Compiler-support for Critical Data Persistence in NVM

REEM ELKHOULY, Tanta University, Egypt and Waseda University, Japan

MOHAMMAD ALSHBOUL, North Carolina State University, USA

AKIHIRO HAYASHI, Georgia Institute of Technology, USA

YAN SOLIHIN, University of Central Florida, USA

KEIJI KIMURA, Waseda University, Japan

Non-volatile Main Memories (NVMs) offer a promising way to preserve data persistence and enable computation recovery in case of failure. While the use of NVMs can significantly reduce the overhead of failure recovery, which is the case with High-Performance Computing (HPC) kernels, rewriting existing programs or writing new applications for NVMs is non-trivial. In this article, we present a compiler-support that automatically inserts complex instructions into kernels to achieve NVM data-persistence based on a simple programmer directive. Unlike checkpointing techniques that store the whole system state, our technique only persists user-designated objects as well as some parameters required for safe recovery such as loop induction variables. Also, our technique can reduce the number of data transfer operations, because our compiler coalesces consecutive memory-persisting operations into a single memory transaction per cache line when possible.

Our compiler-support is implemented in the LLVM tool-chain and introduces the necessary modifications to loop-intensive computational kernels (e.g., TMM, LU, Gauss, and FFT) to force data persistence. The experiments show that our proposed compiler-support outperforms the most recent checkpointing techniques while its performance overheads are insignificant.

CCS Concepts: • **Software and its engineering** → **Source code generation; Runtime environments; Main memory**; • **Hardware** → *Emerging languages and compilers*;

Additional Key Words and Phrases: Compiler-support, NVM, data persistence, valid recovery

ACM Reference format:

Reem Elkhoully, Mohammad Alshboul, Akihiro Hayashi, Yan Solihin, and Keiji Kimura. 2019. Compiler-support for Critical Data Persistence in NVM. *ACM Trans. Archit. Code Optim.* 16, 4, Article 54 (December 2019), 25 pages.

<https://doi.org/10.1145/3371236>

New paper, not an extension of a conference paper.

Authors' addresses: R. Elkhoully, Tanta University, Al-Geish St. Tanta, Gharbeya, 31512, Egypt, Waseda University, 27 Waseda-machi, Shinjuku-ku, Tokyo, 162-0042, Japan; email: reem_elkhoully@f-eng.tanta.edu.eg; M. Alshboul, North Carolina State University, 2931 Ligon St, Raleigh, NC, 27607, USA; email: maalshbo@ncsu.edu; A. Hayashi, Georgia Institute of Technology, 266 Ferst Drive, Atlanta, GA, 30332, USA; email: ahayashi@gatech.edu; Y. Solihin, University of Central Florida, 4364 Scorpius St. Orlando, FL, 32816, USA; email: yan.solihin@ucf.edu; K. Kimura, Waseda University, 27 Waseda-machi, Shinjuku-ku, Tokyo, 162-0042, Japan; email: keiji@waseda.jp.

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 the author(s) 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](https://permissions.acm.org).

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

1544-3566/2019/12-ART54

<https://doi.org/10.1145/3371236>

1 INTRODUCTION

NVMs offer byte-addressable, non-volatile memories along with DRAM-comparable speed [32]. Since NVMs can preserve the data from the cache coherence domain across failures, the use of NVMs can be a prospective approach for fault tolerance. Common fault tolerance techniques such as checkpointing and logging incur high overheads and large amounts of data transfer [31, 48]. Editing kernels to selectively make critical data persistent imposes significant burdens on programmers, because the programming for NVM is non-trivial [36]. Furthermore, as the number of processor cores in HPC environments grows, the mean time between failures (MTBF) shrinks [28], which means HPC resilience requires maintaining data across more frequent failures to avoid corrupted results. However, taking large checkpoints and retrieving them after such frequent failures will be troublesome [39].

To reiterate, the most common fault tolerance technique in HPC is checkpoint and restart. The application's critical data is captured periodically and saved to disks. Whenever a failure happens, the last stored state is recovered and computations are resumed [19]. The preliminary technique was improved to consume less processing time and storage space. However, scientific applications that operate on massive data still suffer from significant data transfer time. Frequent checkpointing would imply larger data to store, while scarce checkpointing would cause higher recovery overhead [47]. Other techniques are compiler-based, which adopt data transfer following to every store instructions or, sometimes, a block of instructions. Moreover, techniques that require extensive analysis and sometimes expensive hardware exist [26].

Lazy persistency [3] is a persistence technique that does not insert any additional NVM instructions for data persisting. Instead, the normal data write-back operations for coherence are considered sufficient. Meanwhile, a checksum is calculated to discover persistence errors in NVM and recompute inconsistent results. In References [2, 17], recomputing is used for a chunk of data whose consistency is questionable. Unlike lazy persistency, NVM instructions are inserted to force data flushing from cache to persistent memory periodically. If transferring a chunk of data is interrupted by a failure, then this data are discarded and recomputed to avoid the risk of inconsistency. This data can be precisely determined, because the values of induction variables are eagerly persisted. While these techniques succeeded to reduce the overheads of checkpointing data to NVM, the process of hand-tailoring code segments to recompute inconsistent data or to persist recently computed data is complex and error-prone.

In this article, we present an HPC-software fault tolerance technique that automatically tailors the data persistence scheme along with small overhead. Our system is based on the previous work from References [2, 17], where we automate the manual code modifications to avoid complexity and erroneous code updates. Further, we propose additional optimizations that improve the performance of the resultant programs after modifications. We implement a compiler-support that only persists enough data for restoring a kernel after failure. Moreover, it tunes the granularity at which the critical data is made durable to decrease data transfer operations and time overhead. Our compiler-support targets scientific applications' kernels that spend massive time executing loop nests. We offer a programmer directive whereby the compiler-support behavior is tuned for different kernels. This directive is used to provide information about (1) the critical data object to be persisted and (2) the nesting level at which the persistence operations are performed. According to this information, complex memory persistence instructions are inserted into the code to guarantee consistency and failure recovery. Meanwhile, the compiler-support is designed to reduce the consecutive data transfer to NVM by adopting a single operation for a cache line block instead of dedicating an individual operation for every element in the target data object. Even if the data in the cache are not aligned, our technique can properly determine the cache blocks to persist.

Although the data in the forehead cache line may be unaligned, they are handled by setting the bounds of the loop to precisely address the required data as discussed subsequently.

This work focuses on single loop-nest based kernels that process multi-dimensional arrays where array accesses are statically analyzable. The current implementation only works for countable nested loops assuming unique induction variables. For simplicity, we limited the persisted critical data to a single (1D or 2D) array per loop-nest in the current implementation. In case of imperfect nested loops, although the proposed algorithm detects the accessed chunk of elements from the induction variables' boundaries, the programmer may have to explicitly provide the access range in the directive if the kernel has some irregular stores.

Our main contributions can be summarized as follows:

- A compiler-support that is capable of generating complex NVM data persisting code for loop-based kernels (Section 3). The generated code checkpoints the critical data regularly and also checkpoints the values of loop induction variables to enable calculations resumption after recovery (Section 3.4).
- Our compiler-support facilitates data persistence achievement, because it requires no special hardware and avoids complicated manual code editing (Section 3).
- We developed a simple directive, thereby the programmer can provide hints about the critical data needs persisting and the suggested granularity at which persisting should occur. This helps to avoid the exhaustive analysis and reduce the number of redundant data transfer operations (Section 3.1).
- Our scheme uses a single data transfer operation per cache line instead of executing a separate operation for every data element in the targeted critical data structure (Section 3.4).
- Our compiler-support can detect and transfer data from partial critical ranges—i.e., triangle, diagonal, and partially changed matrices (Section 4.2)—instead of a systematic data transfer procedure that walks through a complete range of elements in the target data object. We also offer the programmer the option to explicitly determine the range in the directive whereby imperfect nested loops and other range detection difficulties can be bypassed.
- We show that our compiler-support works for four important loop-based computational kernels even if the ranges to be persisted are non-contiguous (i.e., scattered) data access in FFT, or incomplete (i.e., triangular) matrices in LU and Gauss (Section 5). Currently, we consider only single-threaded implementations of these kernels, while multi-threading support will be considered in future work.

The rest of this article is organized as follows: Section 2 overviews the necessary background and previous work. Section 3 demonstrates the design of our compiler-support and the necessary NVM instructions used. Section 4 explains the benchmarks we used for the evaluation of our method and explains the different styles of data access in each of them. In Section 5, we expose the performance statistics of the tested kernels when modified by our compiler-support for NVM data persistence and the experimental setup. Section 6 discusses the cache behavior and an example of the checkpointing overheads that influence the experimented kernels. A survey of the related work in literature is presented in Section 7. Finally, we conclude the presented work and discuss future work in Section 8.

2 BACKGROUND

As previously mentioned, our technique is based on the work presented in Reference [17]. In that study, an efficient checkpointing technique for NVMs that is applicable for loop-based codes was developed. The technique neither relies on durable transactions to store the updated data to NVM nor creates a durable log, a copy of the original matrix, saving storage space and copy time. Instead, by using non-atomic instructions, it persists the updated matrix as the computations proceed.

```

1  for (kk=0; kk<n; kk+=bsize) {
2      for (ii=0; ii<n; ii+=bsize) {
3          for (jj=0; jj<n; jj+=bsize) {
4              for (i=ii; i<(ii+bsize); i++) {
5                  for (j=jj; j<(jj+bsize); j++) {
6                      sum = c[i][j];
7                      for (k=kk; k<(kk+bsize); k++)
8                          sum += a[i][k]*b[k][j];
9                      c[i][j] = sum;
10                     } //end of for j
11                 } //end of for i
12             } //end for jj
13         for (l=ii; l<(ii+bsize); l++){
14             for (r=0; r<n; r+=CLELEMENTS)
15                 CLFLUSHOPT(&c[l][r]);
16         }
17         SFENCE;
18         lastPersistedII = ii;
19         CLFLUSHOPT(&lastPersistedII);
20         SFENCE;
21     } //end of for ii
22     lastPersistedKKLog = lastPersistedKK;
23     CLFLUSHOPT(&lastPersistedKKLog);
24     SFENCE;
25     insideTxKK = true;
26     CLFLUSHOPT (&insideTxKK);
27     SFENCE;
28     lastPersistedII = -bsize;
29     lastPersistedKK += bsize;
30     CLFLUSHOPT (&lastPersistedII);
31     CLFLUSHOPT(&lastPersistedKK);
32     SFENCE;
33     insideTxKK = false;
34     CLFLUSHOPT(&insideTxKK);
35     SFENCE;
36 } //end of for kk

```

(a) Normal execution with data persisting.

```

1  //Step1: Zero out the cells to be recomputed.
2  // bsize of rows that were NOT persisted.
3  firstII = lastPersistedII;
4  lastII = lastPersistedII + bsize;
5  for (i= firstII; i< lastII; i++) {
6      for (j=0; j<n; j++)
7          c[i][j] = 0;
8  }
9  //Step2: Recompute bsize of rows from kk=0
10 // until the kk iteration of failure.
11 for (kk=0; kk<=lastPersistedKK; kk+=bsize) {
12     for (jj=0; jj<n; jj+=bsize) {
13         for (i= firstII; i< lastII; i++) {
14             for (j=jj; j<(jj+bsize); j++)
15                 Perform_Multiplication(i, j)
16         }
17     }
18     //Step3: Make the recomputed cells durable
19     for (i= firstII; i< lastII; i++) {
20         for (j=0; j<n; j+=CLELEMENTS)
21             CLFLUSHOPT(&c[i][j])
22     }
23     SFENCE;

```

(b) Recovery with recompute.

Fig. 1. From Reference [17], tiled matrix multiplication including data persisting and recovery with recompute.

Periodically, the recently updated part of the matrix is persisted, and the computations are suspended until the data persistence operation is complete. Compared to persisting the whole computed data after the computations are done, this technique has three advantages. First, performance gain, because data is more likely to exist in the cache when flushing to NVM. Second, the excess data transfer to NVM can be hidden by overlapping with computations of other parts of the data. Third, if the program fails, the size of data that was computed but not made durable yet will be relatively small and can be recovered faster. A consistent state cannot be guaranteed, because some elements will be inconsistent if a failure happens before making all the updated elements durable. To handle this, the induction variables of loops are updated atomically. Therefore, the part of the matrix that is at risk of being inconsistent can be precisely identified. When a failure occurs, the corresponding part of the matrix is discarded then recomputed to restore the pre-failure matrix state. The example shown in Figure 1 demonstrates data persisting and recovery processes in the tiled matrix multiplication (TMM) kernel. Figure 2 is an auxiliary representation of the progress of

Fig. 2. Tiled matrix multiplication computation illustration.

computations in the 6-level loop nest of TMM showing how the indices proceed within two input (a , b) and one output (c) matrices.

The overhead of data persistence is decreased during computation in comparison with flushing every updated data element immediately after each store, and contrariwise, the required work for failure-recovery is increased. Figure 1(a) shows the data persisting code, Lines 13 to 35, in the TMM kernel. The two-loop nest in Line 13 of Figure 1(a) is responsible for persisting the recently updated part of the result matrix c followed by an instruction, Line 17, that holds following store instructions to the same cache lines till persisting is done. If a failure occurs before the completion of the persisting, then this relatively small part of the result matrix is discarded and recomputed. Lines 25 to 33 are an atomic transaction that updates the designated induction variables to guarantee consistency of the persisted part of the matrix.

In case of failure, only persisted data can be retrieved, while data that was not made durable is lost. Consequently, in the given TMM example, a set of rows in the result c matrix that is equivalent to a single row of tiles is at risk of being incompletely persisted. These rows can be easily identified from the most recent value of ii that was made durable before failure, *lastPersistedII* in Figure 1(a) Line 18. Because cells are computed by accumulating on their previous values, when it is started in recovery mode, the kernel initially resets the designated cells to zeros as in Figure 1(b), Lines 5 to 8. Thereafter, the computation of the cells within the affected region, a few cells of the large matrix, is repeated using the code in Lines 11 to 17, which is expected to be done fast. Finally, the recovered data should be persisted to NVM entirely, Lines 19 to 23, before normal execution resumption. This recovery technique can recover from a recent checkpoint instead of repeating all computations from the start if no checkpoints were stored. However, it reduces the cost of computed-data durability by taking infrequent checkpoints instead of persisting every store operation immediately.

That technique is brilliant in waiving the majority of the persisting overheads from the computations. However, creating and inserting such complex code manually, besides being difficult and time-consuming, is error-prone. Here arises the need for automating this approach through a compiler-support that can generate and insert this code with minimum programmer involvement. Further performance optimizations are also introduced through the compiler-support, including fewer data transfer operations and cache alignment handling.

3 NVM COMPILER-SUPPORT

Data persistence in NVM systems is important for HPC applications to resume computation after recovering from failures. The proposed compiler-support is implemented as a compiler pass in the LLVM backend in addition to a custom pragma directive in the Clang C/C++ frontend. The compiler pass is responsible for designated code analysis and NVM supporting code generation in

```

1 #pragma clang loop persist (targetArray[rowStart:
    rowEnd][colStart:colEnd], targetLoopIV)
2 for(targetLoopIV){
3     ...
4     for (...) {
5         //targetArray is modified
6         //in this loop or deeper.
7         ...
8         for (...) {
9             ...
10            ...
11        }
12    }
13    // Data persistence handling code
14    //synchronization
15    //targetLoopIV logged, log persisted
16    //targetLoop IV log atomic update
17 }

```

(a) Persisting data at the outermost loop.

```

1 #pragma clang loop persist (targetArray[rowStart:
    rowEnd][colStart:colEnd], targetLoopIV)
2 for(outerLoopIV){
3     ...
4     for(targetLoopIV){
5         ...
6         for (...) {
7             //targetArray is modified
8             //in this loop or deeper.
9             ...
10        }
11        // Data persistence handling code
12        //synchronization
13        //targetLoopIV logged, log persisted
14    }
15    //outerLoopIV logged, log persisted
16    // target/outerLoop IVs logs atomic update
17 }

```

(b) Persisting data at the 2nd outermost loop.

Fig. 3. Pragma directive for data persistence in different nesting levels.

light of the hints provided by the programmer through the directive. The generated code for data persistence uses the NVM instructions *clflushopt* and *sfence* provided by Intel.

3.1 Custom Pragma Directive for Data Persistence

Prior approaches, for example References [13, 19], require tedious analysis to highlight the critical data structures that should be persisted to enable a valid failure-recovery. Furthermore, for a given application, it is non-trivial for compilers to appropriately decide the granularity and rate of the data transfer operations. Therefore, we propose a pragma directive that holds some information from the programmer; the name of the target data array where the range to be persisted is optionally appended if necessary, and the induction variable of the loop to which the flushing code will be appended. It simplifies NVM programming through a simple abstraction that allows the programmer to forward hints to the backend compiler pass.

The programmer adds this pragma directive just before the outermost loop of the loop-nest that dominates the computations and where the calculated data are written to the target data structure. The programmer can select a candidate loop in the loop-nest to be the target loop for inserting data persistence code according to the appropriate durable data granularity. The programmer should think about the balance between fast recovery and infrequent computation interrupts. If the programmer chooses to persist smaller parts of the data more often to keep the durable data up to date and to enable fast recovery, then it can interrupt the computations more frequently to transfer the data to NVMs. However, if he/she chooses to persist larger parts of the data less often to maintain computations continuity, this can cause slower recovery due to the risk of losing more data upon failures.

This programmer directive is implemented as a loop pragma directive in the Clang compiler, which is supported as a C/C++ frontend for LLVM. For example, the programmer may choose the outermost loop to persist the updated data by the end of every iteration at a coarse granularity or the second outermost loop for smaller granularity. Figure 3 shows a simple overview of how to use the pragma directive in a kernel to invoke data persistence handling in the outermost loop in

Fig. 4. Compiler-support for NVMs as an LLVM compiler pass.

Figure 3(a) or in the second outermost loop in Figure 3(b). While more complicated examples from common computation kernels are discussed later in this article, in this section, we highlight that the pragma directive holds the following *three* hints from the programmer:

- The outermost loop of the computation intensive loop-nest, which is defined by the location of the directive in the code.
- The critical data structure that requires durability and will be the target of data persistence instructions. As an option, the range to be persisted may be provided (in symbolic or numeric representation) to simplify the analysis in case of imperfectly nested loops.
- The induction variable of the target loop nesting level that would achieve a proper data persistence granularity.

The latter two hints are explicitly included as pragma arguments and then parsed and recorded as metadata of the outermost loop. The bounds of the designated range can be expressed either as numbers, variables' symbols that will inherit the values of their corresponding variables for every iteration at runtime, or a simple arithmetic expression such as $n-1$. The symbols used in identifying the range bounds and the induction variable of the target loop should be unique to eliminate ambiguity. In LLVM IR, metadata are attached to the designated loop's branch instruction in the latch block. The provided information is passed to the next stage of the compiler-support to guide code analysis and NVM data persistence code generation as shown in Figure 4.

3.2 Compiler Pass

In LLVM compiler infrastructure, analysis and optimization tools are implemented individually as a set of code passes [24]. We implemented our compiler-support that inserts the necessary instructions for data persistence in NVM as a stand-alone pass that can be invoked via command-line options. This arrangement allows the usage of this support along with any combination of a wide variety of efficient analysis and optimization passes. As shown in Figure 4, the proposed

compiler-support operates on the code in the intermediate representation (IR) form. It locates the target loop-nest that carries the programmer-provided information in its outermost loop's metadata.

The details of the compiler pass are described as pseudo-code in Algorithm 1. Besides a reference to the innermost loop, the other inputs include two arguments that are provided through the custom pragma directive: *targetLoopInductionVariable* and *targetArrayName*; as well as two command-line options: *isTiled* and *isPartial*. The programmer specifies *targetLoopInductionVariable* to determine the granularity of persisting critical data as the flushing code to be inserted in *targetLoopInductionVariable*'s loop. Also, the programmer specifies *targetArrayName* to determine the critical data structure that needs to be persisted. We assume that (1) induction variables in the same loop nest have unique names, (2) the critical data structure is a single or multi-dimensional array with affine accesses, and (3) the pragma directive is used to address a single critical array to be flushed within the corresponding loop nest. Also, the use of the two command-line options can decrease the overheads of unnecessary flush operations that flush possibly unchanged data. Specifically, when multiple rows in the target array are required to be flushed, setting *isTiled* would cause our compiler pass to construct a two-level loop structure to enable iterating over a 2-D array segment. The other option, *isPartial*, is set to notify the pass that flushing the whole row is unnecessary. Consequently, it will retrieve the boundaries of updated elements from the boundaries of the child of the target loop unless the indexing boundaries are provided by the programmer.

The kernel's loop nest that is tagged by the pragma is visited inside-out starting from the innermost loop where a reference to the target array is located. For specific loops, as will be discussed later, the loop induction variable, lower bound, upper bound, and step are registered, because they determine the range of the modified data within the loop. Our system right now handles natural *for-loops* where an induction variable can be retrieved through loop analysis. At the nesting level where the induction variable has the same name as *targetLoopInductionVariable*, the previously collected information is used to construct a one- or two-level loop structure to persist recently computed data elements. When the data changed within the nesting level selected by the programmer via *targetLoopInductionVariable* reside in multiple rows of the result array, *targetArrayName*, a two-level loop nest is needed to navigate through the 2-D segment of the array. To cover the whole segment, this loop starts from the same value as the index of *target loop*—where the newly generated loop will be inserted—in the current iteration. Thereafter, it proceeds in steps equal to 1 and ends after visiting all rows in the segment where the number of these rows is equal to the *target loop* step. For example, in TMM the newly generated loop in *ii* level will start from *ii* value in the current iteration, steps 1 per iteration, and ends at $ii + ii.step$ where $ii.step = bsize$, as shown in Figure 1(a).

Another loop is always generated either as a second-level loop of the previously generated loop or as a single loop that represents the whole data persisting loop structure. This loop is responsible for walking through a designated row of the target array to flush updating data at a step equal to the cache-line size. Therefore, the induction variable of this loop starts from the lower bound of the *targetLoopInductionVariable* in case of *isPartial* is true or *zero* otherwise. It ends at either the upper bound of the *target loop*'s induction variable in case of *isPartial* is true or *array row length* otherwise. The step interval is equal to the number of elements that resides in a single cache line per iteration. Using one/two induction variable/s, an element of the target array is indexed and its address is passed to the flush instruction; as a result, the containing cache line is transferred to NVM. One or more stages of synchronization and induction variables logging are prepared, and corresponding instructions are inserted. The purpose and layout of synchronization instructions will be discussed in Section 3.3 when we describe the NVM instructions. However, logging the induction variables is necessary for resuming the computation when recovering from a failure. We

ALGORITHM 1: Pseudo code of the compiler pass code analysis and generation.

Input: Loop L in LLVM IR // Innermost loop of the loop-nest tagged by the custom pragma directive
isTiled, isPartial // Compile time commandline options
targetLoopInductionVariable, targetArrayName[[from:to],[from:to]] // Pragma arguments, optional range limits
Output: Updated Loop L' with necessary flushing one-level (LC_{flush}) or two-level (LR_{flush} and LC_{flush}) loop-structure
 and logging/synchronization instructions for persisting data

```

1  TargetArray ← get pointer to targetArrayName
2  CLEMENTS ← Cache line size / sizeOf(*TargetArray) // No. of array elements per cache line
3   $N$  ← Number of elements per row in TargetArray
4  childLoop ←  $L$ 
5  currentLoop ←  $L$ .getParentLoop() // outermostLoop.getDepth() = 1
6  while currentLoop.getDepth() > 0 do
7      if currentLoop.inductionVariable is targetLoopInductionVariable then
8          codeInsertionPoint ← after last instruction of currentLoop
9          if optionalRangeProvided then
10             fill  $LR_{flush}$ .LowerBound,  $LR_{flush}$ .UpperBound,  $LC_{flush}$ .LowerBound, and  $LC_{flush}$ .UpperBound
11          end
12          if isTiled then
13              $LR_{flush}$  ← newLoop() // loop to iterate over multiple rows to be flushed; ex. TMM
14             if optionalRangeProvided then
15                  $LR_{flush}$ .LowerBound ← currentLoop.inductionVariable
16                  $LR_{flush}$ .UpperBound ←  $LR_{flush}$ .LowerBound + currentLoop.step // loop on block-size rows
17             end
18              $LR_{flush}$ .Step ← 1
19             codeInsertionPoint ←  $LR_{flush}$ .body
20          else
21             row ← currentLoop.inductionVariable // Single row should be flushed at a time
22          end
23          Adjust loop bounds for cache alignment(TargetArray, Cache line size) code
24           $LC_{flush}$  ← newLoop() // loop on columns within a single row with step = cache line size
25          if optionalRangeProvided then
26             if isPartial then
27                  $LC_{flush}$ .LowerBound ← childLoop.lowerBound // Partial-row flushing
28                  $LC_{flush}$ .UpperBound ← childLoop.upperBound
29             else
30                  $LC_{flush}$ .LowerBound ← 0 // Full-row flushing
31                  $LC_{flush}$ .UpperBound ←  $N$ 
32             end
33          end
34           $LC_{flush}$ .Step ← CLEMENTS
35          codeInsertionPoint ←  $LC_{flush}$ .body
36          insert instruction to flush cache line at &TargetArray[row*  $N$  + col]
37          codeInsertionPoint ← after  $LR_{flush}$ 
38          break// IV logging is inserted by the next loop for this level and outer levels if exist
39      end
40      childLoop ← currentLoop
41      currentLoop ← currentLoop.getParentLoop() // Depth of currentLoop is decremented by 1
42  end
43  while currentLoop.getDepth() > 0 do
44      codeInsertionPoint ← after last instruction of currentLoop
45      insert logging and synchronization code for currentLoop nesting level
46      currentLoop ← currentLoop.getParentLoop()
47  end

```

describe the logging operation for different kernels in Section 4. The description of the recovery using recompute for TMM is described in Section 2 and can be simply constructed for other kernels in a similar way.

The detailed code generation of this loop structure is described in Section 3.4. The generated code is appended to the original loop body along with the loop index logging code. Persisting the loop index value is important, because it enables the recognition of the data that are persisted for certain. The index value is made durable after the completion of persisting the data segment. If recovery is needed, then recomputation can start from the next iteration to the persisted index value. Thus, a small part of persisted data might be recomputed, but there is no doubt about the persistence of previous data. From this point up to the outermost loop, while traversing the loop-nest, each loop-index value is persisted for the purpose of valid recovery. Updating the persisted values of the indices is performed as a transaction if needed to ensure either a complete or no update, hence to guarantee consistency. The compiler pass inserts the entire data persistence instructions. Thereafter, the object code is generated, including the code inserted automatically.

3.3 NVM Instructions

Recently, Intel has provided instructions to support NVM programming [9]. We use the *clflushopt* and *sfence* instructions to achieve the functionalities offered by our design. Although we use *clflushopt* in the current implementation for comparison to our previous work that we extend in this article, *clwb* will be considered in future work. Because we expect *clwb* to introduce performance gain when applications other than the loop-based computational are considered.

- **clflushopt** This instruction invalidates the whole cache line that contains the memory location referenced by the provided operand from all levels of cache coherence domain. Our compiler-support has an advantage compared to previous works that use a flush operation per data element. We limit the flushes of consecutive data elements to once per all elements that reside in the same cache line, while previous work used to perform a flushing operation per data element. This feature of the proposed compiler-support effectively decreases the number of instructions, cache misses, and the program runtime.
- **clwb** This instruction writes back the cache line that contains the memory location referenced by the provided operand, if modified, from any level of cache coherence domain. For optimization, the cache line may be retained in or invalidated from cache hierarchy as decided by the hardware. *Clwb* is ordered only by store-fencing operations and is implicitly ordered by older stores to the same address.
- **sfence** This instruction synchronizes the execution of memory stores so store instructions before *sfence* are executed before the store instructions that come after it. *Clflushopt* instructions are ordered with respect to the following accesses to the same cache line being invalidated. It allows the overlapping of store instructions that write to different cache lines. *Sfence* instruction is necessary to enforce the ordering with respect to other write operations to the same cache line or *clflushopt* instructions that target other cache lines.

3.4 Code Generation

Our compiler-support generates two parts of the code to make the critical data persistent and to log necessary parameters for recovery at a valid state.

Using the information provided by the programmer and the information extracted from the code, a one- or two-level loop structure is generated to persist a one- or two-dimensional range of data elements, as shown in Figure 5. The borders of the range are extracted from the original code's loops in the loop nest. Hence, our method can adapt to partial-range data persistence

```

1 // CLSIZE: cache line size in byte
2 for (r=rowStart; r<rowEnd; r++) {
3     startAddress = &array[r][ colStart ] & (CLSIZE-1);
4     endAddress   = &array[r][ colEnd ];
5     //Data flushing starts at the adjusted bounds
6     for (a=startAddress; a<endAddress; a+=CLSIZE)
7         _mm_clflushopt(a);
8 }
9 _mm_sfence(); //synchronization

```

Fig. 5. Example of the automatically generated code for data persistence with loop bounds adjusted for cache alignment.

requirements, which diminishes the unnecessary data transfer operations. The effect of this precise determination of range borders on reducing wasted runtime, and data transfer in different benchmarks is discussed in the following sections.

The body of the generated loop structure contains flushing and synchronization instructions. A single *clflushopt* instruction is responsible for causing the elements that reside in a single cache line to be invalidated and moved to the NVM level. An adjustment for the loop bounds is used to overcome cache alignment issues if they exist. After the end of the flushing loop, *sfence* instruction is inserted to synchronize writing to the cache lines being invalidated. Synchronizing accesses to the same cache line requires the usage of *sfence* instruction. To enable recovery at a valid state, a log should be kept to identify the last computation results that were made durable. Our method tracks the most recent valid state by logging the loop induction variables' values starting from the automatically generated loop upwards to the outermost loop in the loop nest. Updating this log is done as an atomic transaction to ensure consistency and correctness of the recovery. Each parameter is flushed, then an additional *sfence* is generated to enforce synchronized access to the cache line being persisted. In case of failure, the program can restart loops from the last stored indices values to recompute any data that were not transferred to the NVM before failure. Meanwhile, the validity of the previous computation results is ensured.

The approach to persist a critical array with software in a non-atomic manner while atomically persisting induction variables is described in Reference [17]. Our method uses the same precautions to achieve failure safety and cache consistency when persisting kernels' critical data. Recovery is possible, because we can identify the part of data that might be inconsistent and can recompute. Input data are not changed, because the computations are done out-of-place, so, re-computing any part of the result data is possible. Moreover, one operation is used to persist all elements in a cache line block, and loop bounds adjustment is used to handle cache alignment issues. An example of the generated code would look like the code shown in Figure 5. In this example, we assume that it is required to persist a critical data structure named *array*, which is a 2-D matrix. Also, we assume that the granularity of persistence from the programmer hint leads to the region defined by rows from *rowStart* to *rowEnd* and columns from *colStart* and *colEnd*.

To allow overlapping between multiple flushing instructions that target different cache line blocks, *sfence* synchronization instruction is kept out of the flushing loop. It is added to guarantee a complete data persistence before the start of updating persisted values of indices. The generated code adapts to the modified kernel, as we will explain in the following sections.

4 BENCHMARKS

There are many common computational benchmarks that are intensively used in scientific applications. Our compiler-support targets those with deep loop nests that do the major part of

computations and store the results in large data objects (mainly matrices). The resulting data need to be transferred to the NVM periodically to maintain durability. The most recent data transfer operations are tracked through logging the values of loops indices at the iterations when the data are checkpointed. At recovery, the latest persisted values of indices are retrieved, whereby loops can resume computation. In this section, we show how our framework modifies four of the intensively used kernels in many scientific applications for data persistence.

Data transfer to NVM comes with overheads that we tried to minimize by decreasing the essential amount of data transfer and the number of operations. Cache techniques make it faster to transfer a block than transfer individual elements. So, we refrained from making every element in the result matrix persistent individually. Thus, for a chunk of elements that needs to be transferred, we transfer a whole cache block per operation and adjust loop bounds to handle the cache alignment issues. The performance gain that is introduced by this feature is shown in Section 5.

Another main contribution of our work is that the compiler-support can recognize when the computations result in changing only some of the elements in the considered range; for example, when the result matrix is diagonal, triangle, or other. This is common in arithmetic algorithms such as Lower-Upper Decomposition (LU) and Gauss Transformation. Moreover, for imperfectly nested loops that are difficult to analyze, the programmer has the option to provide the range to be persisted explicitly in the pragma to enhance the performance. The performance gain from tackling this issue can be clearly seen in Section 5, where our compiler-support successfully transferred only the updated elements.

4.1 FFT

Fast Fourier Transform (FFT) is a computational algorithm of a certain significance to many of the scientific HPC applications. Because FFT is widely used among such applications, it would be important to guarantee failure recovery at a negligible overhead. The experimental study in Reference [37] showed FFT to be of weak recomputability in case of no flushing or logging used for recovery that necessitates data durability procedures. Figure 6 shows the loop-based code segment that is responsible for the computation of FFT iterative kernel [10, 45].

The innermost two loops in this kernel introduce a scattered access pattern when writing to the result matrices x and y where a single element in a logarithmic-shrinking interval is written. Consequently, flushing the result elements at this level would waste the effort of flushing a complete cache line in which a single element changed. That eliminates the inner loop-nesting levels from being candidates for flushing the result elements. Meanwhile, the outermost loop iterates over the whole matrix for values precision improvement. That is why it is the best candidate for inserting flushing instructions where the whole matrix will be copied to the NVM before the next iteration—given that the programmer can simply provide the preferable granularity of enforcing data durability using the provided *pragma* directive.

The issue of FFT scattered data access is shown in Figure 7. The outermost loop iterations are represented by the dashed rectangles. The second-level loop's iterations are represented as horizontal layers inside the rectangles. Finally, the innermost loop iterations are represented as chunks in the layers of the containing loops. The innermost loop (i loop) updates two elements separated by 2^0 to $2^{\log n - 1}$ elements per iteration. Whereas the number of iterations decreases from $2^{\log n - 1}$ to 2^0 due to the change of the starting index value that depends on the second level loop (j -loop) index. By the time of j -loop iterations completion, the whole elements of the matrix have been updated, making this a suitable insertion point for our compiler-support data persistence flushing code. This minimizes the number of necessary flushing operations to the *size of array/size of cache line* in the best-case scenario. However, as we decrease the amount of data transfer, we increase the data exposed to the risk of being lost and should be recomputed after failures.

```

1 //compute the FFT
2 m = log(n) //n is no. of elements
3 c1 = -1.0;
4 c2 = 0.0;
5 l2 = 1;
6 #pragma clang loop persist (x, l)
7 for(l=0; l<m; l++){
8     l1 = l2;
9     l2 <=<= 1;
10    u1 = 1.0;
11    u2 = 0.0;
12    for(j=0; j<l1; j++){
13        for(i=j; i<n; i+=l2){
14            i1 = i + l1;
15            t1 = u1 * x[i1] - u2 * y[i1];
16            t2 = u1 * y[i1] + u2 * x[i1];
17            x[i1] = x[i] - t1;
18            y[i1] = y[i] - t2;
19            x[i] += t1;
20            y[i] += t2;
21        }
22        z = u1 * c1 - u2 * c2;
23        u2 = u1 * c2 + u2 * c1;
24        u1 = z;
25    }
26    c2 = sqrt ((1.0 - c1) /2.0) ;
27    c2 = -c2;
28    c1 = sqrt ((1.0 + c1) /2.0) ;
29    // Persistence handling code inserted here.
30 }

```

(a) Directive added to the original FFT code.

```

1 //Compiler generated code
2 //CLSIZE: Size of cache line block
3 startAddress = &x[0] & (CLSIZE-1);
4 endAddress = &x[n-1]
5
6 // Persisting data
7 for(a=startAddress; a<endAddress; c+=CLSIZE)
8     _mm_clflushopt(a);
9 //Store Synchronization
10 _mm_sfence();
11 // Persisting induction variable
12 lastPersistedL = l;
13 _mm_clflushopt(&lastPersistedL);
14 //Store synchronization
15 _mm_sfence();

```

(b) Persistence handling code generated and inserted by the compiler-support.

Fig. 6. Non-recursive Fast Fourier Transform (FFT) computation loop-nest.

As we explained previously, our compiler-support benefits from the ability of *clflushopt* to invalidate a whole cache data line block per operation. Instead of flushing each element immediately after it has been calculated, we choose a coarser grain granularity of making the result elements durable. Doing so, our compiler-support can flush multiple elements using a single *clflushopt* instruction to reduce the overhead. However, if the computational algorithm data access pattern is scattered, this means that we should find a point at which a large number of consecutive elements has been changed. Thus, our compiler-support will flush these consecutive elements achieving the optimum usage of the *clflushopt* cache line invalidation.

4.2 LU

Although LU decomposition is a very important computational algorithm that is widely used, it is not the only reason why we choose it for our evaluation. LU computational loop nest, as shown in Figure 8(a), performs a partial row-based update of the result matrix elements. Ignoring such property would result in storing too many elements that did not change in both checkpointing and logging techniques. The larger the amount of data to be persisted, the longer the time computations

Fig. 7. Three-level loop-nest in FFT showing the scattered data access by the innermost two loops (similar level of color darkness implies adjacency).

will be suspended during data transfer. The partial data access for updating some of the matrix elements due to the nature of the LU computations is an opportunity for performance gain.

The provided LU code's endeavor is to reconstruct the given matrix in an upper-triangular matrix where all elements below the diagonal are zeros. This is done through consecutive steps of converting all column elements below the corresponding diagonal-element into zeros. Meanwhile, to preserve the system of equations validity, the remaining elements in the same row of any changing element should also be updated according to the same mathematical operations. In other words, assuming that the top left corner of the matrix is the start point, when converting the element (i, i) , the visited elements will be contoured below the (i) th row and to the right of the $(i+1)$ th column, as shown in Figure 9. Consequently, the total amount of required data transfer operations decreases, from n^3 to $\sum_{i=1}^n i^2 = (n^3)/3 + (3n^2 + n)/6$, which is significant for large matrices. This is a significant potential that can improve the performance of data durability techniques and reduce data transfer operations overhead.

Our compiler-support can avoid flushing unmodified elements in the array through copying the loop index boundaries to the generated flushing loop's induction variable to index the same set of modified array elements. However, if the computational algorithm implementation includes imperfectly nested loops that introduce irregular stores out of the predictable range, as in Figure 8, the programmer can provide the matrix indices boundaries of the range to be persisted in the pragma. This limitation is induced by the LLVM IR where arrays are represented with pointers but, recently, some work is done aiming to provide more IR abstraction that we may consider for future work [46].

Unlike FFT, the updated elements that need to be persisted are consecutive in LU result matrix. Besides, as long as the size of the updated elements in one row (i.e., the remaining elements in the designated contour) is larger than or equal to the cache line block size, the *clflushopt* behavior that flushes the full cache line will be fully utilized. Nevertheless, the expected misutilization for the final part of the computations when few elements remain is negligible for the large matrices of the HPC applications. This is an advantage of our compiler-support compared to the previous techniques and the performance gain due to it being notable in the results of our experiments. The code generated to modify the kernel for data persistence is shown in Figures 8(b) and 8(c).


```

1  #pragma clang loop persist (A[j][i:n], j)
2  for(i = 0; i < n; i++){
3      maxA = 0.0;
4      imax = i;
5      for(k = i; k < n; k++){
6          if ((absA = fabs(A[k][i])) > maxA){
7              maxA = absA;
8              imax = k;
9          }
10         if (maxA < Tol) return 0; // failure
11         if (imax != i) { // pivoting P
12             j = P[i];
13             P[i] = P[imax];
14             P[imax] = j;
15             // pivoting rows of A
16             for(int swap=0; swap<n; swap++){
17                 ptr = A[i][swap];
18                 A[i][swap] = A[imax][swap];
19                 A[imax][swap] = ptr;
20             }
21             //counting pivots starting from N
22             P[n]++;
23         }
24         for (j = i + 1; j < n; j++){
25             A[j][i] /= A[i][i];
26             for (k = i + 1; k < n; k++){
27                 A[j][k] -= A[j][i] * A[i][k];
28                 //The persistence handling code
29                 //inserted here
30             }
31             // Persisted induction variables update
32             //inserted here
33         }

```

(a) Directive added to the original LU code.

```

1  //Compiler generated code
2  // CLSIZE: Size of cache line block
3  startAddress = &A[j][i] & (CLSIZE-1);
4  endAddress = &A[j][n];
5
6  for(a=startAddress; a<endAddress; a+=CLSIZE)
7      _mm_clflushopt(a);
8      _mm_sfence();
9      lastPersistedJ = j;
10     _mm_clflushopt(&lastPersistedJ);
11     _mm_sfence();

```

(b) Persistence handling code generated and inserted by the compiler-support.

```

1  //Compiler generated code
2  lastPersistedILog = lastPersistedI;
3  _mm_clflushopt(&lastPersistedILog);
4  _mm_sfence();
5  startUpdating = true;
6  _mm_clflushopt (&startUpdating);
7  _mm_sfence();
8  lastPersistedJ = i;
9  lastPersistedI ++;
10 _mm_clflushopt (&lastPersistedJ);
11 _mm_clflushopt(&lastPersistedI);
12 _mm_sfence();
13 startUpdating = false;
14 _mm_clflushopt(&startUpdating);
15 _mm_sfence();

```

(c) Persisted induction variables update code generated and inserted by the compiler-support.

Fig. 8. LU computation loop-nest.

4.3 Gauss

The Gaussian elimination is an algorithm of solving a system of equations implemented in a 3-level deep loop structure. It tends to convert the input matrix into an upper triangle form. The proposed compiler-support handles this kernel similarly to the LU.

4.4 TMM

Tiled matrix multiplication is a modified version of ordinary matrix multiplication optimized for cache temporal locality [51]. In the 6-loop tiling algorithm, both input matrices and out matrix are divided into equal-size blocks of $blockSize \times blockSize$ elements. Every iteration of the second-level loop includes a complete row of blocks calculation in the result matrix. This loop represents the best candidate for NVM persistence that is done by our automatically generated code. Updated data that need to be persisted are perfectly consecutive where the cache line block transfer with a single flushing operation will perform the best. The range of persisted data is uniform, which

Fig. 9. LU shrinking partial-range data access.

makes the loop structure formation straightforward. Persisting the result matrix and recovery process in TMM is thoroughly discussed in Section 2 and previous work [17].

5 EVALUATION

In this section, we show the performance of our compiler-support and the significant reduction of overheads compared to traditional checkpointing techniques. We use a real machine with DRAM that supports NVM instructions to model NVM. The machine is equipped with Intel(R) Xeon(R) Gold 5120 x86-64 at 2.20 GHz CPUs and runs Ubuntu 16.04.5 LTS. Because our technique is designed for HPC applications, we tested the kernels with 64 MB workloads that are relatively larger-scale compared to the 1 MB workloads in the previous work [2]. We chose to set the input matrix size to 64 MB in all tested kernels, which is not feasible to evaluate using simulators. Although the machine is not equipped with an NVM, the statistics from the performance counters describe kernels' behavior comprehensively, which makes it possible to speculate the influence of NVMs. For the compiler infrastructure, we used Clang-7.0 and LLVM-7.0.0 for the compiler-support implementation.

The performance events monitored at runtime for the set of tested kernels are shown in Table 2, and to facilitate the readability in a comparative aspect they are plotted in Figure 10. As the compiler-support inserts additional instructions for NVM data persistence, it is normal to see an increase in the number of instructions for all kernels that ranges from 0.05% in LU and Gauss, where only a part of every row is flushed, and this part shrinks as the computations proceed, to 0.53% in TMM, where the kernel adopts a larger number of iterations for the tiled computations. However, the percentage of the increase in the number of instructions is trivial. The number of cache misses drops slightly in FFT and LU due to cache pollution, and this observation is discussed in the next section. The overall extra CPU cycles are too low, because the testing system does not really contain an NVM. Although we are aware that it should be higher in the case of using a system that includes an NVM chip, we believe that this overhead will still be very small based on the results of the experiments from References [2, 17] that tested the kernels on a GEM5-simulated system. These results showed that using the same code arrangement for TMM (i.e., hybrid checkpoint recompute at ii granularity) added 7% and 8% overheads in running time and number of writes, respectively, for $1k$ -squared input matrix and tile $bsize$ of 64.

Fig. 10. Compiler-support overheads normalized to original code.

Table 1. Loop-nest Level for Persisting Data and Granularity of Persisted Data (Outermost Loop is the 1st Nest-level)

Kernel	Loop-nest Depth	Critical Data Structure	Our Compiler-Support		Checkpoint	
			Nest-level	Data Persisted	Nest-level	Data Persisted
FFT	3	1-D array [N]	1 st	N	1 st	N
LU	3	2-D array [n×n]	2 nd	1×n	1 st	n×n
Gauss	3	2-D array [n×n]	2 nd	1×n	1 st	n×n
TMM	6	2-D array [n×n]	2 nd	block size×n	1 st	n×n

Table 2. Modified Benchmarks (with Data Persistence Instructions) Performance Variation from the Original Program

Kernel	cache-misses	cache-references	cpu-cycles	instructions	cpu-clock
FFT	−0.96%	−0.68%	−0.58%	+0.14%	−0.58%
LU	−2.03%	+0.62%	+0.61%	+0.05%	+0.60%
Gauss	+2.54%	+0.41%	−0.10%	+0.05%	−0.18%
TMM	+2.23%	−13.68%	+0.45%	+0.53%	+0.42%

6 DISCUSSION

6.1 Cache Behavior Analysis

HPC applications have various algorithms that handle the computations and resultant data differently. As discussed previously, our method will guarantee better performance when computations include consecutive stores for adjacent data in the cache. This enables fine-grain data persistence and consequently, a smaller amount of lost data at failures. However, a scattered data access pattern forces coarse-grain data persistence. This comes with the risk of losing a larger amount of data if the system fails before making the recent data durable. The pattern of data access in the critical data structure would significantly affect the data persistence code that our compiler-support will generate. Table 1 shows this effect in each of the four benchmarks we used for testing our method.

Table 3. Performance Monitoring Events of L2 Cache for the Tested Kernels Variation from the Original Programs

Performance Monitoring Event	FFT	LU	Gauss	TMM
L2_RQSTS.REFERENCES	-0.61%	+0.42%	+0.44%	-0.62%
L2_RQSTS.ALL_PF	-4.87%	+1.83%	-0.97%	-4.00%
L2_RQSTS.MISS	-0.72%	+0.56%	+0.46%	-13.25%
L2_RQSTS.DEMAND_DATA_RD_HIT	+8.96%	-28.33%	-6.13%	-0.19%

Table 4. Performance Statistics of Kernels with Critical Data CRAFT Checkpoint Normalized to the Original Kernels without Checkpoint

Kernel	Cache Misses	CPU Clock	Instructions
FFT	1.24	1.42	1.88
LU	13.97	22.44	14.57
Gauss	14.03	28.76	18.78
TMM	3.53	1.13	1.42

Our experiments showed interesting cache behavior that results from data persistence instructions usage. This behavior is obvious in the case of nonconsecutive data access as what happens in FFT and in some way in LU also. In the normal execution of the original program, the cache prefetcher causes a higher rate of cache misses due to cache pollution it causes. However, in case of using the data-flushing instructions to write back cache lines, the opportunity of cache pollution is much lower. This can be observed from the performance statistics shown in Table 3.

6.2 Example of Checkpointing Overheads

NVMs research is motivated by the desire of mitigating large overheads that the checkpoint techniques introduce to kernels. Checkpointing includes a large amount of data transfer to disk at slow rates. Recently developed checkpoint tools tackle smaller stored data size, offer checkpointing rate control, and permit the selection of critical data structures to store durably. As an example, CRAFT [40] is an application-level checkpoint/restart library that provides multiple APIs to checkpoint methods. To checkpoint an array, a checkpoint instance is created, then parameters that describe the array including the array name and number of dimensions, and each dimension size is passed through *add()* function call. As the checkpoint instance holds enough information, it can specify the array to be checkpointed and the size of data to store. The rate of making checkpoints is controlled by choosing the loop nesting level in the computation loop structure that iterates at the desirable rate to update the checkpoint by calling *updateAndWrite()*. We designed an experiment to reveal the performance enhancement opportunities in checkpoint techniques. We tuned CRAFT to make checkpoints of the same target data structure that was targeted by our compiler-support. It is not possible to tune CRAFT to checkpoint only the updated part of the matrices, so, we checkpoint the whole matrix at the least possible rate to minimize redundant data transfer and storage space. In other words, we insert the call for *updateAndWrite()* in the outermost loop of the computation loop nest in each kernel.

Table 4 shows the performance statistics of the tested kernels that use CRAFT normalized to the original programs with no checkpointing. Checkpointing FFT with CRAFT introduces relatively low overheads, because the number of checkpoint updates is $O(\log n)$, the same as the outermost loop total number of iterations. The overheads are 42% in CPU clock and 88% in the number of

instructions. LU and Gauss kernels incur the large overheads due to CRAFT, which are 22.4× and 28.8× CPU clock and 14.6× and 18.8× the number of instructions for the two kernels, respectively. Although both kernels change only a portion of the matrix as a result of the computations, the whole matrix is checkpointed at $O(n)$. In TMM, CRAFT causes the lowest CPU clock and number of instructions overheads, 13% and 42%, respectively, among the four tested kernels. The reason behind these low overheads is that TMM has a ratio of cache references to the number of instructions, which is as low as 0.08%. TMM is optimized for temporal cache locality, but the additional checkpoint instructions cause the cache misses to grow to 3.5×.

7 RELATED WORK

Data persistence for fault tolerance and crash consistency due to various failures were elaborated in multiple previous works. For many years, researchers developed many techniques for logging and checkpointing where the application data are periodically stored to disks to minimize the probability of data loss. Some approaches required complicated hardware support or processor design manipulation. Others are pure software solutions that are portable over a various set of architectures. The recent adoption of NVM by some architectures encouraged research to utilize the byte addressability, large size, and high speed, which are offered by NVMs in innovative data persistence techniques.

The spread of distributed and parallel systems motivated the community to come up with efficient checkpointing techniques. The shift from using uni-processor supercomputers to clusters of processing nodes and parallel processors revealed the necessity of preserving the data from being lost due to probable unavailability of nodes. Diskless checkpointing [33] was one of the primary techniques that proposed a failure-safe execution of HPC applications on a network of workstations. By replacing the necessity of storage disks by memory and processor redundancy, this technique can recover from single (and sometimes multiple) node failures but not a whole system failure. A set of N workstations assigned for application execution is supported with additional parity-nodes where checkpoints of the processor and memory data in all fundamental nodes are calculated and stored. By the time any node failed to continue calculations, its checkpoint is recovered, and the data are restored at a replica. SafetyNet [44] is a hardware scheme that generates a system-wide checkpoint periodically without affecting the instruction set. To make it lightweight, SafetyNet logs registers and memory updates at a coarse granularity, ensures a consistent checkpoint by using global time and atomic transactions, and overlaps checkpoint validation with normal execution to hide latency. ThyNVM [38] reduces the program execution stalls that are necessary for creating checkpoints by increasing the stored metadata to allow variable granularity (i.e., cache block or page) checkpointing. DICE [1] is a differential checkpoint software plugin that is compatible with several existing checkpoint techniques. It is designed for energy-ambient embedded devices that perform intermittent computing. Automatic code instrumentation is used to determine differentials in volatile memory to minimize costly NVM accesses. In contrast to traditional checkpointing techniques, our system stores only critical data objects and some parameters that enable recomputation of a small lost portion of computed data if necessary.

Some innovated file-system techniques were tailored to offer persistent memory with high I/O throughput. BPFS [8] is a file system that uses special hardware to allow byte-addressable persistent memory to perform fast atomic durable transactions. PMFS [16] is a file-system for NVM that uses a hardware primitive to make data persistent at the level of stores. PMFS uses undo logging to preserve old data entries until the transaction is completely committed. However, multiple hardware-based solutions were proposed, and the following techniques are examples for them. PiCL is an FPGA prototype in Verilog for a software-transparent write-ahead-logging checkpointing mechanism that reduces overhead by relaxing the durability timing [30]. ReDU is a redo

logging hardware-assisted technique that dedicates a part of the RAM to serve as a write cache for the NVM, moving these expensive operations away from the critical path [22]. Another hardware-based data persistence for NVMs is a memory controller that reduces the amount of metadata, and the number of excessive writes is presented in Reference [53]. Data security in addition to persistence are the concerns of SuperMem [58], a programmer- and compiler-transparent hardware system. It introduces a counter write coalescing scheme and a cross-bank counter storage to decrease the number and the execution time of write requests, respectively. More effort towards low-overhead encrypted NVMs is presented in Osiris [54], a new memory controller design that provides crash consistency for encryption counters. The number of NVM writes is reduced by eliminating the unnecessary evictions from the counter cache, which waives significant performance overheads. Proteus [41] is a hardware logging technique that provides two new instructions to create a log entry and to store the entry data to NVM as well as hardware support for transactions' durability. While the log space is specified by the program, several log registers are added to the register file, and a log pending queue is used to keep log operations out of the way of regular write-backs. Janus [27] is a software-hardware technique that reduces latency of Backend Memory Operations (BMOs) in NVM systems due to writes in the critical path. BMOs including encryption, integrity verification, and deduplication are decomposed into sub-operations that can run in parallel. Janus introduces manual or compiler-automated instrumentation besides pre-execution requests to the memory controller that can be issued by the processor.

Software solutions ranged from manually editing source code to fully automating the process via compilers. A technique for checkpointing reduction to logging some parameters that are sufficient for recomputation is presented in [17]. This can be done by modifying the kernel code by adding instructions that perform synchronized data flushing to NVM periodically. A data management scheme is implemented for durable data structures with no logging in the main operations [11]. NVL-C [13] provides a static analysis language-level support that extends NVM features specifically designed for and implemented in C programming language. The same system was extended [14] to enable compiler-based automatic undo log aggregation using a special heap construct and an atomic directive. NV-Heaps [6] is a lightweight object system that manages persistent objects for NVMs. It provides familiar interfaces for the programmer to persist various objects from the application. It is organized in the system such that it can directly access the NVM without involving the operating system. That makes NV-Heaps faster than previous techniques that communicate with persistent memory through the operating system and its file system. Another lightweight system is Mnemosyne [49], which presented a user-abstracted NVM access. It used a low-level programming interface that enables the programmer to allocate NVM for arbitrary objects dynamically. Meanwhile, it provided in-place persistent transactions to make selected data objects durable. CrabTree [50] is a variation of B+-tree designed for ARMv8 architecture to achieve data persistence using NVM at a lower performance cost compared to existing B+-tree's implementations. The technique selects one of copy on write and shifting in place strategies to perform data insertion or deletion at runtime thereby to enable failure resilience. In Reference [23], the authors present a transaction mechanism for small data structures, limited to the half of a cache line, to persist data in NVM. Although this mechanism cannot be used for general arbitrary applications, it is designed to be a building block of techniques that persists more complicated data structures. Pangolin [56] is a fault-tolerance NVM library that persists arbitrary data objects. To protect objects against media and software failures, this library uses a combination of parity, replication, and object-level checksums.

NVRAM-aware logging [18] uses NVM for transaction logging only instead of entirely replacing the disk. JUSTDO [21] is a library that enables delimiting code sections with locks to be failure atomic. Every store is logged in a thread-specific log that is persisted before the store is performed,

and the recovery resumes execution from the last store instruction. JUSTDO does not discard any changes made within the locked section when recovering from failures; where iDo [25] extended JUSTDO by defining idempotent code regions within every failure-safe block. To reduce logging overheads, data are logged at the beginning of each idempotent region instead of per-store instruction. An analytical study [37] for the necessity of flushing and logging for application recomputation when recovering after failure proved to be critical for large input computational applications. Lazy persistency [3] does not perform any additional memory operations for data persistence other than those provided by the system. Meanwhile, it discovers inconsistency using checksums. NVCL [57] is a lightweight differential logging that optimizes the log frame size and flushes only the dirty cache lines. InCLL [7] is an undo logging that stores log for the data field in the same cache line making flushes unnecessary but is limited by the tight space. InCheck injects additional instruction by modifying back-end compiler passes to preserve the register file values into memory and checkpoint single memory location upon every store instruction and uses them to recover by repeating the execution of a single basic block [15].

Software-only compiler techniques for fault tolerance were introduced a long time ago. For example, a technique that allows the compiler to duplicate the program instructions and run two versions on separated register files, then periodically compares the output of corresponding instructions for validity is described in [35]. Atlas [4] is a compiler pass that instruments the code for persistent memory operations, then calls a runtime library to log them for reconstruction after failure. Breeze [29] is a compiler-based toolchain that reduces the required code modification for undo-logging. CoSpec [5] is a hybrid compiler-architecture technique to provide crash consistency for in-order processors that are used in NVM-equipped power harvesting systems. The compiler partitions the program into recoverable regions where all committed stores are held in a buffer and are moved to NVM by the end of the corresponding region execution. If the system fails while executing a region, then its stores disappear from the buffer without affecting the NVM state and the region can be restarted.

Most of the literature in NVM programming models support was directed to C/C++. However, significant work has been accomplished to provide NVM crash consistency in Java platforms. Earlier, Java Persistence API (JPA) [12] provided some easy-to-use coarse-grained abstraction APIs for the programmer to persist his data, but it caused overheads due to Java objects transformation into serialized data. In contrast, Intel offered Persistent Collections for Java (PCJ) [20], a fine-grained object-level persistence, but it is incompatible with the previous Java programs. Jdap [55] is an extension of JavaScript to support in-memory data persistence where parallel access of persistent shared data is allowed and a set of easy-to-use APIs is provided. Recently, Espresso [52] extends Java and its runtime, enabling the programmer to apply NVM persistency to their applications. This data persistency management framework overcomes the disadvantages of the previous work as it proposes Persistent Java Heap and Persistent Java object offering coarse and fine-grained persistency along with the compatibility with existing applications. Espresso adds keywords that enable creating objects in NVM while allowing same-type objects to be stored in DRAM and NVM. It also requires marking every persisted object or cache line write-back to NVM explicitly.

AutoPerist [42] is a runtime framework to support NVM data persistency implemented in Java Virtual Machine (JVM). JVM bytecodes are changed to perform extra actions that introduce execution overheads besides those caused by the NVM Metadata headers. It mainly targets database applications that use complicated data structures such as the doubly-linked list. Initially, the programmer marks entry points to critical data objects as durable roots. Then, at runtime, all objects reachable from the durable root are moved to NVM and persisted when modified. AutoPersist uses sequential persistency outside of failure-atomic regions and epoch persistency inside failure-atomic regions. QuickCheck [43] is a further extension of this work where the persistence checks

overheads are decreased. The baseline compiler collects profiling information to activate biases of persistence checks according to their execution frequency. Hereby, the optimizing compiler classifies checks into four states and generates optimized code for each of them. Unlike our system, Autopersist and QuickCheck perform a profile-guided optimization, and although they handle different kinds of applications, they require some logical analysis to choose the objects to be marked as durable roots and biasing based on activation rate. For database systems, NVM adoption is elaborated as in Reference [34] where NVM latency is hidden by interleaving arbitrary co-routines in parallel work for index joins and tuple reconstruction.

Our proposed compiler-support is a software tool implemented as a compiler pass that provides a simple pragma directive to pass information from the programmer. This information waives the need for the complex analysis to locate critical data objects and the loop-nest level that offers proper granularity of data persistence. Thus, the compiler-support can insert complex data persistence instructions that force data durability and logs a valid state to recover after failure. Recomputing a small part of the data that was computed before the failure, but was not moved to the NVM, may be needed. Furthermore, the data transfer operations are reduced to one operation per cache line block size.

8 CONCLUSION AND FUTURE WORK

HPC applications require failure-safety techniques that enable recovery at a valid state where computations can be resumed but not restarted. Recent NVMs systems offer a promising opportunity of persisting critical data with lower performance overheads compared to checkpointing. However, they require complex code transformations. Manual editing for the existing and newly developed HPC applications for NVM is non-trivial.

We present an automatic data persistence compiler-support for loop-based intensive computing HPC applications. We use a simple programmer directive to generate complex data persistence code that enables resuming computations when the application recovers from failures. Critical data structures durability, the granularity of persisting data, selective ranges of updated data, and fewer data transfer operations are all tailored for every application. Our compiler-support generates lower performance overheads compared with the previous work in logging and checkpointing.

In future work, we plan to extend our technique to cover multi-dimensional arrays with more than two dimensions, multi-array simultaneous persisting, additional types of data structures, and algorithms other than the loop-based computational kernels. It is also important to consider memory-intensive workloads, such as database or key-value workloads. We will also consider the optimized *clwb* instruction to write back data to NVM and a wider range of benchmarks for evaluation. The next step will be to apply our technique to parallel applications that run on multicore systems that require ensuring the compatibility of our compiler pass and its custom pragma directive with the support of OpenMP in LLVM.

REFERENCES

- [1] Saad Ahmed, Naveed Anwar Bhatti, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. 2019. Efficient intermittent computing with differential checkpointing. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'19)*. ACM, New York, NY, 70–81. DOI: <https://doi.org/10.1145/3316482.3326357>
- [2] Mohammad Alshboul, Hussein Elnawawy, Reem Elkhoully, Keiji Kimura, James Tuck, and Yan Solihin. 2019. Efficient checkpointing with recompute scheme for non-volatile main memory. *ACM Trans. Archit. Code Optim.* 16, 2, Article 18 (May 2019), 27 pages. DOI: <https://doi.org/10.1145/3323091>
- [3] M. Alshboul, J. Tuck, and Y. Solihin. 2018. Lazy persistency: A high-performing and write-efficient software persistency technique. In *Proceedings of the ACM/IEEE 45th International Symposium on Computer Architecture (ISCA)*. 439–451. DOI: <https://doi.org/10.1109/ISCA.2018.00044>

- [4] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging locks for non-volatile memory consistency. *ACM SIGPLAN Not.* 49, 10 (Oct. 2014), 433–452. DOI : <https://doi.org/10.1145/2714064.2660224>
- [5] Jongouk Choi, Qingrui Liu, and Changhee Jung. 2019. CoSpec: Compiler directed speculative intermittent computation. In *Proceedings of the 52nd IEEE/ACM International Symposium on Microarchitecture (MICRO’52)*. ACM, New York, NY, 399–412. DOI : <https://doi.org/10.1145/3352460.3358279>
- [6] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2012. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. *ACM SIGPLAN Not.* 47, 4 (2012), 105–118.
- [7] Nachshon Cohen, David T. Aksun, Hillel Avni, and James R. Larus. 2019. Fine-grain checkpointing with in-cache-line logging. *CoRR abs/1902.00660* (2019). arxiv:1902.00660 <http://arxiv.org/abs/1902.00660>.
- [8] 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, 133–146. DOI : <https://doi.org/10.1145/1629575.1629589>
- [9] Intel Corporation. 2016. Intel architecture instruction set extensions programming reference. *Technical Report*, 319433–030, Intel Corporation, Mountain View, CA.
- [10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms*. The MIT Press.
- [11] Tudor David, Aleksandar Dragojević, Rachid Guerraoui, and Igor Zablotchi. 2018. Log-free concurrent data structures. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC’18)*. USENIX Association, 373–386. <https://www.usenix.org/conference/atc18/presentation/david>.
- [12] Linda DeMichiel and Michael Keith. 2006. Java persistence API. Presented in JavaOne Conference. <https://docs.huihoo.com/javaone/2006/java-ee/ts-3395.pdf>.
- [13] Joel E. Denny, Seyong Lee, and Jeffrey S. Vetter. 2016. NVL-C: Static analysis techniques for efficient, correct programming of non-volatile main memory systems. In *Proceedings of the IEEE International Symposium on High Performance Distributed Computing (HPDC’16)*. ACM, 125–136.
- [14] J. E. Denny, S. Lee, and J. S. Vetter. 2017. Language-based optimizations for persistence on nonvolatile main memory systems. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS’17)*. 1163–1173. DOI : <https://doi.org/10.1109/IPDPS.2017.60>
- [15] Moslem Didehban, Sai Ram Dheeraj Lokam, and Aviral Shrivastava. 2017. InCheck: An in-application recovery scheme for soft errors. In *Proceedings of the 54th Design Automation Conference (DAC’17)*. ACM, New York, NY, Article 40, 6 pages. DOI : <https://doi.org/10.1145/3061639.3062265>
- [16] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys’14)*. ACM, New York, NY, Article 15, 15 pages. DOI : <https://doi.org/10.1145/2592798.2592814>
- [17] H. Elnawawy, M. Alshboul, J. Tuck, and Y. Solihin. 2017. Efficient checkpointing of loop-based codes for non-volatile main memory. In *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques (PACT’17)*. 318–329. DOI : <https://doi.org/10.1109/PACT.2017.58>
- [18] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. 2014. NVRAM-aware logging in transaction systems. *Proc. VLDB Endow.* 8, 4 (Dec. 2014), 389–400. DOI : <https://doi.org/10.14778/2735496.2735502>
- [19] Yingchao Huang, Kai Wu, and Dong Li. 2017. High performance data persistence in non-volatile memory for resilient high performance computing. *CoRR abs/1705.00264* (2017). arxiv:1705.00264 <http://arxiv.org/abs/1705.00264>.
- [20] Intel. 2016. Persistent collections for Java. <https://github.com/pmem/pcj>.
- [21] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-atomic persistent memory updates via JUSTDO logging. *ACM SIGPLAN Not.* 51, 4 (Mar. 2016), 427–442. DOI : <https://doi.org/10.1145/2954679.2872410>
- [22] J. Jeong, C. H. Park, J. Huh, and S. Maeng. 2018. Efficient hardware-assisted logging with asynchronous and direct-update for persistent memory. In *Proceedings of the 51st IEEE/ACM International Symposium on Microarchitecture (MICRO’18)*. 520–532. DOI : <https://doi.org/10.1109/MICRO.2018.00049>
- [23] Marcel Köppen, Jana Traue, Christoph Borchert, Jörg Nolte, and Olaf Spinczyk. 2019. Cache-line transactions: Building blocks for persistent kernel data structures enabled by AspectC++. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems (PLOS’19)*. ACM, New York, NY, 38–44. DOI : <https://doi.org/10.1145/3365137.3365396>
- [24] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. IEEE Computer Society, 75.
- [25] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung. 2018. iDO: Compiler-directed failure atomicity for nonvolatile memory. In *Proceedings of the 51st IEEE/ACM International Symposium on Microarchitecture (MICRO’18)*. 258–270. DOI : <https://doi.org/10.1109/MICRO.2018.00029>

- [26] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. 2015. Clover: Compiler directed lightweight soft error resilience. *ACM SIGPLAN Not.* 50, 5, Article 2 (June 2015), 10 pages. DOI: <https://doi.org/10.1145/2808704.2754959>
- [27] Sihang Liu, Korakit Seemakhupt, Gennady Pekhimenko, Aasheesh Kolli, and Samira Khan. 2019. Janus: Optimizing memory and storage support for non-volatile memory systems. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA'19)*. ACM, New York, NY, 143–156. DOI: <https://doi.org/10.1145/3307650.3322206>
- [28] C. D. Martino, W. Kramer, Z. Kalbarczyk, and R. Iyer. 2015. Measuring and understanding extreme-scale application resilience: A field study of 5,000,000 HPC application runs. In *Proceedings of the 45th IEEE/IFIP International Conference on Dependable Systems and Networks*. 25–36. DOI: <https://doi.org/10.1109/DSN.2015.50>
- [29] A. Memaripour and S. Swanson. 2018. Breeze: User-level access to non-volatile main memories for legacy software. In *Proceedings of the IEEE 36th International Conference on Computer Design (ICCD'18)*. 413–422. DOI: <https://doi.org/10.1109/ICCD.2018.00069>
- [30] T. Nguyen and D. Wentzlaff. 2018. PiCL: A software-transparent, persistent cache log for nonvolatile main memory. In *Proceedings of the 51st IEEE/ACM International Symposium on Microarchitecture (MICRO'18)*. 507–519. DOI: <https://doi.org/10.1109/MICRO.2018.00048>
- [31] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacher et al. 2002. *Recovery-oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies*. Technical Report. Technical Report UCB//CSD-02-1175, UC Berkeley Computer Science.
- [32] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory persistency. *SIGARCH Comput. Archit. News* 42, 3 (June 2014), 265–276. DOI: <https://doi.org/10.1145/2678373.2665712>
- [33] J. S. Plank, Kai Li, and M. A. Puening. 1998. Diskless checkpointing. *IEEE Trans. Parallel Distrib. Syst.* 9, 10 (Oct. 1998), 972–986. DOI: <https://doi.org/10.1109/71.730527>
- [34] Georgios Psaropoulos, Ismail Oukid, Thomas Legler, Norman May, and Anastasia Ailamaki. 2019. Bridging the latency gap between NVM and DRAM for latency-bound operations. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*. 8. DOI: <https://doi.org/10.1145/3329785.3329917>
- [35] George A. Reis, Jonathan Chang, and David I. August. 2007. Automatic instruction-level software-only recovery. *IEEE Micro* 27, 1 (Jan. 2007), 36–47. DOI: <https://doi.org/10.1109/MM.2007.4>
- [36] Jinglei Ren, Qingda Hu, Samira Khan, and Thomas Moscibroda. 2017. Programming for non-volatile main memory is hard. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*. ACM, 13.
- [37] Jie Ren, Kai Wu, and Dong Li. 2018. Understanding application recomputability without crash consistency in non-volatile memory. In *Proceedings of the Workshop on Memory Centric High Performance Computing (MCHPC'18)*. ACM, New York, NY, 27–36. DOI: <https://doi.org/10.1145/3286475.3286476>
- [38] 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 *Proceedings of the 48th IEEE/ACM International Symposium on Microarchitecture (MICRO'15)*. 672–685. DOI: <https://doi.org/10.1145/2830772.2830802>
- [39] Bianca Schroeder and Garth A. Gibson. 2007. Understanding failures in petascale computers. *J. Phys.: Conf. Series* 78 (July 2007), 012022. DOI: <https://doi.org/10.1088/1742-6596/78/1/012022>
- [40] F. Shahzad, J. Thies, M. Kreutzer, T. Zeiser, G. Hager, and G. Wellein. 2019. CRAFT: A library for easier application-level checkpoint/restart and automatic fault tolerance. *IEEE Trans. Parallel Distrib. Syst.* 30, 3 (Mar. 2019), 501–514. DOI: <https://doi.org/10.1109/TPDS.2018.2866794>
- [41] Seunghye Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. 2017. Proteus: A flexible and fast software supported hardware logging approach for NVM. In *Proceedings of the 50th IEEE/ACM International Symposium on Microarchitecture (MICRO'17)*. ACM, New York, NY, 178–190. DOI: <https://doi.org/10.1145/3123939.3124539>
- [42] Thomas Shull, Jian Huang, and Josep Torrellas. 2019. AutoPersist: An easy-to-use Java NVM framework based on reachability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)*. ACM, New York, NY, 316–332. DOI: <https://doi.org/10.1145/3314221.3314608>
- [43] Thomas Shull, Jian Huang, and Josep Torrellas. 2019. QuickCheck: Using speculation to reduce the overhead of checks in NVM frameworks. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'19)*. ACM, New York, NY, 137–151. DOI: <https://doi.org/10.1145/3313808.3313822>
- [44] Daniel J. Sorin, Milo M. K. Martin, Mark D. Hill, and David A. Wood. 2002. SafetyNet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. *SIGARCH Comput. Archit. News* 30, 2 (May 2002), 123–134. DOI: <https://doi.org/10.1145/545214.545229>
- [45] Willi-Hans Steeb. 1999. *The Nonlinear Workbook: Chaos, Fractals, Cellular Automata, Neural Networks, Genetic Algorithms, Fuzzy Logic with C++, Java, Symbolic C++ and Reduce Programs*. World Scientific Publishing Company.
- [46] TensorFlow. 2019. Multi-Level Intermediate Representation. <https://github.com/tensorflow/mlir>.
- [47] D. Tiwari, S. Gupta, and S. S. Vazhkudai. 2014. Lazy checkpointing: Exploiting temporal locality in failures to mitigate checkpointing overheads on extreme-scale systems. In *Proceedings of the 44th IEEE/IFIP International Conference on Dependable Systems and Networks*. 25–36. DOI: <https://doi.org/10.1109/DSN.2014.101>

- [48] N. H. Vaidya. 1997. Impact of checkpoint latency on overhead ratio of a checkpointing scheme. *IEEE Trans. Comput.* 46, 8 (Aug. 1997), 942–947. DOI : <https://doi.org/10.1109/12.609281>
- [49] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight persistent memory. *ACM SIGPLAN Not.* 47, 4 (Mar. 2011), 91–104. DOI : <https://doi.org/10.1145/2248487.1950379>
- [50] Chundong Wang, Sudipta Chattopadhyay, and Gunavaran Brihadiswarn. 2019. Crash recoverable ARMv8-oriented B+-tree for byte-addressable persistent memory. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'19)*. ACM, New York, NY, 33–44. DOI : <https://doi.org/10.1145/3316482.3326358>
- [51] Michael E. Wolf and Monica S. Lam. 1991. A data locality optimizing algorithm. In *ACM SIGPLAN Not.*, Vol. 26. ACM, 30–44.
- [52] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan. 2018. Espresso: Brewing Java for more non-volatility with non-volatile memory. *ACM SIGPLAN Not.* 53, 2 (Mar. 2018), 70–83. DOI : <https://doi.org/10.1145/3296957.3173201>
- [53] Song Wu, Fang Zhou, Xiang Gao, Hai Jin, and Jinglei Ren. 2019. Dual-page checkpointing: An architectural approach to efficient data persistence for in-memory applications. *ACM Trans. Archit. Code Optim.* 15, 4, Article 57 (Jan. 2019), 27 pages. DOI : <https://doi.org/10.1145/3291057>
- [54] M. Ye, K. Zubair, A. Mohaisen, and A. Awad. 2019. Towards low-cost mechanisms to enable restoration of encrypted non-volatile memories. *IEEE Trans. Depend. Sec. Comput.* (2019), 1–1. DOI : <https://doi.org/10.1109/TDSC.2019.2941193>
- [55] Litong You, Hao Xu, Qipeng Zhang, Tianyou Li, Chen Li, Yuting Chen, and Linpeng Huang. 2019. JDap: Supporting in-memory data persistence in JavaScript using Intel's PMDK. *J. Syst. Archit.* 101 (2019), 101662. DOI : <https://doi.org/10.1016/j.sysarc.2019.101662>
- [56] Lu Zhang and Steven Swanson. 2019. Pangolin: A fault-tolerant persistent memory programming library. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'19)*. USENIX Association, Berkeley, CA, 897–911. <http://dl.acm.org/citation.cfm?id=3358807.3358884>.
- [57] M. Zhang, X. Yao, and C. Wang. 2018. NVCL: Exploiting NVRAM in cache-line granularity differential logging. In *Proceedings of the IEEE 7th Non-Volatile Memory Systems and Applications Symposium (NVMSA'18)*. 37–42. DOI : <https://doi.org/10.1109/NVMSA.2018.00011>
- [58] Pengfei Zuo, Yu Hua, and Yuan Xie. 2019. SuperMem: Enabling application-transparent secure persistent memory with low overheads. In *Proceedings of the 52nd IEEE/ACM International Symposium on Microarchitecture (MICRO'19)*. ACM, New York, NY, 479–492. DOI : <https://doi.org/10.1145/3352460.3358290>

Received June 2019; revised October 2019; accepted November 2019