Compiler Aided Checkpointing using Crash-Consistent Data Structures in NVMM Systems

Tyler Coy
School of Engineering and Computer Science
Washington State University
Vancouver, WA, USA
tyler.coy@wsu.edu

Shuibing He
College of Computer Science and Technology
Zhejiang University
Hangzhou, Zhejiang, China
heshuibing@zju.edu.cn

Bin Ren
Department of Computer Science
College of William & Mary
Williamsburg, VA, USA
bren@cs.wm.edu

Xuechen Zhang
School of Engineering and Computer Science
Washington State University
Vancouver, WA, USA
xuechen.zhang@wsu.edu

ABSTRACT

Scientific applications use checkpointing for failure recovery. The existing checkpointing approaches were proposed for storing persistent states of applications as checkpoints in disk-based file systems via the block interface. As non-volatile main memory (NVMM) will be included in high-performance computing systems, storing the checkpoints in NVMM-based file systems can significantly waste the performance benefits of NVMM. This is because it underutilizes memory resources and it does not take advantage of the byte-addressability of NVMM.

In this paper, we propose an NVMM-aware checkpointing approach, named NV-Checkpoint. It uses a compiler-aided technique to automatically generate multi-version data structures, which consist of both the persistent version of data stored in NVMM for failure recovery and the ephemeral version of data placed across DRAM and NVMM. Because of the byte-addressability of NVMM, any versions can be accessed via the memory interface. The multiple versions may share data that are not mutated during the program's execution to reduce data redundancy. NV-Checkpoint provides the same level of guarantee of failure recovery compared to the conventional checkpointing approaches proposed for file systems. Furthermore, its runtime system manages the layout of the data structures to reduce the number of writes to NVMM. It also manages the checkpointing frequency to reduce persistence overhead using machine learning models. Our experimental results with real-world scientific applications show that the performance of annotated programs with NV-Checkpoint using a hybrid of DRAM and NVMM matches the performance of best-effort hand-written versions. It achieves similar scalability as those with ephemeral data structures using only DRAM. It offers up to 121X speedup of execution time compared to the conventional checkpointing approaches using the Atlas parallel file system on the Titan supercomputer.

CCS CONCEPTS

• **Computer systems organization → Reliability**; **Processors and memory architectures**;

KEYWORDS

Checkpointing, Compiler Aided, Non-Volatile Main Memory

ACM Reference Format:

Tyler Coy, Shuibing He, Bin Ren, and Xuechen Zhang. 2020. Compiler Aided Checkpointing using Crash-Consistent Data Structures in NVMM Systems. In *2020 International Conference on Supercomputing (ICS '20), June 29-July 2, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3392717.3392755

1 INTRODUCTION

Ordinary data structures are *ephemeral* in the sense that only the newest version of data is stored in memory after making changes [14]. They are parallelized and ubiquitous in scientific simulations and analytics, from *arrays* in LAMMPS for molecular dynamics simulations [39] to *octrees* in Gerris for solving partial differential equations describing fluid flows [1, 33], to *graphs* in biological networking analysis [63]. These ephemeral data structures are usually well-tuned for DRAM-based high-performance computing (HPC) systems [4, 19, 45, 46]. Large-scale HPC systems are likely to be interrupted by failures because its components are not reliable [21, 35, 50]. Consequently, checkpointing is required for long-running HPC applications designed with the ephemeral data structures to provide failure recovery.

Checkpointing can be implemented at the system level or the application level. The system-level checkpointing approaches save the memory states of the entire memory systems [20, 26]. They are usually transparent to the end-users of the HPC systems at the cost of high checkpointing overhead. Application-level checkpointing saves only application states as snapshots (or logs) and store them in file systems [6, 15]. Because of its much smaller checkpointing overhead, the application-level checkpointing is widely adopted in HPC applications (e.g., Gerris and LAMMPS). For checkpointing,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICS '20, June 29-July 2, 2020, Barcelona, Spain

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-7983-0/20/06…$15.00
https://doi.org/10.1145/3392717.3392755

saving the persistent memory states to the disk-based file systems may cause a performance bottleneck in large-scale HPC systems [61, 62]. As non-volatile main memory (NVMM) is increasingly adopted in HPC systems, we attempt to use NVMM for storing the persistent states of applications to reduce CPU stall time caused by accessing the slow I/O buses.

When NVMM is used as a block device, simply storing the snapshots in NVMM-based file systems seems a straightforward approach because no changes are required for HPC applications. However, it can significantly waste the full benefits of NVMM for two reasons. (1) *This approach under-utilizes memory resources.* After the checkpointing operations are executed, serialized snapshot files containing application states are saved in file systems. At the same time, the deserialized application states still live in the form of memory objects (e.g., arrays and trees) in DRAM. They may contain mostly duplicated data stored in DRAM and NVMM, separately. (2) *It does not take advantage of the byte-addressability of NVMM.* This is because the snapshots in the file format in NVMM are only accessed for failure recovery. It cannot be accessed by an application during its normal execution.

In this paper, we propose an NVMM-aware compiler-aided approach for application-level checkpointing. It is named NV-Checkpoint and has three novel features. First, it replaces ephemeral data structures with crash-consistent multi-version data structures. Each snapshot is mapped to a version of the data managed by the data structures. At least one version will be stored in NVMM and immutable during the execution of applications until a newer snapshot becomes persistent. The persistent version can be used as a snapshot for failure recovery. NV-Checkpoint allows the ephemeral versions to be placed in both DRAM and NVMM together with its persistent version in NVMM. Because of the byte-addressability of NVMM, any versions can be accessed via memory interface supported by operating systems/runtime. Furthermore, NV-Checkpoint allows data sharing between the ephemeral versions and its corresponding persistent version of the data. As a result, it does not need to store duplicated data separately in DRAM and NVMM.

Second, it uses a compiler-aided technique to transform the data structures automatically. End-users may annotate program variables that are vulnerable to data inconsistency upon failures. The compiler can then generate NVMM-aware code from the annotated version of the code. For annotations, NV-Checkpoint provides a simple but flexible set of annotations to specify the ephemeral data structures and provide application's hints. After the code annotations, NV-Checkpoint uses a source-to-source compiler to create a data enclave structure that uses multi-version data structures with persistent pointers allocated in NVMM to replace ephemeral data structures. The transformed data structures provide the same level of guarantee of failure recovery compared to the conventional checkpointing approaches proposed for file systems. Compiler-aided code annotation and transformation are general and can be applied to applications using various data structures. NV-Checkpoint requires no expertise in software design using NVMM. It keeps the baseline code structure for good readability and maintenance.

Third, its runtime system manages the checkpointing frequency to reduce persistence overhead using machine learning models. It also manages the layout of the data structures to reduce the number of writes to NVMM using the application's hints specified by end-users.

Many scientific applications will benefit from adopting NV-Checkpoint, such as those that demand more memory, those that require consistent checkpoints, and those that do not checkpoint data but may benefit from NVMM for persistence. Specifically, we made the following contributions.

- We propose a novel compiler-aided technique to automatically transform an ephemeral data structure to its corresponding crash-consistent multi-version data structures for application-level checkpointing in NVMM systems. It significantly reduces the burden of programmers of using NVMM models.
- In the runtime system of NV-Checkpoint, we use machine learning models to automatically determine when to create a persistent version of a data structure from its ephemeral version which resides in both DRAM and NVMM considering the cost of data persistence and the time of recomputing for failure recovery.
- We implement a software prototype of NV-Checkpoint and apply it to parallel programs (e.g., LU decomposition, adaptive-mesh refinement, page ranking, and LAMMPS). Our experimental results show that the programs using NV-Checkpoint achieve similar performance and scalability compared to its ephemeral version on the Titan supercomputer. It reduces checkpointing overhead by up to 99% compared to that using disk-based application-level checkpointing approaches.

The rest of the paper is organized as follows. Section 2 explains why ephemeral data structures are not crash-consistent in NVMM systems using examples. Section 3 describes how to build crash-consistent multi-version data structures. Section 4 describes the software architecture of NV-Checkpoint, presents the annotations and source-to-source compiler, and describes its runtime system. Section 5 discusses implementation issues and Section 6 describes and analyzes experimental results. In Section 7 we introduce related work. And Section 8 concludes the paper.

2 EPHEMERAL DATA STRUCTURES ARE NOT CRASH-CONSISTENT IN NVMM SYSTEMS

Crash consistency is the recoverability of persistent data from memory in a consistent state after system failures. We studied the crash consistency of three representative ephemeral data structures (i.e., arrays, quad/octrees, and graphs), which are widely used in scientific simulations and analysis. We found none of them is crash-consistent upon failures when its ephemeral version of the data structures resides in NVMM.

Array objects are not crash-consistent because of *partial updates after failures*. We use LU decomposition as an example. LU decomposition factors a matrix A as the product of a lower triangular matrix L and an upper triangular matrix U [56]. The execution of LU decomposition program typically consists of a sequence of iterations. The normal output after iteration i is denoted as A^i in Figure 1(a). All the elements in the shaded area are updated in iteration i. If a failure happens before the completion of iteration i or before a CPU cache flush to make A^i persistent in NVMM, the matrix A after failed LU decomposition (shown in Figure 1(b))

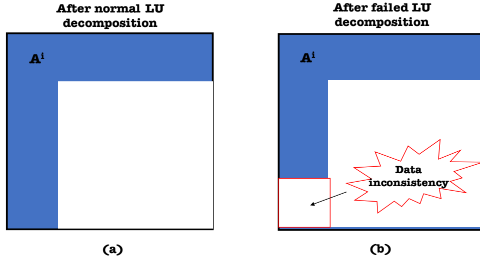

Figure 1: Inconsistent matrix with partial updates. The shaded elements are updated in the iteration i.

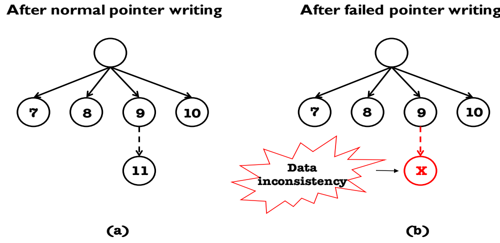

Figure 2: Inconsistent octree with the pointer linking to incorrect NVMM address.

may not match the normal output without failures (shown in Figure 1(a)). This will cause data inconsistency in NVMM concerning array elements in the red block.

Quad/octree objects are not crash-consistent because of *out-of-order memory writes from CPU cache.* Let's use the quadtree in Figure 2(a) as an example. If quadrant 9 needs to be refined, we can initialize a new quadrant 11 and then write a pointer linking quadrant 9 to 11. Because CPU cache does not guarantee the order of writing quadrant 11 (w_1) and writing the pointer (w_2) for optimization of memory access [51], the pointer might be written to NVMM before the new quadrant 11. A failure between w_2 and w_1 can cause the pointer to link to an undefined region in NVMM (shown in Figure 2(b)), thus resulting in data inconsistency in NVMM after failed pointer writing.

Graph objects are not crash-consistent because of *inconsistent updates of correlated variables.* The in-memory graph data structures mainly consist of three parts: topology data (i.e., node tables and edge tables), runtime states (e.g., activeness/inactiveness of nodes), and application-defined data (e.g., page ranks of nodes). For a large body of graph applications, every write access to a node in NVMM consists of at least two separate write requests to NVMM. For example, in page ranking, each write access to a node requires one write (w_a) to update its rank score and the other one (w_b) to update its activeness for determining the termination of the application. If a system failure occurs after w_a and before the completion of w_b, the memory controller would observe a stale activeness value upon system recovery, introducing data inconsistency.

In summary, while current OS services supporting NVMM technologies are well developed, scientific simulations using these ephemeral data structures are still vulnerable to data inconsistency upon failures in HPC systems. Thus, there is a need for efficient application-level checkpointing approaches exploring both the persistence and byte-addressability of NVMM.

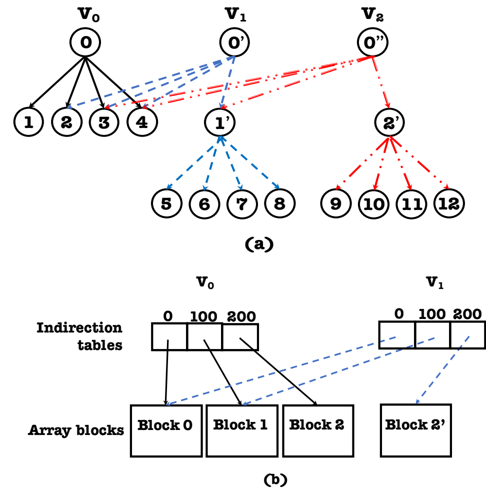

Figure 3: An illustration of a three-versions quadtree (a) and a two-version array (b).

3 MULTI-VERSION DATA STRUCTURES FOR CHECKPOINTING IN NVMM SYSTEMS

In this paper, we assume scientific simulations and analytics consist of multiple iterations (e.g., loop-based code). We also assume the applications need to save the memory states of ephemeral data structures that are required for failure recovery at the end of an iteration as a snapshot of the memory states. We assume that applications themselves determine the checkpointing frequency. NV-Checkpoint does not change their consistency models.

We map each snapshot of an ephemeral data structure to a unique version managed by its corresponding multi-version data structure. As a result, using NV-Checkpoint, creating a new snapshot is essentially creating a new persistent version and storing it in NVMM. Next, we illustrate the in-memory representation of multi-version quad/octrees, arrays, and graphs and how to provide failure recovery with them.

Multi-version quad/octrees. We use the quadtree data structure as an example (shown in Figure 3(a)) to show the memory representation of its corresponding multi-version quadtree. It has three versions, denoted as V_0, V_1, and V_2, each of which represents the memory state of a quadtree at the end of its corresponding iteration 0, 1, and 2 respectively. V_0 consists of a unique header node 0, quadrants 1, 2, 3, and 4, and pointers (solid-arrow arrows). V_1 has a unique header node $0'$, five additional quadrants $1'$, 5, 6, 7, and 8, and pointers (dashed blue arrows). V_2 has a header node $0''$, five more quadrants $2'$, 9, 10, 11, and 12, and pointers (dashed red arrows).

Multi-version arrays. We may build multi-version arrays using indirection tables. Each version of the array has its own indirection table for indexing the reference address of each element in the array in the specified version. A two-version array is shown in Figure 3(b).

The ephemeral array is partitioned into three blocks, whose size is 100 elements. V_0 consists of Block 0 − 2 and V_1 consists of Block 0 − 1 and Block 2′. V_0 and V_1 share Block 0 and 1. Users can specify the block size using NV-Checkpoint API.

Multi-version graphs. Because most of the graph data structures may be implemented using arrays in compressed-sparserow (CSR) format [38], we can use multi-version arrays to construct multi-version graphs. A more detailed discussion of multi-version graphs can be found in the prior work [28, 29].

Checkpointing using multi-version data structures in NVMM systems: Using the conventional checkpointing approach, each version of the data structure is saved in a checkpoint in file systems for durability. *In this project, we store such a snapshot as a persistent version of the corresponding data structure in NVMM.* Leveraging the byte-addressability of NVMM, NV-Checkpoint uses the multi-version data structure for both computing and in-memory storage. Let's use the quadtree in Figure 3(a) as an example. In iteration 0, V_0 is ephemeral initially. It can be placed across DRAM and NVMM. At the end of iteration 0, checkpointing operation is executed by writing all the quadrants in DRAM to NVMM to create a persistent version of the quadtree. Then V_0 becomes persistent in NVMM. In iteration 1, it is immutable while V_1 is being updated. Any writes to V_0 will result in writes to V_1. The root nodes (or indirection tables for multi-version arrays) of the persistent version is recorded in the pre-defined NVMM addresses. If a fail-stop failure happens in iteration 1, NV-Checkpoint may use V_0 to provide failure recovery.

4 NV-CHECKPOINT DESIGN

The objective of NV-Checkpoint is to automatically generate NVMM-aware code from an annotated version of the code. We use a compiler-aided technique to overcome the difficulties in manually transforming data structures. This approach could disrupt system performance if it were indiscriminately applied. To be effective, NV-Checkpoint continuously evaluates the cost-effectiveness of data persistence overhead in determining when to persistent them in NVMM at runtime to improve checkpointing performance.

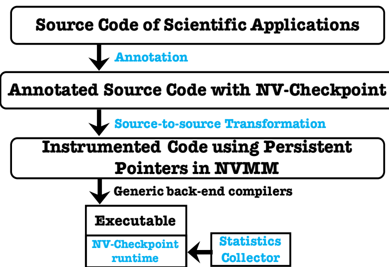

Figure 4: The software architecture of the NV-Checkpoint framework.

Figure 4 illustrates the overall procedure of the NV-Checkpoint framework. NV-Checkpoint consists of *annotation functions*, a *source-to-source compiler*, and a *runtime system*. We assume programmers have sufficient knowledge of the code to identify data structures and variables in the source code that are required to save in NVMM for checkpointing. They first annotate the code using the annotation functions described in Section 4.1. Then the compiler will transform the annotated source into the one that uses multi-version data structures with persistent pointers allocated in NVMM to replace ephemeral data structures. The new data structure is crash-consistent because at least one version of data will be stored in NVMM and immutable until a newer version becomes persistent. Its runtime system determines an optimal persistent interval using a cost model taking into account data persistence overhead, which is predicated by machine learning models, and recomputing time (discussed in Section 4.2). Finally, it hides NVMM latency through the layout transformation of data structures considering applications' hints (discussed in Section 5). NV-Checkpoint supports the same consistency model as the conventional checkpointing approaches. If the existing checkpointing functions in HPC applications requires a synchronizing collection operation, the instrumented function with NV-Checkpoint also implements the synchronizing operation across compute nodes.

4.1 Source Code Annotations

The language interface is designed to be as simple as possible. It comprises the following directives.

#pragma nvcp. It tells the compiler to generate a multi-version data structure with persistent head pointers in NVMM for the specified ephemeral data structures in the code.

#pragma nvcp init(ds_type, #version, [block_size]). It instructs the compiler to create a data enclave for managing the related data and metadata stored in NVMM. First, NV-Checkpoint creates persistent data structures (e.g., *struct _nv_FttCell* in Figure 5(b)) using the specified ephemeral data structures ((e.g., *struct _FttCell*)) by replacing DRAM pointers with NVMM pointers (e.g., *persistent_ptr<FttOct> * parent*) which would then reference the persistent version of the data stored in NVMM at runtime. Second, it creates a head node structure (*e.g., struct _FttCell_head*) for managing the data enclave. The structure includes a data field, a DRAM pointer to the ephemeral data, and NVMM pointers to the persistent data. The enclave head node manages at least two version pointers V_i and V_{i-1} in NVMM. For the quad/octrees, they point to the respective head nodes of the tree snapshots. For the arrays, they point to the addresses of indirection tables. Third, it initializes the head node (e.g., *FTT_head * head*) to track the memory that has been allocated. The addresses of V_i and V_{i-1} are recorded at pre-defined addresses and used to access all persistent data after a restore operation for failure recovery. Programmers use *#version* to specify the number of versions in NVMM. Its default value is 2 and the maximum value is capped by the NVMM capacity. Currently, it supports three types of general data structures: arrays, quad/octrees, and graphs with the CSR representation [29]. *block_size* is only valid for the array data structure.

#pragma nvcp add_head(). It links the ephemeral data structure to its corresponding head node of the data enclave. The head node structure residents in both DRAM and NVMM via a memory interface supported by operating systems/runtime[1]. We use an emulated NVMM dynamic allocator and make sure it is compatible with Intel Persistent Memory Programming Interface PMDK [2]

[1] We assume NVMM is attached to the CPU bus alongside DRAM as DIMMs.

```
1 #pragma nvcp
2 typedef struct _FttCell{
3     p<gpointer> data;
4     FttOct *parent, *children;
5     ...
6 }FttCell;
7 #pragma nvcp init(octree, 2)
8
9 new_oct(...){
10     FttCell *rootcell;
11     rootcell = g_malloc0(...);
12     #pragma nvcp add_head(rootcell);
13     rootcell->data = X;
14     ftt_cell_insert(rootcell, ...);
15     FttCell *cell1;
16     ...
17     write_snapshot(rootcell, ...);
18     #pragma nvcp persistent(head, write_snapshot, FttCellRefineFunc);
19 }
```

(a)

```
1 typedef struct _FttCell{
4   ...
5 }FttCell;
6 typedef struct _nv_FttCell{
7   p<gpointer> data;
8   persistent_ptr<FttOct> *parent, *children;
9   ...
10 }nv_FttCell;
11 typedef struct _FttCell_head{
12   p<gpointer> data;
13   FttCell *next;
14   nv_FttCell *version[2];
15 }FttCell_head;
16 FttCell_head * head = malloc(sizeof(FttCell_head));
17
18 nv_new_oct(...){
19   FttCell * rootcell;
20   rootcell = g_malloc0(...);
21   head->next = rootcell;
22   rootcell->data = X;
25   ...
26   persistent_oct(head, FttCellRefineFunc);
27 }
```

(b)

Figure 5: An example of code annotation. (a) Octree creation using NV-Checkpoint with annotations; (b) Automatically generated code for the multi-version octree structure, the head structure, and the *new_oct()* function.

for dynamic allocation from NVMM in the development of NV-Checkpoint because of its compatibility with the emerging 3D XPoint DIMMs.

#pragma nvcp persistent(head, [ADDR(write_func)], [hints]). It calls the NV-Checkpoint runtime to create a new persistent version of the data structure referenced from the version pointer V_{i-1}. Specifically, the data in DRAM referenced using V_i are copied to NVMM referenced using the NVMM pointers. After the completion of the operation, we swap the addresses of V_i and V_{i-1} using atomic CPU instructions. Then all the nodes in V_i but not overlapped with V_{i-1} are marked as 'deleted'. The deleted data will be freed by the garbage collector asynchronously. Finally, new data in V_i may be partitioned across DRAM and NVMM for hiding NVMM write latency. Users can provide *hints* for layout optimization in data partitioning so that frequently-updated data can be stored in DRAM while the rest of them are placed in NVMM. The annotation function may replace the existing checkpoint write functions referenced at *ADDR*(*write_func*()) which were developed for file I/Os using POSIX or MPI-IO API, e.g., *write_snapshot*() in Figure 5(a).

#pragma nvcp restore(head, [ADDR(read_func)]). It calls the NV-Checkpoint runtime to restore the persistent data referenced using the version pointer V_{i-1} to memory objects in DRAM. The objects should be identical to the most recent consistent version of the data structure. It may replace the existing checkpoint file read function using POSIX I/Os with memory address at *ADDR*((*read_func*()).

Figure 5(a) presents an AMR code[2] with annotations after the first pass of the code. This new version is almost identical to the original code except for four embedded directives. The NVMM-aware crash-consistent code appears in Figure 5(b) after the second pass of the code transformer. It includes a head structure consisting of two NVMM pointers referring to each of the two versions (V_i and V_{i-1}) in NVMM and one DRAM pointer to the data in DRAM. The V_i and V_{i-1} components of the octree in NVMM can be referenced at *head*→*version*[0] and *head*→*version*[1] respectively. V_i and the DRAM pointer are visible to users during normal execution. V_{i-1} is immutable and used for both computing with the data shared between V_i and V_{i-1} and providing recovery upon failures.

[2]We use octrees in Gerris to illustrate concepts, but all techniques are designed and applicable to other types of data structures.

4.2 Runtime System for Persistence Management

```
1 void mpilu(array_head * head, double **a, ...){
2    /* The main loop */
3    for (k = 0; k < n; k++) {
4      /*Compute LU decomposition with partial pivoting*/;
5      ...;
6      potential_persistent(head);
7    }
8 }
9
10 void potential_persistent(head){
11    /* Calculate recomputing time. */
12    time_recompute = clock() - time_persistent;
13    /* Compare to persistent cost */
14    if (time_recompute > persistent_cost(head){
15      /* Persistent the data structure. */
16      persistent(head);
17      /* Record the time when a checkpoint is created. */
18      time_persistent = clock();
19    }
20 }
```

Figure 6: Pseudocode of the LU decomposition code and *potential_persistent*() **function.**

Scientific applications may use the persistent data stored in NVMM to roll back to the most recent checkpoint where the state of the simulations is known to be correct. If functions that write checkpoint files to secondary storage exist in the legacy code, the compiler of NV-Checkpoint can automatically replace them with its corresponding persistent version provided in NV-Checkpoint library. If they do not exist, NV-Checkpoint will insert a potential persistent function *potential_persistent*() at the last line inside the main loop. An example is shown in Figure 6. The LU composition code[3] has a main loop which consists of *n* iterations. The time required to establish a persistent checkpoint might be longer than

[3]Following the guidelines of double-blind review, we hide the code repository link here. But it is available upon requests.

Data Structure	Feature
Arrays	ne: total number of array elements in DRAM $size_e$: array element size dim: array dimensions
Quad/octrees	no: total number of quadrants/octants in DRAM $size_o$: quadrant/octant size $depth$: depth of tree $fanout$: tree fanout
Graphs	nn: total number of nodes in DRAM $size_n$: node size $degree$: average degree per node

Table 1: Features collected from arrays, quad/octrees, and graphs.

the time of recomputing from the last checkpoint. This is because the compute time varies across iterations.

To opportunistically create a checkpoint in NVMM, NV-Checkpoint uses a greedy approach inside the potential functions. Specifically, NV-Checkpoint calculates the recomputing time (T_{rc}) which is the duration from the time when the last persistent version of the data structure is generated in NVMM to the time that the clock is examined in *potential_persistent*(). Then it computes the time (T_p) required to establish a new persistent version using a cost model considering the amount of data that needs to be merged from DRAM to NVMM. When the recomputing time is smaller than the cost, NV-Checkpoint skips the persistent operations; otherwise, a new persistent version of the data structure is created in NVMM to replace the previous version. In Figure 6, T_p is produced by the function *persistent_cost*(), which estimates the persistence cost using a cost model.

Cost model: NV-Checkpoint partitions data structure across DRAM and NVMM. We observed that the cost of merging data in DRAM with data in NVMM or the time of executing the *persistent*() function primarily depends on the total size of objects in DRAM and the type of data structures. Our approach is thus to monitor the state of major data structures (e.g., arrays, quad/octrees, and graphs) in simulations and analytics, collect statistics from them as features, and use these features to build machine learning models to predict the time of executing the *persistent*() function at runtime.

More specifically, NV-Checkpoint is designed to record statistics of data structures. Table 1 lists the statistics that we collect for the three data structures related to applications used in this paper. All of these features affect memory consumption and consequently the execution time of object persistent functions. Our experimental results show that the overhead of obtaining the values of these statistics is 1% on average.

With the statistics, we use the following steps to build a machine learning model. (1) To collect training datasets, we randomly partition the data structures across DRAM and NVMM and then run *persistent*() and measure its execution time T_p^i in test runs, where i is the sequence number of a test run. (2) After each test run, we record the value of the data features as a vector FV_i and the execution time T_p^i as a pair. (3) To provide sufficient coverage of potential layout partitioning schemes of data structures, we run the test 70,000 times for each data structure offline. (4) We then feed the pairs of (FV^i, T_p^i) to M5P model [40, 55] from Weka [18] since it gives us the most accurate predictions overall. The detailed results are shown in Section 6.4. (5) A trained model is deployed in HPC systems to predict the cost of persistent operations (i.e., *persistent_cost*() in Figure 6). The decision is made by a centralized master process.

In this paper, we train a model for each of the application, respectively, because we found that an application-specific model achieves higher accuracy than a universal model for all applications. We train the model offline, thus not incurring execution overhead.

5 IMPLEMENTATION ISSUES AND OPTIMIZATION

Latency-aware data placement. NVMM has longer read/write latency than DRAM. At runtime, NV-Checkpoint dynamically changes the layout of in-memory objects to hide the latency according to applications' hints. We found three hints that are particularly effective in partitioning data structures between NVMM and DRAM for scientific applications using the following data structures. *Arrays:* In the matrix-based applications, users can provide matrix indices to inform NV-Checkpoint runtime which matrix regions may be accessed in each iteration. *Quad/octrees:* We may partition quad/octrees using data features which are application-level knowledge and realized as functions for quadrants or octants refinement/coarsening or solver functions traversing the tree in computations. A similar approach was used in PMOctree [33]. *Graphs:* We may partition the application-defined data and runtime states of graphs according to network properties. For example, end-users can use the network centrality [31] of nodes for data partitioning. Research has shown that the network centrality of nodes is correlated to their memory access frequency [28].

Permanent node failures. In the scenario that the compute nodes may be lost permanently after failures, NV-Checkpoint runtime can be configured to add parity data in NVMM to tolerate losing one or multiple compute nodes. The parity data is created using erasure coding schemes [41]. Previous work has shown that the overhead of parity data management is 19% on average for large-scale simulations [32]. We may also leverage the staging nodes or burst-buffer nodes on supercomputers to store the parity data.

Simplicity and generality. We would like to emphasize that the main goal of building NV-Checkpoint is to demonstrate the power of the compiler-aided approach. Although we use specific data structures for illustration throughout the paper, the proposed approach can be applied to other data structures with minor changes (e.g., adding structure-specific functions for accessing data in DRAM in the runtime system) leveraging the power of static analysis tools and compilers. When an ephemeral data structure is designed and implemented, developers and users only need to specify the simple data-structure-specific knowledge (e.g., hints that direct layout transformation) through the annotation functions in NV-Checkpoint. During our evaluation, the time we spent on implementing annotation API is little after understanding each data structure. We believe the actual application developers will take even less effort as they have a better understanding of the code as they implement the applications.

Integration with existing scientific workflows. Checkpoints in real applications are used for more than just fault tolerance. For integration with existing scientific workflows requiring visualization and data steering, NV-Checkpoint can be configured to maintain multiple versions in NVMMs. The data of old versions in NVMMs can be used for in-situ visualization and data steering. Furthermore, NV-Checkpoint can be configured to write checkpoints to file systems when NVMM is full. Users may still read the checkpoint files for visualization and data steering without changing the design of existing workflows.

6 EVALUATION

We conduct an extensive performance study of NV-Checkpoint to experimentally benchmark the performance and scalability of NVMM-aware data structures compared to their corresponding ephemeral data structures and hand-written code.

6.1 Experimental Setup

Computer clusters for NVMM emulation: We evaluated the code annotated with NV-Checkpoint on the Titan supercomputer [49] at Oak Ridge National Laboratory. Titan consists of 18,688 nodes, each of which is configured with a 16-core AMD Opteron 6274 CPU and 32 GB memory. Each node runs Cray Linux Environment operating system. All nodes were interconnected with a Gemini network. They share a site-wide Lustre parallel file system Atlas [47]. Because of the real NVMM hardware is not available on supercomputers when we conducted this work, we model NVMM using DRAM on Titan using an emulation based approach, which is similar to those in other projects [23, 34, 53, 54]. The emulation approach makes our work not depend on any specific NVMM implementation and deployment.

We tried our best to ensure the emulation parameters resemble realistic NVMM. Specifically, the emulated read and write latency are 100 and 150 ns respectively. Our NVMM emulator introduces extra latency for NVMM write and read through routines that write to or read from DRAM. We create delays using a software spin loop [34, 53] that uses the x86 RDTSP instruction to read the processor timestamp counter and spins until the counter reaches the intended delay. We also model NVMM bandwidth by inserting a proper delay after the request sequence completes to limit the effective bandwidth. Specifically, the bandwidth of NVMM is limited to 10 GB/s for write and 35 GB/s for read in the experiments. A similar approach was used in Mnemosyne [54].

Scientific applications. We use four applications with distinct data structures in their compute kernels. They cover a wide range of applications' characteristics from array computation with regular memory access to graph computation with irregular memory access, and from simple matrix computation to complex multi-length-scale fluid physics computation.

- *LU-Decomposition (LU)*: This program is designed to factor a matrix A as the product of a lower triangular matrix L and an upper triangular matrix U. Multiplying the results of the factored matrix (e.g., $L * U$) should return the original matrix A. It is a C program, which duplicates the major functionalities of its corresponding Fortran version in the ScaLAPACK software [36]. All MPI processes of *LU* are used for computations. Specifically, *LU* generates a random matrix in a 2d array named *A*. The data to be computed by all the processes is distributed with cyclic distribution and uses partial pivoting [52]. The maximum array size is 31,500 * 31,500 elements of double precision floating point number.
- *Adaptive Mesh Refinement with Octree (AMR)*: This *AMR* code is designed to simulate droplet movement [22, 48] using MPI. It uses the Cartesian mesh based finite method to simulate the flow. The code consists of five major routines, including creating a new octree on each processor, refining and/or coarsening a domain, balancing the octree, partitioning and distributing octants among processors, and extracting mesh structures for visualization. This is a C++ program and developed based on the popular open-source Gerris flow solver [44]. The maximum input size is 130 million elements in a mesh and the maximum tree depth is 9.
- *Page Ranking (PR)*: The program outputs a probability distribution which represents the relative importance of web pages in networks [37]. It executes a random walk which jumps to a random node with a certain probability α, and follows a randomly chosen outgoing edge of a node with probability $1 - \alpha$ from the current node. *PR* is a C program and uses a master/slave MPI implementation, which is described by Sangamuang et al. [42]. The master sends the number of graph nodes to the slaves which will be responsible for calculating the rank. The graph data is stored in compressed sparse row (CSR) [29] format in memory. The program continues to run until the difference between the values of page ranks computed in consecutive iterations is below a threshold of .000001 or the limit of the number of iterations is reached. We use MAWI graph datasets which were generated from packet trace data from the WIDE backbone maintained by the MAWI working group [12]. The graph has 129 million vertices and 270 million edges. We may use a subset of the graph data in experiments.
- *LAMMPS*: The program is designed as a large-scale atomic and molecular massively parallel simulator [39] from Sandia National Laboratories. It is written in C++ with MPI and performs force and energy calculations on discrete atomic particles. For checkpointing, the *atoms* arrays and their related metadata (i.e., array size, space boundaries, and time steps) are written to persistent storage devices. In the experiments, we use 2d crack simulation. The maximum number of atoms is 253 million.

We evaluate the applications with five combinations of implementations of data structures and storage options. (1) *Vanilla (DRAM)*: it denotes that the program is implemented using ephemeral data structures and executed using only DRAM. (2) *Vanilla (NVMM)*: it denotes that the program is implemented using ephemeral data structures and executed using only NVMM. Both (1) and (2) may suffer data inconsistency upon failures. (3) *Atlas-FS*: it denotes that the program is executed using only DRAM with ephemeral data structures. To provide crash consistency its runtime states are written to the Atlas parallel file system periodically using parallel I/Os. We use the default setting of the file system, in which the files are striped

over 4 object storage targets (OSTs) with 1 MB stripe size. (4) *NVMM-FS*: it denotes that the program is executed using only DRAM with ephemeral data structures. Its runtime states saved in the node-local in-memory file system set up using NVMM. The checkpoints are accessed via file-system interface. (5) *NV-Checkpoint*: we transform the code using NV-Checkpoint and store a persistent version in NVMM. (3), (4), and (5) enforce crash consistency. Besides these, we also compare the performance of NV-Checkpoint to other state-of-the-art checkpointing approaches [16, 24, 28, 33, 56] which require manual instrumentation in the source code in Section 6.6.

6.2 Strong Scaling

Application	Data structures	Problem size
LU	Array	20,480 * 20,480 double float
AMR	Octree	125 million elements in a mesh
PR	Graph	5 million vertices and 69 million edges
LAMMPS	Array	18 million atoms

Table 2: Problem size of applications for strong scaling.

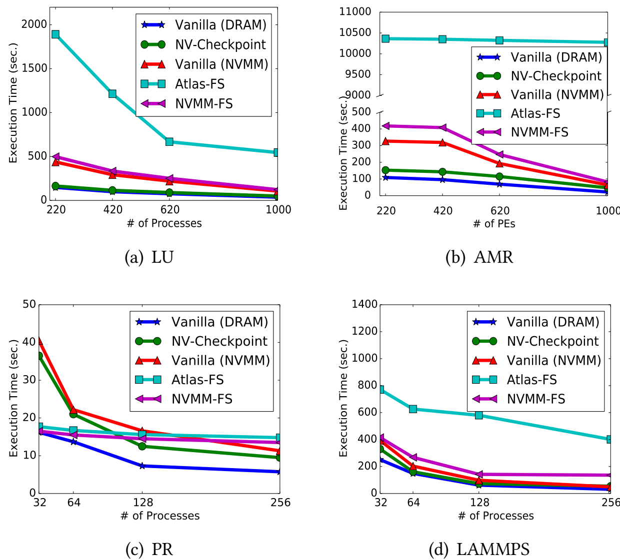

Figure 7: The execution time of applications in the strong-scaling experiments.

To study the strong scaling of the applications with NV-Checkpoint, we keep the problem size constant and increase the number of processes. On Titan, each node has 16 cores. We run one process on a dedicated CPU core. Table 2 shows the problem size for *LU*, *AMR*, *PR*, and *LAMMPS* respectively in the experiment. We have the following observations from the execution time shown in Figure 7. First, the programs with NV-Checkpoint achieve similar strong scaling as the code with the ephemeral data structures. Its average speedup with NV-Checkpoint is only 11.7% lower than that using only DRAM. This is because of the overhead of persistent operations. Second, the program using the Atlas file systems for checkpointing scales poorly when the file size of the snapshots is significant. For example, *AMR* wrote 15.4 GB data to the Atlas file system. Even though Atlas has very high storage bandwidth, the average speedup of *AMR* is only 1.0 with *Atlas-FS* compared to 3.3 with NV-Checkpoint. Third, the scalability of checkpointing approaches using file systems is determined by whether the checkpointing data can be buffered in the cache of file servers. Specifically, when the file size of checkpoints is 38.78 MB for *PR*, we found that it is small and can be effectively served in the cache. In contrast, the file size of *AMR* checkpoints is 406X larger than that of *PR*, making it difficult to hide the I/O latency using the cache.

6.3 Weak Scaling

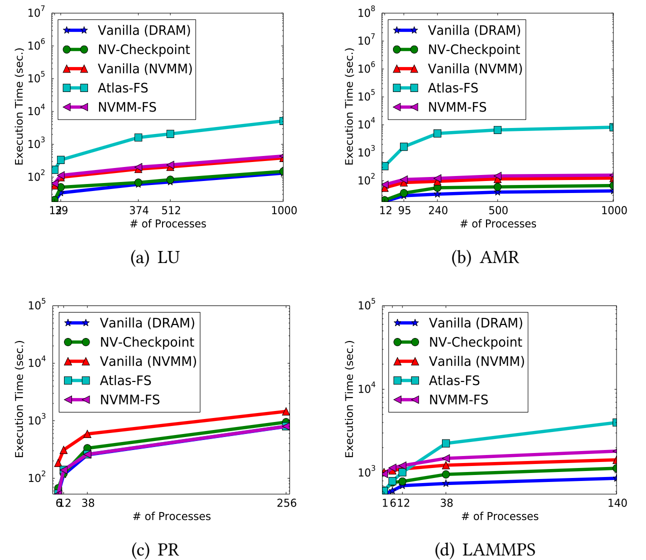

Figure 8: The execution time of applications in the weak-scaling experiments. A logarithmic y-axis is used for plotting.

We study the weak scaling of the applications in this section. We measure their execution time while increasing the problem sizes and the number of processors. For *LU*, the dimensions of arrays are increased from 2,750*2,750 to 31,500*31,500. For *AMR*, the number of elements in a mesh is increased from 800K to 130M. For *PR*, the number of nodes is increased from 282K to 129M. For *LAMMPS*, the number of atoms is increased from 500K to 253M. Figure 8 shows the execution time with NV-Checkpoint compared to other implementations. We have three observations. (1) None of them achieves optimal speedup. It is because of the communication overhead in the programs using the MPI programming model. For example, the ratio of communication time to the total execution time of *LU* is increased from 0.3% to 21.6% when the number of processes is increased from 6 to 1000. For *LU*, the MPI communication time is mostly spent on distributing blocks from master to slave processes and collecting computed data from slave to master processes. Similarly, we observe the same trend for other programs.

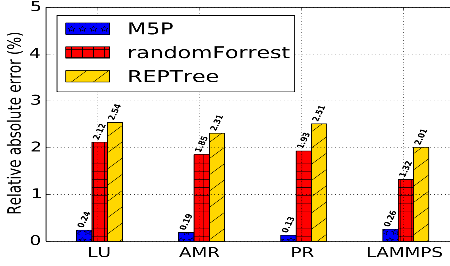

Figure 9: Cost model accuracies on 10-fold cross-validation.

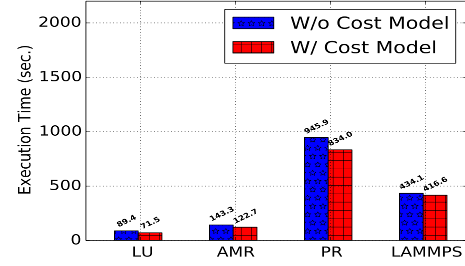

Figure 10: Execution time without and with using the cost model to determine persistence interval.

For example, *AMR* requires 51% of communication time for tree partitioning when the number of processes is 1000. (2) The implementation using NVMM achieves similar weak scaling as that using DRAM. All the implementations scale well except with the Atlas file system. It scales poorly when the file size of checkpoints cannot be effectively buffered in the cache of file servers. (3) The execution time with NV-Checkpoint is 40% longer on average than that with only DRAM, while NVMM write and read latency are 150% and 67% longer than DRAM write and read latency respectively. This is because the dynamic layout transformation (discussed in Section 5) supported by NV-Checkpoint runtime reduced the number of DRAM writes by 50%, 39%, and 8% for *LU*, *AMR*, and *PR*, respectively. We did not partition the arrays for *LAMMPS* because the array elements have an equal access frequency.

6.4 Accuracy and Effectiveness of Cost Model

An important component of NV-Checkpoint runtime system is its cost model to produce desired persistence intervals. We evaluate its model in this section. For each application, we randomly partition the data structures (e.g., octrees in *AMR*). We then trigger the $persistent()$ function and record its execution time T_p^i along with its corresponding data features FV^i (described in Table 1). We collected a total of 70,000 tuples of FV^i and T_p^i for each application. 60,000 of them are used for training machine learning models and the rest are used for testing. We studied the prediction accuracy of the three models available in Weka for data classification [18], including M5P, randomForrest, and REPTree. We measure their accuracies using relative absolute error (RAE), which is defined as $\frac{\sum_{i=1}^{N}|\hat{\theta}_i-\theta_i|}{\sum_{i=1}^{N}|\bar{\theta}_i-\theta_i|}$ (θ_i is the real value. $\hat{\theta}_i$ is corresponding predicted value. $\bar{\theta}_i$ is the mean value of θ). As shown in Figure 9, the RAE of M5P is 89% smaller than that of randomForrest and REPTree. This explains why we chose M5P in all the experiments.

With the cost model, we further evaluate its impact in determining a desired persistence interval at runtime. In the experiment, we instrumented the programs to have better coverage of different CPU burst lengths of computing phases in parallel programs. Specifically, if the computing time of each iteration is T_{comp} in the existing applications, we will inject additional computing time T_{inject}, which is randomly selected within the range of $[0, 5 * T_{comp}]$ in the instrumented programs. All results reported here are measured by executing the instrumented programs.

We first run the instrumented programs without the cost models. In this setup, the $persistent()$ function is executed at the end of every iteration even though the computing time might be smaller than the time of persistent operations. Then we run the instrumented programs with the cost models which are trained offline. The execution time of the programs is reduced by 15% on average with the cost model as shown in Figure 10.

6.5 Fault Injection Experiments

In this section, we compare the time to restart the applications after injected failures using NVMM and parallel file systems. Specifically, for *LAMMPS*, we use 140 processes and 18 million atoms. For *LU*, we use 240 processes and matrix size 20,480*20,480. For *AMR*, we use 220 processes and 125 million elements. For *PR*, we use 256 processes and 129 million vertices. In the experiments, we sent a SIGTERM signal in the middle of iteration 10 and forced it to execute failure recovery handlers in the programs. We implemented three types of handlers. (1) *NVMM (local)*: this handler recovers the memory states from the persistent data stored in NVMM on the same compute nodes. (2) *NVMM (network)*: this handler recovers the memory states from the persistent data stored in NVMM to a new set of compute nodes with data being transferred via the Gemini network on Titan. (3) *Checkpoint*: this handler recovers the memory states from checkpoint files stored in the Atlas file system.

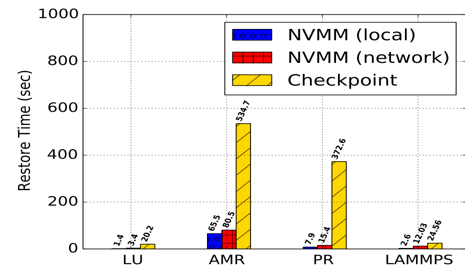

Figure 11: Time to restore memory states using NVMM via memory bus and using checkpoint files in Atlas.

The time to restore memory states is shown in Figure 11. Compared to *Checkpoint*, the restore time is reduced by 93%, 88%, 98%, and 90% for *LU*, *AMR*, *PR*, and *LAMMPS* respectively via NVMM on the same compute nodes. This is because *Checkpoint* needs to read the data via slow I/O bus while NV-Checkpoint operates at memory bandwidth. In addition, the restore time is reduced by 83%, 85%, 96%, and 81% for *LU*, *AMR*, *PR*, *LAMMPS* respectively

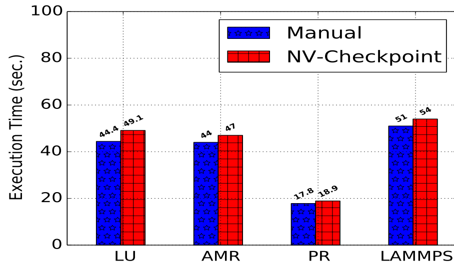

Figure 12: The execution time with NV-Checkpoint compared to those with the manual implementations.

Application	Directives	Total line changes
LU-decomposition	4	101
AMR	4	538
PageRank	4	94
LAMMPS	12	317

Table 3: A summary of the application of NV-Checkpoint.

via NVMM on a new set of compute nodes. The time of transferring data to the new node accounts for 73% of the restore time. And it becomes a dominant factor in the restore time when the size of persistent data to transfer is larger than 1.6 GB. Finally, the restore time using the Atlas file system is limited by I/O bandwidth.

6.6 Performance Compared to Manual Implementations

All the state-of-the-art checkpointing approaches using NVMM for scientific applications require substantial changes in the source code. These are done manually. In this section, we compare the performance of NV-Checkpoint to these manual implementations. Because the hand-written code does not have a runtime system for persistence management, NV-Checkpoint was instrumented to perform persist() as the hand-written code for every iteration. Specifically, we implement the checkpointing approaches using in-place versioning [24, 56] in the applications (i.e., LU-decomposition and LAMMPS) using the array data structure. We manually implement multi-version PMOctree [33] and NVGRAPH data structures [28] for checkpointing in the AMR and page ranking respectively. We use the setting described in Table 2 in the experiment. We use 1000, 1000, 256, and 256 processes for the LU-decomposition, AMR, page ranking, and LAMMPS respectively. As shown in Figure 12, the execution time of the applications with NV-Checkpoint is 7% on average longer than those with the hand-written code. The reason is that NV-Checkpoint needs to track data features and execute model inference at runtime for persistence management. It also incurs additional communication overhead for making a decision by the centralized master process. We also observe that the code with NV-Checkpoint achieves similar scalability.

Next, we study the performance of NV-Checkpoint compared to checkpointing using recompute for loop-based code [16]. We use LU-decomposition with 1000 processes in this experiment. For the checkpointing approach using recompute, the code runs only in NVMM. The execution time is 108.2 sec and 49.1 sec for recompute and NV-Checkpoint respectively. Checkpointing with recompute spends 1.2X more time because it runs only in NVMM while NV-Checkpoint places its ephemeral version in DRAM for computation and its persistent version in NVMM for checkpoint.

Table 3 summarizes the total number of added directives (excluding #pragma nvcp) and total number of added lines by the compiler to the source code. It clearly supports the claim that NV-Checkpoint can easily transform the ephemeral data structures to crash-consistent data structures with a small number of directives.

7 RELATED WORK

A wide range of systems and libraries have been developed to provide checkpointing to address the resilience issues of HPC applications. The work closely related to NV-Checkpoint is discussed below.

Checkpointing Approaches. System-level checkpointing approaches write the memory states of the entire memory systems to storage systems [43] as snapshots. They are usually transparent to end-users. For large-scale HPC systems, writing system-level snapshots using parallel I/Os can easily overload a disk-based storage system and become a severe performance bottleneck [3, 59, 60]. Snapshot files can be compressed to reduce checkpointing overhead [25]. Moody et al. proposed a multi-level checkpointing system, which can reduce the amount of checkpointing I/Os which are directed to slow hard disks by temporarily storing them in DRAM and flash on compute nodes [30]. GPU snapshot was designed to reduce checkpointing cost using asynchronous checkpoint offloading from GPUs to hosts [27]. Chakraborty et al. proposed EREINIT to reduce checkpointing overhead for bulk-synchronous MPI applications [9] by implementing fault-tolerance in low-level software layers. Application-level checkpointing approaches save only the main data structures and their metadata for checkpointing [6]. For example, the LAMMPS code saves the arrays which record the properties of *atoms* used in simulations [39] in snapshots. Its overhead is much smaller than system-level approaches. Elnawawy et al. reduce checkpointing overhead using recompute of loop-based code [16]. But this approach was only proposed for array data structures.

Algorithm-based fault tolerance can be used to detect and correct errors for applications using matrix computations [56, 58]. But it is difficult to extend them to other types of data structures.

Simply replacing disks with NVMM may reduce the checkpointing overhead because of its much lower read/write latency than disks. Caulfield et al. studied the impact of non-volatile memory on scientific applications [8]. Dong et al. proposed a hybrid local/global checkpointing mechanism leveraging 3D PCRAM [13]. Kannan et al. designed a system-level checkpointing mechanism using virtual memory of operating systems [26]. However, none of them exploits the byte-addressability of NVMM. Most recently, multi-version data structures have been used for application-level checkpointing in NVMM systems. Wu et al. designed NVMM-aware algorithms that store an additional persistent version of main data structures as a checkpoint in NVMM along with its ephemeral version in DRAM [24, 56]. The persistent version can be used for both

checkpoints and computation leveraging the byte-addressability of NVMM. PMOctree was designed as a crash-consistent multi-version octree data structure for adaptive meshing [32, 33]. NVGRAPH was designed as a multi-version graph data structure for NVMM-aware evolving graph computation [28]. However, all of them focused on manual transformation of main data structures and require programmers to have a deep understanding of NVMM memory models. The changed code layout for exploring NVMM makes the source code difficult to read and understand. In this paper, we design NV-Checkpoint to serve two purposes: (1) automatically transforming ephemeral data structures to its corresponding NVMM-aware crash-consistent version with the aid of compilers, dramatically reducing the burden of programmers; and (2) implementing a runtime system which determines when to create a persistent version of a data structure using machine learning models.

Compiler-Aided Data Structure Transformation. Compiler-aided approaches have been widely used in automatically transforming sequential data structures to concurrent data structures that are aware of UMA [5, 57] or NUMA [7]. For example, the node replication technique was proposed by Calciu et al. to produce NUMA-aware concurrent data structures satisfying linearizability using shared logs [7]. To the best of our knowledge, NV-Checkpoint is the first compiler-aided approach that is designed to transform ephemeral data structures to its corresponding crash-consistent multi-version data structures in NVMM systems. To make persistence very simple to use, SoftPM was proposed to provide *orthogonal persistence* for sequential data structures [17]. It identifies structures of in-memory objects (e.g., linked lists) using static analysis and persistent arbitrary data structures using containers on storage devices. Compared to SoftPM, NV-Checkpoint uses multi-version data structures as the container of ephemeral data structures and targets on both sequential and parallel/distributed data structures. In addition, NV-Checkpoint automatically tunes the performance of NVMM-aware data structures at runtime by managing the persistence interval and the layout of data structures placed in a hybrid of DRAM and NVMM considering the characteristics of HPC applications and NVMMs.

Other Work in Context. NV-Checkpoint's annotation API uses C/C++ compiler front-end Clang [10] and Java parser for source-to-source translation and source code analysis. Other compilers have been designed for different programming languages. For example, ROSE compiler infrastructure [11] offers analysis tools for large-scale Fortran, C, OpenMP, and UPC applications. ROSE uses a uniform abstract syntax tree (AST) to represent source code using a high-level intermediate code, while Clang uses a lower-level representation to simplify the intermediate code.

8 CONCLUSION AND FUTURE WORK

We analyze the root causes of crash inconsistency of scientific applications using ephemeral data structures (i.e., arrays, quad/octrees, and graphs) upon fail-stop failures in NVMM systems. We then propose, implement, and evaluate a general framework, called NV-Checkpoint, which can automatically transform source code for enforcing crash consistency using multi-version data structures with the aid of compilers. Its runtime system uses such data structures to provide crash consistency because at least one version of its data is immutable until a newer version becomes persistent. We use a machine learning based approach to determine a desired persistence interval considering persistence overhead and recomputing time. For the evaluation of NV-Checkpoint, we use four representative real-world scientific applications: LU-Decomposition, adaptive mesh refinement, page ranking, and molecular simulation using LAMMPS. The experimental results show that the performance of annotated programs using NV-Checkpoint is commensurate with the version using the corresponding ephemeral data structures. It scales well up to 1000 processes on Titan. It offers up to 121X speedup of program execution time and 16X speedup of restore time compared to those using parallel file systems on the Titan supercomputer. Finally, NV-Checkpoint significantly reduces programmers' burden of using NVMM in HPC systems.

Our work suggests several avenues for future research, including (1) automatically identifying variables that may suffer from data inconsistency upon failures using static analysis, (2) a richer set of annotations, (3) characterizing different types of applications and building uniform machine-learning models for persistence management, and (4) evaluation of NV-Checkpoint with scientific applications using other popular data structures (e.g., B-trees, hash tables) on other HPC platforms, especially those with real NVMM hardware.

ACKNOWLEDGMENT

We are grateful to the anonymous reviewers. This research was supported by US National Science Foundation under CNS 1906541. This work was also funded in part by WSU Vancouver Research Mini Grant and the Key Scientific Technological Innovation Research Project by Ministry of Education in China.

REFERENCES

[1] 2003. Gerris: a tree-based adaptive solver for the incompressible Euler equations in complex geometries. *J. Comput. Phys.* 190, 2 (2003), 572 – 600.

[2] 2019. pmdk: Persistent Memory Development Kit. https://github.com/pmem/pmdk.

[3] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. 2009. PLFS: A Checkpoint Filesystem for Parallel Applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*.

[4] Amanda Bienz, Robert D. Falgout, William Gropp, Luke N. Olson, and Jacob B. Schroder. 2016. Reducing Parallel Communication in Algebraic Multigrid through Sparsification. *SIAM J. Scientific Computing* 38 (2016).

[5] Silas Boyd-wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Technical Report TR-2014-019, MIT CSAIL. OpLog: a library for scaling update-heavy data structures.

[6] Greg Bronevetsky, Daniel Marques, Keshav Pingali, Peter Szwed, and Martin Schulz. 2004. Application-Level Checkpointing for Shared Memory Programs. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*. Association for Computing Machinery, New York, NY, USA, 235âĂŞ247. https://doi.org/10.1145/1024393.1024421

[7] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. 2017. Black-box Concurrent Data Structures for NUMA Architectures. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*.

[8] 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)*.

[9] Sourav Chakraborty, Ignacio Laguna, Murali Emani, Kathryn Mohror, Dhabaleswar K. Panda, Martin Schulz, and Hari Subramoni. 2020. EReinit: Scalable and efficient fault-tolerance for bulk-synchronous MPI applications. *Concurrency*

and Computation: Practice and Experience 32, 3 (2020), e4863. https://doi.org/10.1002/cpe.4863 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4863 e4863 cpe.4863.
[10] clang: a C language family frontend for LLVM. 2019. http://clang.llvm.org/.
[11] ROSE compiler infrastructure. 2019. http://rosecompiler.org/.
[12] MAWI Datasets. 2019. https://graphchallenge.mit.edu/data-sets.
[13] Xiangyu Dong, Naveen Muralimanohar, Norm Jouppi, Richard Kaufmann, and Yuan Xie. 2009. Leveraging 3D PCRAM Technologies to Reduce Checkpoint Overhead for Future Exascale Systems. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC'09)*. Association for Computing Machinery, New York, NY, USA, Article Article 57, 12 pages. https://doi.org/10.1145/1654059.1654117
[14] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. 1989. Making Data Structures Persistent. *J. Comput. Syst. Sci.* 38, 1 (Feb. 1989).
[15] H. Elnawawy, M. Alshboul, J. Tuck, and Y. 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)*. 318–329. https://doi.org/10.1109/PACT.2017.58
[16] 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)*. 318–329. https://doi.org/10.1109/PACT.2017.58
[17] Jorge Guerra, Leonardo Marmol, Daniel Campello, Carlos Crespo, Raju Rangaswami, and Jinpeng Wei. 2012. Software Persistent Memory. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*.
[18] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. 2009. The WEKA Data Mining Software: An Update. *SIGKDD Explor. Newsl.* 11, 1 (Nov. 2009), 10–18. https://doi.org/10.1145/1656274.1656278
[19] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. 2014. Chronos: A Graph Engine for Temporal Graph Analysis. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys'14)*.
[20] 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
[21] Amin Hassani, Anthony Skjellum, Purushotham V. Bangalore, and Ron Brightwell. 2015. Practical Resilient Cases for FA-MPI, a Transactional Fault-Tolerant MPI. In *Proceedings of the 3rd Workshop on Exascale MPI (ExaMPI âĂŽ15)*. Association for Computing Machinery, New York, NY, USA, Article Article 1, 10 pages. https://doi.org/10.1145/2831129.2831130
[22] Stephen D. Hoath. 2016. *Fundamentals of inkjet printing: the science of inkjet and droplets*. Wiley-VCH Verlag GmbH &Co.
[23] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. 2014. NVRAM-aware Logging in Transaction Systems. *Proc. VLDB Endow.* 8, 4 (Dec. 2014).
[24] Yingchao Huang, Kai Wu, and Dong Li. 2017. High Performance Data Persistence in Non-Volatile Memory for Resilient High Performance Computing. arXiv:cs.DC/1705.00264
[25] Dewan Ibtesham, Kurt B Ferreira, and Dorian Arnold. 2015. A Checkpoint Compression Study for High-Performance Computing Systems. *Int. J. High Perform. Comput. Appl.* 29, 4 (Nov. 2015), 387âĂŞ402. https://doi.org/10.1177/1094342015570921
[26] Sudarsun Kannan, Ada Gavrilovska, Karsten Schwan, and Dejan Milojicic. 2013. Optimizing Checkpoints Using NVM as Virtual Memory. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. 29–40.
[27] Kyushick Lee, Michael B. Sullivan, Siva Kumar Sastry Hari, Timothy Tsai, Stephen W. Keckler, and Mattan Erez. 2019. GPU Snapshot: Checkpoint Offloading for GPU-Dense Systems. In *Proceedings of the ACM International Conference on Supercomputing (ICS âĂŽ19)*. Association for Computing Machinery, New York, NY, USA, 171âĂŞ183. https://doi.org/10.1145/3330345.3330361
[28] Soklong Lim, Zaixin Lu, Bin Ren, and Xuechen Zhang. 2019. Enforcing Crash Consistency of Evolving Network Analytics in Non-Volatile Main Memory Systems. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT'19)*.
[29] Peter Macko, Virendra J. Marathe, Daniel W. Margo, and Margo I. Seltzer. 2015. LLAMA: Efficient graph analytics using Large Multiversioned Arrays. In *2015 IEEE 31st International Conference on Data Engineering (ICDE'15)*.
[30] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. 2010. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–11. https://doi.org/10.1109/SC.2010.18
[31] Mark E. J. Newman. 2010. *Networks : An introduction.*
[32] Bao Nguyen, Hua Tan, Kei Davis, and Xuechen Zhang. 2018. Persistent Octrees for Parallel Mesh Refinement Through Non-Volatile Byte-Addressable Memory. In *IEEE Transactions on Parallel and Distributed Systems*.
[33] Bao Nguyen, Hua Tan, and Xuechen Zhang. 2017. Large-scale Adaptive Mesh Simulations Through Non-Volatile Byte-Addressable Memory. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'17)*.
[34] 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)*.
[35] Burcu Ozcelik Mutlu, Gokcen Kestor, Joseph Manzano, Osman Unsal, Samrat Chatterjee, and Sriram Krishnamoorthy. 2018. Characterization of the Impact of Soft Errors on Iterative Methods. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*. 203–214. https://doi.org/10.1109/HiPC.2018.00031
[36] ScaLAPACK-Scalable Linear Algebra Package. 2019. http://www.netlib.org/scalapack/.
[37] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank Citation Ranking: Bringing Order to the Web.* Technical Report 1999-66. Stanford InfoLab. http://ilpubs.stanford.edu:8090/422/ Previous number = SIDL-WP-1999-0120.
[38] Roger Pearce, Maya Gokhale, and Nancy M. Amato. 2010. Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory. 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.2010.34
[39] Steve Plimpton, Roy Pollock, and Mark Stevens. 2007. Particle-Mesh Ewald and rRESPA for Parallel Molecular Dynamics Simulations. In *Proc of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*.
[40] John R. Quinlan. 1992. Learning With Continuous Classes. In *5th Australian Joint Conference on Artificial Intelligence*. World Scientific, 343–348.
[41] I. S. Reed and G. Solomon. 1960. Polynomial Codes Over Certain Finite Fields. *J. Soc. Indust. Appl. Math.* 8, 2 (1960), 300–304.
[42] Sumalee Sangamuang, Pruet Boonma, and Lai Lai W. Kyii. 2015. An algorithm to improve MPI-PageRank performance by reducing synchronization time. In *2015 International Computer Science and Engineering Conference (ICSEC)*. 1–4. https://doi.org/10.1109/ICSEC.2015.7401454
[43] Bianca Schroeder and Garth A Gibson. 2007. Understanding failures in petascale computers. *Journal of Physics: Conference Series* 78 (jul 2007), 012–022. https://doi.org/10.1088/1742-6596/78/1/012022
[44] Gerris Flow Solver. 2019. http://gfs.sourceforge.net/wiki/index.php/Main_Page.
[45] Hari Sundar, Rahul S. Sampath, Santi S. Adavani, Christos Davatzikos, and George Biros. 2007. Low-constant parallel algorithms for finite element simulations using linear octrees. In *Supercomputing, 2007. SC '07. Proceedings of the 2007 ACM/IEEE Conference on*. 1–12.
[46] Hari Sundar, Rahul S Sampath, and George Biros. 2008. Bottom-up construction and 2: 1 balance refinement of linear octrees in parallel. *SIAM Journal on Scientific Computing* 30, 5 (2008), 2675–2708.
[47] Lustre File Systems. 2019. https://lustre.org/.
[48] Hua Tan, Eric Torniainen, David P. Markel, and Robert N. K. Browning. 2015. Numerical simulation of droplet ejection of thermal inkjet printheads. *International Journal for Numerical Methods in Fluids* 77 (March 2015), 544–570.
[49] Titan. 2019. https://www.olcf.ornl.gov/titan/.
[50] Suleyman Tosun, Vahid B. Ajabshir, Ozge Mercanoglu, and Ozcan Ozturk. 2015. Fault-Tolerant Topology Generation Method for Application-Specific Network-on-Chips. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34, 9 (Sep. 2015), 1495–1508. https://doi.org/10.1109/TCAD.2015.2413848
[51] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST'11)*.
[52] Solving Linear Systems via LU Factorization. 2019. http://www.netlib.org/utk/papers/pblas/node21.html.
[53] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. 2014. Aerie: Flexible File-system Interfaces to Storage-class Memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*.
[54] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. *SIGPLAN Not.* 47, 4 (March 2011), 91–104.
[55] Jingjing Wang and Magdalena Balazinska. 2017. Elastic Memory Management for Cloud Data Analytics. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 745–758. https://www.usenix.org/conference/atc17/technical-sessions/presentation/wang
[56] Panruo Wu, Dong Li, Zizhong Chen, Jeffrey S. Vetter, and Sparsh Mittal. 2016. Algorithm-Directed Data Placement in Explicitly Managed Non-Volatile Memory. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC'16)*. Association for Computing Machinery, New York, NY, USA, 141–152. https://doi.org/10.1145/2907294.2907321
[57] Lingxiang Xiang and Michael Lee Scott. 2013. Compiler Aided Manual Speculation for High Performance Concurrent Data Structures. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*.

[58] Shuo Yang, Kai Wu, Yifan Qiao, Dong Li, and Jidong Zhai. 2017. Algorithm-Directed Crash Consistence in Non-volatile Memory for HPC. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*.

[59] Xuechen Zhang, Kei Davis, and Song Jiang. 2010. IOrchestrator: Improving the Performance of Multi-node I/O Systems via Inter-Server Coordination. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'10)*.

[60] Xuechen Zhang, Kei Davis, and Song Jiang. 2011. QoS Support for End Users of I/O-intensive Applications using Shared Storage Systems. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'11)*.

[61] Xuechen Zhang, Kei Davis, and Song Jiang. 2012. Opportunistic Data-driven Execution of Parallel Programs for Efficient I/O Services. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. 330–341. https://doi.org/10.1109/IPDPS.2012.39

[62] Xuechen Zhang, Song Jiang, and Kei Davis. 2009. Making resonance a common case: A high-performance implementation of collective I/O on parallel file systems. In *2009 IEEE International Symposium on Parallel Distributed Processing*. 1–12. https://doi.org/10.1109/IPDPS.2009.5161070

[63] Xuechen Zhang, Ujjwal Khanal, Xinghui Zhao, and Stephen Ficklin. 2017. Making sense of performance in in-memory computing frameworks for scientific data analysis: A case study of the spark system. *J. Parallel and Distrib. Comput.* (2017).