mode. Section 3.3 presents our extension of Fluidanimate to use a heartbeat mechanism to monitor the well-being of its processes. According to Noor [54], the heartbeat timeout, $T_{timeout}$, can be decomposed via the following equation

$$T_{timeout} = T_{hm} + T_i + T_r + T_w \tag{1}$$

Here T_{hm} is the time for a message to reach the heartbeat monitor and is typically set based on the underlying network configuration, T_i is the interval between sending heartbeat messages, T_r is the time for the heartbeat monitor to realize it hasn't received a message from a node, and T_w is the wait time for the heartbeat monitor to declare a node is dead. We defined the values in the heartbeat timeout equation in Equation 1 as follows.

- $T_{hm} = 0ms$, because we run the heartbeat monitor on the same machine as the Fluidanimate compute node, so no network connection exists.
- $T_i = 770ms$ because a single iteration takes 770 ms in the original version of Fluidanimate, and our modified application sends a heartbeat message at each iteration of the computation.
- $T_w = 770ms$ and is set to be the same value as T_i, as done in [54].
- $T_r = 154ms$ and is set to be $\frac{1}{5} * T_i$, as done in [54].

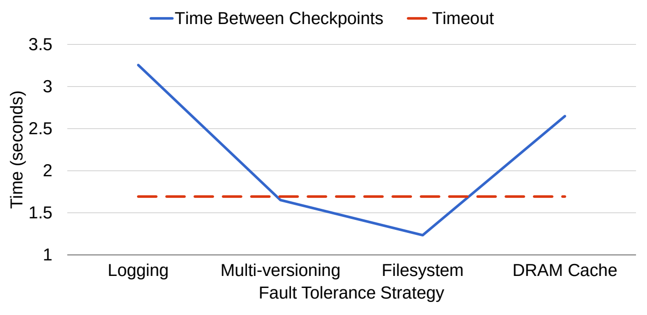

Figure 11: Whether the four NVRAM APIs allows instant restart within the heartbeat timeout (goal, 1.7s).

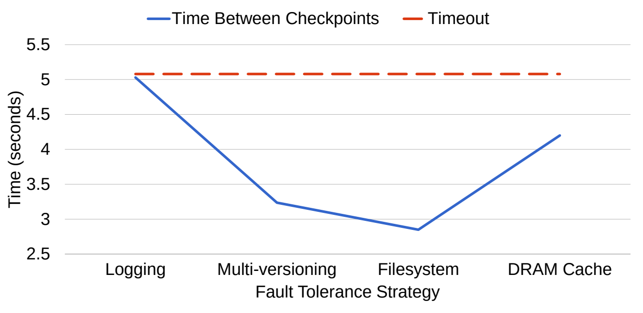

Figure 12: Whether the four NVRAM APIs allow instant restart with the modified heartbeat timeout (goal, 2.4s).

The heartbeat timeout is therefore the addition of the above values, $1,694ms$.

In order to consider Fluidanimate sufficiently resilient, it needs to be able to perform a checkpoint (saving data into NVRAM), computation, and restart (recovering data from NVRAM) all within the timeout period. So the following equation must hold.

$$T_{checkpoint} + T_{restart} + T_{compute} < T_{timeout} \tag{2}$$

Here $T_{checkpoint}$ is the time it takes to perform a checkpoint, $T_{restart}$ is the time it takes to revert the data and restart the process to the last checkpoint, $T_{compute}$ is the time it takes to compute a single iteration, and $T_{timeout}$ is the heartbeat timeout.

Figure 11 shows the time it takes for Fluidanimate to restart and progress to the next iteration, when using each of the four NVRAM APIs. Here using the logging, multi-versioning, file-system, and DRAM cache APIs took 3,255 ms, 1,652 ms, 1,235 ms, and 2,648 ms, respectively. With the timeout being 1,694 ms, the logging and DRAM cache APIs were not fast enough; whereas the two harder-to-use APIs, multi-versioning and file-system APIs, did meet the time constraint.

In order for the Fluidanimate application to continue using the logging and DRAM cache APIs, its timeout strategy must be modified to allow a lower frequency of checkpointing (data saving and recovery). Here we define a variable, $resiliency\_interval$, as the modified application checkpoints data only when its iteration count satisfies $i\%resiliency\_interval == 0$. This changes the timeout equation to:

$$T_{timeout} = (T_{hm} + T_i + T_r + T_w) * resiliency\_interval \tag{3}$$

Figure 12 compares the time it takes for the four NVRAM APIs used by Fluidanimate to support instant restart with a resiliency interval of 3x, thereby scaling the timeout to 5,082 ms. Here the logging, multi-versioning, filesystem, and DRAM cache approaches take 5,029 ms, 3,236 ms, 2,840 ms, and 4,199 ms, respectively. By changing the frequency of checkpoint and heartbeat calls, all four APIs are able to support restart within the modified timeout.

5 RELATED WORK

This work is among the first in studying the use of NVRAM in supporting the fault tolerance of scientific workload [21, 27, 43]. Li et al. [43] and Caulfield et al. [21] considered using NVRAM as replacement of DRAM but without the guarantee of data persistence. Elnawawy et al. [27] developed a persistent matrix multiplication kernel using *clflushopt* and *sfence* instructions. So far, there lacks a more comprehensive study of using persistent memory in scientific applications to support fault tolerance.

Fault tolerance of scientific applications is traditionally supported via checkpointing either through magnetic storage systems [17] or over the network [75]. These checkpointing methods can be classified into system-level and application-level. System-level checkpointing methods, e.g., BLCR [32], Condor [44] and Cao [20], automatically save internal memory of the application periodically without requiring any modification to the application. On the other hand, application-level checkpointing requires modification to the source code [5, 18, 26, 28, 63]. Our work uses the application-level checkpointing methods to minimize the overhead of data saving and recovery. Our work considers the use of DIMM-based NVRAM in scientific workload and studies various implications brought about by this new type of storage.

Application-level checkpointing can be incorporated at different levels. For example, algorithm-based fault tolerance (ABFT) requires applications to be restructured at the algorithmic level [34]. Such algorithmic revisions are used by Wu and Chen [68] to allow various linear algebra routines to correct data errors on-line. Beguelin et al. [15] developed a C++ programming model and library to allow developers to use their data structures and checkpoint invocations, and the framework will automatically checkpoint, load balance, and restart applications to the previous checkpoint. Our modification of the Fluidanimate benchmark does not involve any algorithm or programming model revisions, and is thus more straightforward to integrate.

Additional work has been done to automate the changing of an application to incorporate fault tolerance. Rodríguez et al. [61] developed an open source checkpointing library, CPPC, which uses a compiler to automatically insert checkpoint calls into an application. Bronevetsky et al. [17] created a compiler to insert non-blocking checkpointing into MPI programs.

Checkpointing can also be supported internally at the hardware level, transparent to the software. For example, error masking is a technique that uses circuitry to provide fault tolerance through redundancy, e.g., through triple modular redundancy [48] and quadded logic [36]. Mills et al. [50] introduced an optimization to error masking, *shadow computing*, which reduces the amount of computational redundancy by running process clones on separate nodes at reduced processor speeds to decrease power consumption.

A significant drawback in traditional fault tolerance mechanisms is the performance overhead induced. A study by Gupta et al. [31] showed that the parallel ocean CFD application could only perform a checkpoint once every eight minutes in order to keep 7% of total application execution time to be spent on fault tolerance. When considering a distributed computing environment hosting a CFD application, nodes frequently communicate once per time step [40], where a single time step usually won't exceed a minute [69]. A large amount of communication take place within that eight-minute gap, thus requiring complicated restart procedures. Since NVRAM offers over 40× speed up to the magnetic disks, this eight-minute gap can potentially be reduced significantly to allow a single node to overlap checkpoints with system communications.

NVRAM can be viewed as a hardware option for supporting low overhead fault tolerance. NVRAM has been integrated in persistent memory designs through hardware software co-designs [37, 55, 65, 74] and as a new type of file system [56, 70]. The use of NVRAM is supported via various library APIs [12, 23, 35, 62, 64, 73] and have been used to support databases [38] and persistent data structure [52, 71]. In this paper, we used the Intel PMDK [12] to implement data persistence for scientific workload.

6 CONCLUSION

This paper studies the software development and performance implications of using non-volatile memory (NVRAM) to support fault tolerance in scientific applications. We extend a computational fluid dynamics benchmark, Fluidanimate, with four different APIs to support NVM-based fault tolerance: 1) logging through transactions, 2) multi-versioning through copy-on-write operations, 3) checkpointing through IO operations on a DAX filesystem, and 4) checkpointing with a DRAM cache. We discover that simple extensions of the computation does not take full advantage of the increased IO speeds provided by NVRAM, and additional data structure modifications are often needed to make full use of the speeds. We show that when using a single process, the performance scalability of the original application degrades significantly when using existing techniques, which support only single-threaded checkpointing. Finally, we demonstrate that NVRAM based fault tolerance can be used in a heartbeat-based distributed computing environment to support instant recovery of individual processes when hardware failures occur, without affecting the progress of other active processes.

REFERENCES

[1] 2011. Nuts and Bolts of Multithreaded Programming | Intel® Software. https://software.intel.com/en-us/articles/nuts-and-bolts-of-multithreaded-programming.

[2] 2016. Intel 64 and IA-32 Architectures Software Developer's Manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf.

[3] 2017. perf(1) - Linux manual page. http://man7.org/linux/man-pages/man1/perf.1.html.

[4] 2017. unlink(2) - Linux manual page. http://man7.org/linux/man-pages/man2/unlink.2.html.

[5] . 2018. CRIU: CheckpointRestore In Userspace. https://criu.org/Main_Page

[6] 2019. access time Definition from PC Magazine Encyclopedia. https://www.pcmag.com/encyclopedia/term/37400/access-time.

[7] 2019. Date and time utilities - cppreference.com. https://en.cppreference.com/w/cpp/chrono.

[8] 2019. dd(1) - Linux manual page. http://man7.org/linux/man-pages/man1/dd.1.html.

[9] 2019. fopen(3) - Linux manual page. http://man7.org/linux/man-pages/man3/fopen.3.html.

[10] 2019. https://www.kernel.org/doc/Documentation/filesystems/dax.txt. https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[11] 2019. Intel Optane DC Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[12] 2019. PMDK: Persistent Memory Development Kit. https://github.com/pmem/pmdk/.

[13] 2019. What is Access Time? Webopedia Definition. https://www.webopedia.com/TERM/A/access_time.html.

[14] 2019. What is distributed computing. https://www.ibm.com/support/knowledgecenter/en/SSAL2T_8.1.0/com.ibm.cics.tx.doc/concepts/c_wht_is_distd_comptg.html.

[15] Adam Beguelin, Erik Seligman, and Peter Stephan. 1997. Application level fault tolerance in heterogeneous networks of workstations. *J. Parallel and Distrib. Comput.* 43, 2 (1997), 147–155.

[16] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors.* Ph.D. Dissertation. Princeton University.

[17] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. 2003. Automated application-level checkpointing of MPI programs. In *ACM Sigplan Notices*, Vol. 38. ACM, 84–94.

[18] Greg Bronevetsky, Daniel Marques, Keshav Pingali, Peter Szwed, and Martin Schulz. 2004. Application-level Checkpointing for Shared Memory Programs. *SIGPLAN Not.* 39, 11 (Oct. 2004), 235–247. https://doi.org/10.1145/1037187.1024421

[19] Stewart Cant. 2002. High-performance computing in computational fluid dynamics: progress and challenges. *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences* 360, 1795 (2002), 1211–1225.

[20] J. Cao, K. Arya, R. Garg, S. Matott, D. K. Panda, H. Subramoni, J. Vienne, and G. Cooperman. 2016. System-Level Scalable Checkpoint-Restart for Petascale Computing. In *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS).* 932–941. https://doi.org/10.1109/ICPADS.2016.0125

[21] Adrian M. Caulfield, Joel Coburn, Todor Mollov, Arup De, Ameen Akel, Jiahua He, Arun Jagatheesan, Rajesh K. Gupta, Allan Snavely, and Steven Swanson. 2010. 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 (SC '10).* IEEE Computer Society, Washington, DC, USA, 1–11. https://doi.org/10.1109/SC.2010.56

[22] Huai-Yu Cheng, M BrightSky, S Raoux, CF Chen, PY Du, JY Wu, YY Lin, TH Hsu, Y Zhu, S Kim, et al. 2013. Atomic-level engineering of phase change material for novel fast-switching and high-endurance PCM for storage class memory application. In *2013 IEEE International Electron Devices Meeting.* IEEE, 30–6.

[23] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. *SIGPLAN Not.* 46, 3 (March 2011), 105–118. https://doi.org/10.1145/1961296.1950380

[24] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09).* ACM, New York, NY, USA, 133–146. https://doi.org/10.1145/1629575.1629589

[25] Camille Coti, Thomas Herault, Pierre Lemarinier, Laurence Pilard, Ala Rezmerita, Eric Rodriguezb, and Franck Cappello. 2006. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI. In *SC'06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing.* IEEE, 18–18.

[26] N. El-Sayed and B. Schroeder. 2014. To checkpoint or not to checkpoint: Understanding energy-performance-I/O tradeoffs in HPC checkpointing. In *2014 IEEE International Conference on Cluster Computing (CLUSTER).* 93–102. https://doi.org/10.1109/CLUSTER.2014.6968778

[27] Hussein Elnawawy, Mohammad Alshboul, James Tuck, and Yan Solihin. 2017. Efficient checkpointing of loop-based codes for non-volatile main memory. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT).* IEEE, 318–329.

[28] M. Gamell, D. S. Katz, H. Kolla, J. Chen, S. Klasky, and M. Parashar. 2014. Exploring Automatic, Online Failure Recovery for Scientific Applications at Extreme Scales. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* 895–906. https://doi.org/10.1109/SC.2014.78

[29] Al Geist. 2016. Supercomputing's monster in the closet. *IEEE Spectrum* 53, 3 (2016), 30–35.

[30] NVM Programming Technical Work Group. 2016. pmem.io: How to emulate Persistent Memory. https://pmem.io/2016/02/22/pm-emulation.html

[31] Rinku Gupta, Harish Naik, and Pete Beckman. 2011. Understanding checkpointing overheads on massive-scale systems: Analysis of the ibm blue gene/p system. *The*

International Journal of High Performance Computing Applications 25, 2 (2011), 180–192.

[32] Paul H Hargrove and Jason C Duell. 2006. Berkeley lab checkpoint/restart (BLCR) for Linux clusters. *Journal of Physics: Conference Series* 46 (sep 2006), 494–499. https://doi.org/10.1088/1742-6596/46/1/067

[33] Vincent Heuveline and Andrea Walther. 2006. Online checkpointing for parallel adjoint computation in PDEs: Application to goal-oriented adaptivity and flow control. In *European Conference on Parallel Processing*. Springer, 689–699.

[34] Kuang-Hua Huang and Jacob A Abraham. 1984. Algorithm-based fault tolerance for matrix operations. *IEEE transactions on computers* 100, 6 (1984), 518–528.

[35] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. *SIGPLAN Not.* 51, 4 (March 2016), 427–442. https://doi.org/10.1145/2954679.2872410

[36] Paul A Jensen. 1963. Quadded NOR logic. *IEEE Transactions on Reliability* 12, 3 (1963), 22–31.

[37] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas. 2018. DHTM: Durable Hardware Transactional Memory. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 452–465. https://doi.org/10.1109/ISCA.2018.00045

[38] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. 2019. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 191–205. https://www.usenix.org/conference/fast19/presentation/kaiyrakhmet

[39] Sudarsun Kannan, Ada Gavrilovska, Karsten Schwan, and Dejan Milojicic. 2013. Optimizing checkpoints using nvm as virtual memory. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 29–40.

[40] David E Keyes, Dinesh K Kaushik, and Barry F Smith. 2000. Prospects for CFD on petaflops systems. In *Parallel Solution of Partial Differential Equations*. Springer, 247–277.

[41] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M. Chen, and Thomas F. Wenisch. 2016. Delegated Persist Ordering. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. IEEE Press, Piscataway, NJ, USA, Article 58, 13 pages. http://dl.acm.org/citation.cfm?id=3195638.3195709

[42] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting Phase Change Memory As a Scalable Dram Alternative. *SIGARCH Comput. Archit. News* 37, 3 (June 2009), 2–13. https://doi.org/10.1145/1555815.1555758

[43] D. Li, J. S. Vetter, G. Marin, C. McCurdy, C. Cira, Z. Liu, and W. Yu. 2012. Identifying Opportunities for Byte-Addressable Non-Volatile Memory in Extreme-Scale Scientific Applications. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. 945–956. https://doi.org/10.1109/IPDPS.2012.89

[44] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. 1997. *Checkpoint and migration of UNIX processes in the Condor distributed processing system*. Technical Report. University of Wisconsin-Madison Department of Computer Sciences.

[45] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 329–343. https://doi.org/10.1145/3037697.3037714

[46] Raymond A Lorie. 1977. Physical integrity in a large segmented database. *ACM Transactions on Database Systems (TODS)* 2, 1 (1977), 91–104.

[47] Youyou Lu, Jiwu Shu, and Long Sun. 2016. Blurred Persistence: Efficient Transactions in Persistent Memory. *Trans. Storage* 12, 1, Article 3 (Jan. 2016), 29 pages. https://doi.org/10.1145/2851504

[48] Robert E Lyons and Wouter Vanderkulk. 1962. The use of triple-modular redundancy to improve computer reliability. *IBM journal of research and development* 6, 2 (1962), 200–209.

[49] Micron. 2016. 3D XPoint Technology: Breakthrough Nonvolatile Memory Technology. https://www.micron.com/products/advanced-solutions/3d-xpoint-technology

[50] Bryan Mills, Taieb Znati, and Rami Melhem. 2014. Shadow computing: An energy-aware fault tolerant computing model. In *2014 International Conference on Computing, Networking and Communications (ICNC)*. IEEE, 73–77.

[51] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. 2017. An Analysis of Persistent Memory Use with WHISPER. *SIGOPS Oper. Syst. Rev.* 51, 2 (April 2017), 135–148. https://doi.org/10.1145/3093315.3037730

[52] Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-Optimized Dynamic Hashing for Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 31–44. https://www.usenix.org/conference/fast19/presentation/nam

[53] Xiang Ni, Esteban Meneses, and Laxmikant V Kalé. 2012. Hiding checkpoint overhead in HPC applications with a semi-blocking algorithm. In *2012 IEEE International Conference on Cluster Computing*. IEEE, 364–372.

[54] Ahmad Shukri Mohd Noor and Mustafa Mat Deris. 2009. Extended heartbeat mechanism for fault detection service methodology. In *International Conference on Grid and Distributed Computing*. Springer, 88–95.

[55] M. A. Ogleari, E. L. Miller, and J. Zhao. 2018. Steal but No Force: Efficient Hardware Undo+Redo Logging for Persistent Memory Systems. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 336–349. https://doi.org/10.1109/HPCA.2018.00037

[56] Jiaxin Ou, Jiwu Shu, and Youyou Lu. 2016. A High Performance File System for Non-volatile Main Memory. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. ACM, New York, NY, USA, Article 12, 16 pages. https://doi.org/10.1145/2901318.2901324

[57] Joaquim Peiró and Spencer Sherwin. 2005. Finite difference, finite element and finite volume methods for partial differential equations. In *Handbook of materials modeling*. Springer, 2415–2446.

[58] Simone Raoux, Geoffrey W Burr, Matthew J Breitwisch, Charles T Rettner, Yi-Chou Chen, Robert M Shelby, Martin Salinga, Daniel Krebs, Shih-Hung Chen, Hsiang-Lan Lung, et al. 2008. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development* 52, 4/5 (2008), 465.

[59] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutiu. 2015. ThyNVM: Enabling software-transparent crash consistency in persistent memory systems. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 672–685. https://doi.org/10.1145/2830772.2830802

[60] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-Tree Filesystem. *Trans. Storage* 9, 3, Article 9 (Aug. 2013), 32 pages. https://doi.org/10.1145/2501620.2501623

[61] Gabriel Rodríguez, María J Martín, Patricia González, Juan Tourino, and Ramón Doallo. 2010. CPPC: a compiler-assisted tool for portable checkpointing of message-passing applications. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 749–766.

[62] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. *SIGPLAN Not.* 47, 4 (March 2011), 91–104. https://doi.org/10.1145/2248487.1950379

[63] John Paul Walters and Vipin Chaudhary. 2006. Application-Level Checkpointing Techniques for Parallel Programs. In *Distributed Computing and Internet Technology*, Sanjay K. Madria, Kajal T. Claypool, Rajgopal Kannan, Prem Uppuluri, and Manoj Madhava Gore (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 221–234.

[64] T. Wang, J. Levandoski, and P. Larson. 2018. Easy Lock-Free Indexing in Non-Volatile Memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 461–472. https://doi.org/10.1109/ICDE.2018.00049

[65] T. Wang, S. Sambasivam, and J. Tuck. 2018. Hardware Supported Permission Checks on Persistent Objects for Performance and Programmability. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 466–478. https://doi.org/10.1109/ISCA.2018.00046

[66] Yi-Min Wang, Pi-Yu Chung, Yennun Huang, and Elmootazbellah N Elnozahy. 1997. Integrating checkpointing with transaction processing. In *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*. IEEE, 304–308.

[67] Yi-Min Wang, Yennun Huang, Kiem-Phong Vo, Pe-Yu Chung, and Chandra Kintala. 1995. Checkpointing and its applications. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers*. IEEE, 22–31.

[68] Panruo Wu and Zizhong Chen. 2014. FT-ScaLAPACK: Correcting soft errors online for ScaLAPACK Cholesky, QR, and LU factorization routines. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. ACM, 49–60.

[69] Ren Xiaoguang, Xu Xinhai, Tang Yuhua, and Fang Xudong. 2014. The Analysis of Checkpoint Strategies for Large-Scale CFD Simulation in HPC System. In *2014 Fourth International Conference on Communication Systems and Network Technologies*. IEEE, 1097–1101.

[70] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 323–338. https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu

[71] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. USENIX Association, Santa Clara, CA, 167–181. https://www.usenix.org/conference/fast15/technical-sessions/presentation/yang

[72] Tatu Ylönen. 1992. Concurrent shadow paging: A new direction for database research. (1992).

[73] Lu Zhang and Steven Swanson. 2019. Pangolin: A Fault-Tolerant Persistent Memory Programming Library. *CoRR* abs/1904.10083 (2019). arXiv:1904.10083 http://arxiv.org/abs/1904.10083

[74] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi. 2013. Kiln: Closing the performance gap between systems with and without persistence support. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 421–432.

[75] Avi Ziv and Jehoshua Bruck. 1996. Efficient checkpointing over local area networks. In *Proceedings of IEEE Workshop on Fault-Tolerant Parallel and Distributed*

Systems. IEEE, 30–35.