Efficient Checkpointing with Recompute Scheme for Non-volatile Main Memory

MOHAMMAD ALSHBOUL, North Carolina State University

HUSSEIN ELNAWAWY, North Carolina State University

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

KEIJI KIMURA, Waseda University, Japan

JAMES TUCK, North Carolina State University

YAN SOLIHIN, University of Central Florida

Future main memory will likely include Non-Volatile Memory. Non-Volatile Main Memory (NVMM) provides an opportunity to rethink checkpointing strategies for providing failure safety to applications. While there are many checkpointing and logging schemes in the literature, their use must be revisited as they incur high execution time overheads as well as a large number of additional writes to NVMM, which may significantly impact write endurance.

In this article, we propose a novel recompute-based failure safety approach and demonstrate its applicability to loop-based code. Rather than keeping a fully consistent logging state, we only log enough state to enable recomputation. Upon a failure, our approach recovers to a consistent state by determining which parts of the computation were not completed and recomputing them. Effectively, our approach removes the need to keep checkpoints or logs, thus reducing execution time overheads and improving NVMM write endurance at the expense of more complex recovery. We compare our new approach against logging and checkpointing on five scientific workloads, including tiled matrix multiplication, on a computer system model that was built on gem5 and supports Intel PMEM instruction extensions. For tiled matrix multiplication, our recompute approach incurs an execution time overhead of only 5%, in contrast to 8% overhead with logging and 207% overhead with checkpointing. Furthermore, recompute only adds 7% additional NVMM writes, compared to 111% with logging and 330% with checkpointing. We also conduct experiments on real hardware, allowing us to run our workloads to completion while varying the number of threads used for computation. These experiments substantiate our simulation-based observations and provide a sensitivity study and performance comparison between the Recompute Scheme and Naive Checkpointing.

CCS Concepts: • **Computer systems organization** → **Multicore architectures; Processors and memory architectures**; • **Hardware** → **Memory and dense storage; Non-volatile memory**;

Additional Key Words and Phrases: Memory systems, emerging memory technologies, computer architecture

Extension of Conference Paper: This article is an extension of work originally presented in PACT 2017 [23]. This work is supported in part by the National Science Foundation through awards CNS-171748, 1829142, and 1914717. Opinions expressed in this article are solely the authors' and not necessarily those of NSF.

Alshboul is a Ph.D. student of Solihin. Elnawawy was a Ph.D. student of Solihin at the time of him contributing to this work.

Authors' addresses: M. Alshboul and H. Elnawawy, North Carolina State University, USA; emails: {maalshbo, hmelnawa}@ncsu.edu; R. Elkhoully, Tanta University, Egypt and Waseda University, Japan; email: reem_elkhoully@f-eng.tanta.edu.eg; K. Kimura, Waseda University, Japan; J. Tuck, North Carolina State University, USA; email: jtuck@ncsu.edu; Y. Solihin, University of Central Florida, USA; email: yan.solihin@ucf.edu.

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.

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

1544-3566/2019/05-ART18

<https://doi.org/10.1145/3323091>

ACM Reference format:

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.
<https://doi.org/10.1145/3323091>

1 INTRODUCTION

Non-volatile memory (NVM) technologies have been advancing rapidly, and some of them are a strong contender for use as a future main memory, either for augmenting or replacing DRAM. One such an example is 3D Xpoint memory, which will be brought to market in 2017 by Intel and Micron [28]. These new *non-volatile main memory* (NVMM) technologies are byte-addressable and have access latencies that are not much slower than DRAM [3, 6, 8, 28, 34, 37, 38, 50]. NVMMs are expected to have a limited write endurance, making it imperative to keep the number of writes low [7]. Despite the limited write endurance and relatively high write latency compared to DRAM, NVMM's density and cost advantage over DRAM and near-zero idle power consumption make them a compelling candidate to replace or augment DRAM in high-performance computers [5].

At the same time, as high-performance computing (HPC) relies on an increasing number of nodes and components, it becomes increasingly likely that long-running computation will be interrupted by failures before completing. Frequent checkpointing has become essential because it allows applications to resume from a recent snapshot rather than re-execute from the beginning. Checkpoints can be taken for the entire system memory (*system-level checkpointing* or SLC) or only for the application address space (*application-level checkpointing* or ALC [12]). With ALC, the programmer or compiler determines when checkpoints are created and possibly which data is included in the checkpoint. Hence, ALC can reduce the amount of data that is checkpointed. Checkpointing overheads in SLC/ALC can be lowered when using NVM [33] to hold checkpoint states since NVM write latency is lower and write bandwidth is higher than hard drive or Solid-State Drive. However, in this context, NVM is simply treated as a faster secondary storage.

NVMM offers new opportunities to reconsider how failure safety can be achieved because the main memory can now hold the persistent state of a program, thereby serving the same role as a checkpoint. With NVMM, a consistent state can be constructed *in-place* in the NVMM utilizing the working data structures used by the applications. Consequently, only very minimal additional state beyond what the program already saves to memory needs to be recorded. A common approach to in-place checkpointing is logging, where modifications to key data structures are made durable at a transaction granularity. If the programmer wraps changes that need to be durable together in a transaction, data structures can always be in a consistent state. Durable transactional logging has been applied to the file system (e.g., BPFS [16]), heap (e.g., NV-HEAPS [15], NVML [27], and Mnemosyne [55]), and linked data structures (e.g., see Refs [31, 36, 48]). We implemented logging using the Intel PMEM instruction extension [27] on a machine model built on gem5 and measured the performance of tiled matrix multiplication. Our implementation reduces the execution time and write overheads compared to ALC: 8% vs. 207% execution time overheads, and 111% vs. 330% additional NVMM writes. While this represents much improvement in execution time overheads, the write overheads are still unacceptable. Hence, we explore another avenue.

In this work, we propose a new *Recompute* approach where we do away with the log entirely and allow state to become inconsistent. Some of the result matrix could become or appear inconsistent at the point of failure. However, we track enough additional information so that we can discard the possibly inconsistent state and re-run computation as needed. Recompute has the interesting property of being *restore-free*, in that no checkpoint needs to be copied back to memory or

consistent state restored using a log. We demonstrate our Recompute approach on loop-based codes that are essential in HPC workloads and present it in details for matrix multiplication. Loop-based code is interesting because we must consider optimizations such as tiling in the study.

We show that Recompute achieves failure safety with almost no penalty in execution time and write endurance. For tiled matrix multiplication, we find that Recompute has an execution time overhead of only 5% compared to an unmodified version of the kernel, while adding only 7% write overhead. Across various workloads, the geometric mean execution time overhead ratio is $1.03\times$ for *Recompute* vs. $1.91\times$ for traditional checkpointing, and write overhead ratio of $1.08\times$ vs. $1.38\times$ for traditional checkpointing. Experimenting on real hardware showed consistent results. Furthermore, the article investigates the impact of varying the checkpointing frequency on all the studied benchmarks and provides insights on which scheme is better in each scenario.

The rest of the article is organized as follows. Section 2 gives an overview of Intel PMEM persistency programming support. Section 3 describes checkpointing schemes, and Section 4 introduces our newly proposed Recompute scheme. Section 5 discusses a compiler support automation for Recompute scheme. Section 6 describes the evaluation methodology while Section 7 discusses the results of the experiments. Section 9 concludes the article after discussing the related works in Section 8.

2 BACKGROUND

2.1 Intel Persistency Programming Support

Recently, Intel announced a new PMEM persistency programming model, and published it at pmem.io [27]. PMEM includes a set of new instructions, including cache line write back (*clwb*) and optimized cache line flush (*clflushopt*), that can be used to write fail-safe software in conjunction with existing X86 instructions, such as cache line flush (*clflush*) and store fence (*sfence*).¹

Various persistency programming models have been proposed [16, 27, 31, 43, 48], including strict persistency [48], epoch persistency [16], buffered epoch persistency [16, 31], strand persistency [48], and transactional persistency [36, 55]. In contrast to those, Intel's PMEM extension recognizes that not all applications need persistency support; hence, it was designed to be flexible enough that it can be mixed with traditional programming models. To illustrate the contrast, consider strict persistency. Strict persistency requires that when a store is *globally performed* (i.e., visible to other threads), it is also *durable* (its value is accepted by the NVMM). With PMEM, programmers specify which stores need to persist and in which order they persist. To illustrate this, suppose that we wish to persist a store to address X before a store to address Y, but do not care about persisting or ordering stores to addresses U and V. Then, we would write:

```
i1: st U, 5;
i2: st V, 1001;
i3: st X, 1;
i4: clwb X;
i5: sfence;
i6: st Y, -300;
```

Instructions i1 (store to U) and i2 (store to V) are not augmented with any ordering or persistency constraints. Instruction i3 (store to X), however, is followed by i5 (*clwb*), which forces the store X to

¹Another PMEM instruction that was a part of PMEM is *pcommit*. The instruction was designed to flush a cache block from the memory controller buffer to the NVMM. However, Intel has recently deprecated *pcommit* because it requires the memory controller to have a "failure-safe" queue, where when a cache block is accepted by the queue, it can be considered durable. In light of this deprecation, we omit the use of *pcommit* in our schemes.

be written back from the cache. The *sfence* (i5) stalls the next store instruction (i6) until the cache line affected by store X has been accepted by the NVMM (either received by a failure-safe buffer in NVMM or fully written into NVMM cells), guaranteeing durability. This example illustrates which instructions need to be inserted by programmers in order to specify which stores must persist and the ordering of the persistency relative to other stores. We also note that programmers can construct other persistency models such as strict and epoch persistency using PMEM instructions.

clflushopt is similar in functionality to *clflush*. However, *clflushopt* is optimized for performance by relaxing the ordering requirements of *clflush*. In contrast to *clflush*, *clflushopt* does not impose strict ordering with all other stores. Instead, it is only strictly ordered with respect to stores to the same block address, giving some room to overlap the execution of stores to different blocks. Due to that, we rely on *clflushopt* in our design. *clwb* differs from both *clflush* and *clflushopt* in that it cleans a cache line by writing it back but retains the line in the cache. Another difference is that *clwb* only orders stores to the same address instead of ordering stores to the same block as in *clflushopt* [17].

The last instruction that we use is *sfence*. *sfence* was originally designed for memory consistency to specify a point where all older stores have been made visible to other threads but has now been extended to persistency. If placed right after PMEM instructions such as *clwb* and *clflushopt*, *sfence* waits until the PMEM instructions are durable before it completes, while stalling younger stores or PMEM instructions from being visible to the cache until it completes.

2.2 Durable Transactions with Write-Ahead Logging

Using the PMEM programming extensions we can force stores to become durable in an order of our choosing. However, this alone does not guarantee that memory is left in a failure-safe state since a failure could occur while making changes to data. What are also needed are atomic updates such that either all updates become durable or none of them do.

Transactions have long been used to achieve failure-safe updates to non-volatile storage [36, 55]. We use write-ahead logging or WAL [45] for our implementation of durable transactions. WAL makes a durable undo log of all the data that will be updated in the transaction before making any modifications. If a failure occurs during the transaction, we can use the undo log to recover the correct data. A durable transaction can be constructed in software using the following steps:

- Perform the undo-logging by making a copy of the data we want to update, and make the undo-log durable.
- Set a *transactionRunning* bit indicating the beginning of a transaction and make the bit durable.
- Update the data that needs to be modified and make the updates durable.
- Unset the *transactionRunning* bit to mark the transaction complete and make the bit durable.

The recovery mechanism is quite simple. If a failure occurs, we check the *transactionRunning* bit. If it was set to true, that means the failure occurred in the middle of a transaction. In that case, we restore all data using the entries in the undo-log. Then, we resume execution from the point of the failed transaction. On the other hand, if the *transactionRunning* was unset, it means there was no transaction running when the failure happened, so no further recovery actions are needed on the data protected by the transaction.

2.3 Tiled Matrix Multiplication Case Study

We use five benchmarks in our study, including matrix multiplication, LU decomposition, fast Fourier transform, Gaussian elimination, and two-dimensional convolution. In order to illustrate our technique, we will focus on matrix multiplication with tiling optimization as our case study.

Fig. 1. Tiled matrix multiplication code.

```

1  for (kk=0; kk<n; kk+=bsize) { //moves horizontally on matrix A, and vertically on matrix B
2      for (ii=0; ii<n; ii+=bsize) { // moves vertically on matrix A and C
3          for (jj=0; jj<n; jj+=bsize) { // moves horizontally on matrix B and C
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 of for jj
13     } //end of for ii
14 } //end of for kk

```

Fig. 2. Tiled matrix multiplication.

Suppose that we want to multiply matrix a with matrix b and store the result into matrix c . All matrices are square and have $n \times n$ size.

Tiling is a well-known cache optimization technique to improve temporal locality and reduce the cache miss ratio. In this article, we use a standard six-loop tiling for matrix multiplication. We refer to the tile size as $bsize$ (as in block size). In this version, tiling splits a , b , and c into tiles or blocks of $bsize \times bsize$. The operation of the tiled matrix multiplication (tmm) is shown in Figure 1, and the corresponding code is shown in Figure 2. We will briefly walk through the matrix multiplication example.

tmm consists of six loops: kk , ii , jj , i , j , and k . The kk loop splits matrices a and c into vertical groups; each vertical group consists of a number of $bsize$ columns. The kk loop also splits the b matrix into horizontal groups; each horizontal group consists of $bsize$ rows. The ii loop splits each of matrix a 's vertical kk groups into square $bsize \times bsize$ tiles as illustrated in Figure 1. Similarly, the jj loop splits each of matrix b 's horizontal kk groups into square $bsize \times bsize$ tiles. Moving to the innermost loops i , j , and k , as the iterations names indicate, each of these loops will move through within a tile that was made by the corresponding upper loops, ii , jj , and kk , respectively.

The tiled matrix multiplication operation starts with a kk group from matrix a and its corresponding kk group from matrix b . Then the ii loop selects the first tile from a 's kk group, and the jj loop selects the first tile from b 's kk group. After that, the i loop goes over all of the strides from a 's ii group and multiplies them by (the selected) vertical strides from matrix b 's jj tile. At the end of the i loop, the result is a $(bsize \times bsize)$ square in from matrix c (corresponding to the first ii , jj tiles) with intermediate values (not the final values). Then, we move on to the next jj tile from b 's kk group. When all jj tiles of b 's kk group are done, the ii loop moves to the next ii tile from the

a 's kk group. When all ii tiles of a 's kk group are done, the kk loop moves on to the next vertical kk group of a and the next horizontal kk group of b . The operation keeps going until all kk groups have been computed. We note that each kk loop iteration touches every element of the result matrix c once. Therefore, at the end of the computation, we would have written to every element of matrix c a number of times equal to $\frac{n}{bsize}$.

3 CHECKPOINTING TECHNIQUES

3.1 Traditional Checkpointing

A traditional solution for making an application failure-safe is by creating checkpoints of the application. A checkpoint typically includes a snapshot of memory and the full processor context (all registers, PC, etc.). The snapshot of memory should be taken at a particular instruction, such that any older instructions have their results reflected in the state but none of the younger instructions have taken effect. The first step toward a consistent snapshot is interrupting all processor cores using a precise interrupt mechanism. Then, the computation state in all cores, including the program counter, register files, and memory, must be saved to secondary storage. Note that any dirty data in the caches must be written back or flushed to the main memory before this snapshot is taken.

Since we are targeting a system with NVMM, instead of copying the checkpoint to secondary storage, we can copy it to a region in the NVMM instead. In our implementation, a checkpoint is created by copying matrices a , b , and c elsewhere in non-volatile memory. Our implementation's performance is more efficient than system-level checkpointing because the checkpoint excludes non-application address space, and we ignore the cost of saving the processor context. It is less efficient than application-level checkpointing in that the checkpoint could be made even smaller if we consider deeper application information. For example, if we know matrices a and b are not needed after the matrix multiplication, only the result matrix c needs to be included in the checkpoint.

3.2 Durable Transactions with Write Ahead Logging

For another comparison, we implement a durable transaction-based failure-safe version of the workloads. We use Intel PMEM instructions to achieve that. For the *tmm* code, it is clear that updating the result matrix needs to be wrapped into a durable transaction since the matrix needs to be consistent. However, it is also necessary to know how much of the work was already completed. For loop-based codes, a straightforward solution is to log all of the induction variables in the loop nest and their updates. Thus, we propose to augment the *tmm* code by explicitly writing the current loop indices to memory. Upon failure, we recover a consistent state by reading these indices and determining where in the loop-nest to resume execution.

The transactionalized *tmm* code that illustrates this approach is shown in Figure 3. Each transaction is the body of the ii loop iteration—a larger or smaller transaction on a different loop is also possible. We must log the elements of matrix c and the indices before updating them. The logging part is shown in lines 4 to 7. In this part, we log all the c elements we are about to modify inside the ii loop, which is basically a full horizontal kk group of a number of $bsize$ rows. We log the *lastPersistedII* variable, as well, because we will update it in the transaction (line 9). Then, we make the logs durable by flushing them from the caches using *clflushopt* instructions (lines 10–14). The variable *CLEMENTS* represents the number of matrix elements per cache line. We use it to allow performing only one *clflushopt* instruction per cache line. Then, we use an *sfence* as shown in line 15 to ensure that flushing the log is complete (i.e., the log is made durable). After that, in line 18, we set the *insideTx* bit indicating we are about to begin updating the data. Next, we flush it using *clflushopt* followed by an *sfence* to make sure the update to *insideTx* is durable before


```

1  for (kk=0; kk<n; kk+=bsize) {
2      for (ii=0; ii<n; ii+=bsize) {
3          //Step1: Create the undo log
4          for (r=0; r<(ii+bsize); r++) {
5              for (l=0; l<n; l++)
6                  cLog[r][l] = c[r][l];
7          }
8          //Step2: Make the log durable
9          lastPersistedIILog = lastPersistedII ;
10         for (r=ii; r<(ii+bsize); r++) {
11             for (l=0; l<n; l+=CLELEMENTS)
12                 CLFLUSHOPT(&logC[r][l]);
13         }
14         CLFLUSHOPT(&lastPersistedIILog);
15         SFENCE;
16         //Step3: Set the flag marking beginning
17         //of the transaction and make it durable
18         insideTx = true;
19         CLFLUSHOPT(&insideTx);
20         SFENCE;
21         //Step4: Enter the transaction
22         for (jj=0; jj<n; jj+=bsize) {
23             for (i=ii; i<(ii+bsize); i++) {
24                 for (j=jj; j<(jj+bsize); j++) {
25                     sum = c[i][j];
26                     for (k=kk; k<(kk+bsize); k++){
27                         sum += a[i][k]*b[k][j];
28                     }
29                     c[i][j] = sum;
30                 } //end of for j
31             } //end of for i
32         } //end of for jj
33         //Make the transaction durable
34         for (l=ii; l<(ii+bsize); l++){
35             for (r=0; r<n; r+=CLELEMENTS)
36                 CLFLUSHOPT(&c[l][r]);
37         }
38         lastPersistedII = ii;
39         CLFLUSHOPT(&lastPersistedII);
40         SFENCE;
41         //Step5: Unset the flag marking the end
42         //of the transaction and make it durable
43         insideTx = false;
44         CLFLUSHOPT(&insideTx);
45         SFENCE;
46     } // end of ii
47
48     //Log the LastPersistedKK before modifying it
49     lastPersistedKKLog=lastPersistedKK ;
50     CLFLUSHOPT(&lastPersistedKKLog);
51     SFENCE;
52     updatingIndices = true;
53     CLFLUSHOPT(&updatingIndices);
54     SFENCE;
55     lastPersistedII = -bsize;
56     lastPersistedKK = kk;
57     CLFLUSHOPT(&lastPersistedII);
58     CLFLUSHOPT(&lastPersistedKK);
59     SFENCE;
60     updatingIndices = false;
61     CLFLUSHOPT(&updatingIndices);
62     SFENCE;
63 } //end of for kk

```

Fig. 3. Tiled matrix multiplication with logging.

moving to execute the transaction itself (lines 19 and 20). At this point, we would have a durable log of the data we are about to modify. Then, we begin the transaction execution (lines 22–40). After finishing the transaction, we reset the *insideTx* bit and make it durable, indicating that the transaction has completed (lines 43–45). Once we have reset the *insideTx* flag, we no longer need the log. We follow a similar approach for updating the indices (lines 48–63).

4 OUR RECOMPUTE APPROACH

While the *tmm* with logging allows us to avoid creating copies of the matrices, relying on logs still incurs significant performance overheads. In addition, the log must be made durable, thereby adding execution time and write overhead and reducing the write endurance of NVMM.

We propose a new approach that does not rely on durable transactions for updating matrix *c*. Instead, it persists the result matrix as it goes in a non-atomic manner. Hence, while updating the *c* matrix, there is no guarantee of precisely consistent state. However, we do make all updates durable and periodically wait for them to finish. If failure occurs before we have made the updates to *c* durable, it is possible that some elements of the *c* matrix are in an inconsistent state. However, because we continue to atomically update induction variables in the loop nest, we know the exact

```

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 of 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 Mode for Tiled Matrix Multiplication with Recompute.

```

1  //Step1: Zero out the cells we will recompute.
2  firstII = lastPersistedII;
3  lastII = lastPersistedII + bsize;
4  for (i= firstII; i< lastII; i++) {
5      for (j=0; j<n; j++)
6          c[i][j] = 0;
7  }
8  //Step2: Fix cells we failed while computing.
9  //Recompute until the kk loop in which we failed
10     .
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 Mode for Tiled Matrix Multiplication with Recompute.

Fig. 4. Implementation of recompute scheme in the normal execution mode (a), and the recovery mode (b), for tiled matrix multiplication. For simplicity, we show the implementation of the single-threaded version. Changing to the implementation of the multi-threaded version requires very simple modifications to distribute work to different threads.

region of the c matrix that may be inconsistent. If a failure occurs, we discard just this state and recompute their values to restore the state of the matrix back to where it was at the point of the failure. Thus, we make computation faster and simpler, but at a cost of more complex and longer recovery.

Figure 4(a) shows a tiled matrix multiplication code with the *Recompute* approach. In this code, we compute a full ii iteration before making all updated values durable (similar to the durable transaction example). In lines 13 to 17, we make all the c elements we modified in the ii loop durable by

flushing them to NVMM. The variable *CLEMENTS* represents the number of matrix elements per cache line, and it ensures that we perform only one *clflushopt* instruction per cache line. Then, we update the *lastPersistedII* variable to indicate which loop of *ii* we completed, so that we continue from this loop after recovery (lines 18–20). Upon the completion of the *sfence* instruction in line 20, the modified elements of matrix *c* are guaranteed durable. Thus, with *Recompute*, we eliminated the need for logging while updating the *c* elements and the *lastPersistedII* variable. However, we still use logging for updating the indices (lines 22–35). From this technique, we can see that we remove much overhead from the normal execution path but add more burden on the recovery code to recompute all the previous parts of the elements of matrix *c*, from the beginning up until the point of the failure.

Figure 4(b) shows the recovery code. The recovery starts by setting the indices for the range of *c* matrix cells we will have to recompute (lines 2 and 3). This range is determined from the *lastPersistedII* variable that we saved during normal program execution in Figure 4(a). After that, we reset (zero out) all the elements of matrix *c* that need to be recomputed (lines 4–7). This removes any intermediate values in these elements, and prevents any potential consistency problem. After that, we recompute them from the beginning up until the *kk* loop iteration in which the failure occurs (lines 10–16). We determine this *kk* loop iteration from the *lastPersistedKK* variable we saved during normal program execution in Figure 4(a). This will let us return to the state right before the crash happened. Finally, we use a loopnest of *clflushopt* instructions followed by an *sfence* to make the recomputed values durable (lines 18–22).

4.1 An Example of the Recompute Scheme

To have a better understanding of the Recompute scheme and the code shown in Figure 4. This section provides a simplified version of tiled matrix multiplication to navigate through the steps of the Recompute scheme. Figure 5 illustrates the changes to the content of the result matrix (i.e., matrix *C* in Figure 1). For simplicity, we assume that the changes to the matrix happen at the granularity of a row as a persistency region. However, a real implementation would have the granularity of tiles, as implemented in Figure 4. As shown in Figure 5(a), all the cells in the result matrix get updated at every outermost iteration (referred to as the *kk* loop in Figure 4(a)). For any iteration *i*, calculating the value for any cell *c_i* requires reading the input matrices and the value of the same cell in the previous iteration (*c_{i-1}*). The only exception is for the first iteration, which requires only reading the values of the input matrices (i.e., *A* and *B* in Figure 1). In the case of failure-free execution, all the cells will be updated at every iteration until reaching the final iteration (which is iteration 3 in our simplified example). At that point, the values in all the cells will represent the final output of the kernel (illustrated with the uppercase *C* in Figure 5(a)).

To facilitate recovery from a crash, normal execution with Recompute always keeps track of the achieved progress so that it can locate the point from where the recovery should start. This is illustrated in lines 18 and 29 in Figure 4(a). Keep in mind that these variables are persisted immediately during execution, which guarantees that the recovery code will find them after a crash. The example in Figure 5(b) illustrates a case where a failure happens while computing the third row in iteration 2 (i.e., some cells in the row failed to persist *c2*). The recovery code will start the recovery process from the row following the variable *LastRow* and will use the variable *Iteration* to identify the iteration where the crash happened. Thus, it can accurately identify the progress made before the crash.

With these details about the status before the crash, the recovery code of Recompute can reconstruct the persistency region. The recovery steps are shown in Figure 5(c), which are consistent with the steps shown in the code in Figure 4(b). First, the content of the region following *LastRow* will be zeroed out because it may have inconsistent data due to the possibility of not fully

reconstructed on recovery. This sequence of dependencies can go beyond the persistency region boundaries and does not require the region itself to be idempotent. For example, looking at the code shown in Figure 4(a), the *ii* loop chosen to be the persistency region is not idempotent. More specifically, due to the tiling optimization, the region reads the live-in value of $c[i][j]$ (line 6), and then uses this value to generate the result that will be written to the same memory location $c[i][j]$ (line 9) as a live-out of the region. Thus, having this Write-After-Read dependency² in the same region defines the region as not being idempotent [18, 40]. Because of that, simply re-running the same code region when recovering after a crash might lead to incorrect results.

With the Recompute scheme, this code remains an eligible persistent region because the recovery code will be able to reproduce its computation by starting from the input matrices and recalculating all of the iterations until the point where the crash happens (as shown in Figure 4(b)). This means that it is *required* that the original input to the algorithm (e.g., matrices *A* and *B* in Figure 1) not be modified during the execution. Otherwise, the starting point will not be available for re-generating the pre-crash state of the persistency region. Fortunately, many scientific applications are designed in a way that maintains this requirement. Another requirement is that the persistency region is not to have data dependencies with other regions; this allows reconstructing each region independently after a crash. However, this excludes the data dependencies already included in the sequence of dependencies used for recovery.

The requirements mentioned above are even more lenient when compared to the ones for Lazy Persistency [4]; the Recompute scheme doesn't require persistency regions to be associative. This is because each region is guaranteed to be persisted before moving to the next region.

Loop structures are not the only code structure that can satisfy these requirements. However, the loop structure makes it easier to track the program's progress using the loop's progress information (loop iterators, index, etc.), and thus, identifying the moment where the crash happened. We focused on loop-based codes in this article also because loop-based codes are common in scientific workloads. The granularity of the persistency region can be varied by selecting different loops in a loop-nest, for example the inner-most loop or an outer-loop. In this work, we implemented two granularity options for the loop-nest in our experiments in Section 7.

Parallelism is also allowed in the workloads adopting the Recompute scheme. In fact, as discussed in Section 6, all the workloads we studied were multi-threaded. However, we designed the workloads such that each thread is working on its own memory locations, without any dependencies across threads. This allowed all the threads to perform the recovery code independently after the crash. This assumption is consistent with prior works [4, 24].

4.3 Hybrid Recompute/Checkpointing

One limitation of the *Recompute* approach is the potential large amount of work that must be repeated for the elements of matrix *c* that were left in an inconsistent state, in the event that a crash occurs after running for a long time. This may result in a long recovery time. In the worst case, for a huge matrix, the recovery time may approach or even exceed the Mean Time To Failure (MTTF).

To avoid an overly lengthy recovery time, we can periodically save matrix *c* so that if a failure occurs, recomputing matrix elements can commence from the saved copy rather than starting over from the beginning. The recovery code only needs to re-execute from the iteration in which the copy was made until the point of failure. Furthermore, we can devise approaches that infrequently save a copy of each element by spreading it over many iterations of the outermost loop nest. For

²This is true because the read in our case is *exposed* and not protected by any previous write instructions in the same region. A protecting write instruction preceding the read is referred to as a *clobber anti-dependence* [18].

example, if we save $\frac{1}{64}$ of the matrix at every loop, the entire matrix c is fully copied every 64 iterations of the algorithm. We can determine which part of the matrix is scheduled for copying by using a modulo of the iteration index. Similarly, the recovery code can be changed accordingly to determine how much recomputation is needed based on the iteration in which the last copy was taken.

5 COMPILER SUPPORT

While incurring less execution time and write overheads, *Recompute* requires a more complex code transformation and recovery compared to *Checkpointing*. Matrix multiplication and other applications are commonly used in HPC (and beyond) through a library. Thus, a library can implement the Recompute scheme efficiently, allowing end users to benefit from its performance without programming it on their own. Another approach for mitigating the implementation difficulty could be through the addition of language or compiler support to help automate the process. In this section, we investigate one possible compiler support that can achieve this goal.

In the loop-based benchmarks we evaluated, we observed the same transformation steps for achieving failure safety using *Recompute*. First, the programmer needs to provide the compiler with three hints: (1) the outermost loop of the persistence region (i.e., the kk loop in tmm); (2) the variables to be persisted (i.e., matrix c in tmm); and (3) the granularity of persistence (i.e., ii or jj in tmm). By specifying the outermost loop, the compiler infers which loop induction variables need to be logged. By specifying the name of all variables that must be persisted, the compiler can identify which stores must be made durable. By specifying the granularity of persistence, the compiler knows where to insert the cache line flushing instructions to make data durable. It can also validate that all updates to the specified persistent variables are nested within the specified loop and that it can correctly generate a loop that flushes all modified elements to make them durable. These steps must be performed in a compiler pass after other loop transformations (e.g., tiling and unrolling) are considered. An example of how the programmer can provide such hints is through source code-level directives a la OpenMP directives.

Figure 6(a) shows an example of a compiler directive for the recompute approach and its sample code. A pragma statement of directive “recompute” is placed just in front of the outermost loop, where logging code for induction variables will be inserted. The sentinel “NVM” indicates this directive is included in non-volatile memory API. The clause “persist” specifies variables and arrays with a range to be persisted and which loop performs it. For this example, the array c ranging from $c[ii][0]$ to $c[ii+bsize-1][n-1]$ is persisted at the end of every iteration of ii loop. According to the information provided by this directive, a compiler generates appropriate code in the recomputation approach manner.

Figure 6(b) shows an example of the generated code for persisting the c matrix. Loop peeling is employed to adjust the beginning of the array to be persisted in the $clflushopt$ loop. Thus, the generated code can deal with the array even if it is not aligned to a cache line. In the figure, line 5 is responsible for detecting the case where the result matrix is not aligned to a cache line. In this case, the first elements of the row, which do not start from the beginning of a cache line, are peeled off, and they will be flushed (line 11). The variable $r1$ will index the first element of the row (line 10) that begins a new cache line. Otherwise, the variable $r1$ will be initialized to 0 (line 8). For both cases, it will mark the beginning of the innermost loop that performs the flushes (line 14).

6 METHODOLOGY

6.1 Simulation Configuration

We evaluate our methods on a simulator that was built on top of *gem5* [11], an open source cycle-accurate full system simulator. Our simulator uses x86-64 instruction set architecture (ISA). It

<pre> 1 #pragma NVM recompute persist(c[ii: ii + bsize - 1][0:n-1], ii) 2 for (kk = 0; kk < n; kk += bsize) { 3 for (ii = 0; ii < n; ii += bsize) { 4 for (jj = 0; jj < n; jj += bsize) { 5 for (i = ii; i < (ii + bsize); i++) { 6 for (j = jj; j < (jj + bsize); jj++) { 7 sum = c[i][j]; 8 for (k = kk; k < (kk + bsize); kk++) { 9 sum += a[i][k] * b[k][j]; 10 } 11 c[i][j] = sum; 12 } // j 13 } // i 14 } // jj 15 } // ii 16 } // kk </pre>	<pre> 1 // CLSIZE: cache line size in byte 2 // CLELEMENTS: Elements per cache line 3 4 for (l = ii; l < (ii + bsize); l++) { 5 offset = &c[l][0] & (CLSIZE - 1); 6 7 if (offset == 0) { 8 r1 = 0; // It is already aligned 9 } else { 10 r1 = offset / sizeof(c[0][0]); 11 CLFLUSHOPT(&c[l][0]); 12 } 13 14 for (r = r1; r < n; r += CLELEMENTS) 15 CLFLUSHOPT(&c[l][r]); 16 } </pre>
--	---

(a) Tiled Matrix Multiplication code augmented with the proposed compiler directive.

(b) Transformed code, employing loop peeling to handle unaligned matrices in order to ensure only one CLFLUSHOPT instruction is used for each cache block.

Fig. 6. Code example that illustrates compiler directive (a) and the resulting transformed code (b) for tiled matrix multiplication.

Table 1. The Baseline System Configuration

Component	Configuration
Processor	OoO, 2GHz, 4-wide issue/retire ROB: 196, fetchQ/issueQ/LSQ: 48/48/48
L1I and L1D	64KB, 8-way, 64B block, 2 cycles
L2	512KB, 8-way, 64B block, 11 cycles
NVMM latencies	60ns read, 150ns write

models a detailed four-way out-of-order processor pipeline for each core, with parameters shown in Table 1. The table also shows parameters for the two-level caches that we model, with private per-core L1 caches and an L2 cache shared by all cores. The memory hierarchy model is built on top of Ruby [1], with MESI protocol keeping the L1 caches coherent with respect to one another and with respect to the shared L2 cache. Beyond the L2 cache, the main memory is NVMM with 60ns read latency and 150ns write latency. These latencies are in line with many NVMM studies [9, 14, 35, 39, 55]. We note that if the latencies are higher, the relative benefits of our *Recompute* and *Hybrid Recompute* schemes will increase because they incur the fewest writes to the NVMM.

We implemented Intel PMEM instructions, namely *clflushopt* (*clflush* is implemented but unused). The ordering constraints of *clflushopt* are implemented as described in the Intel manual [17]. Specifically, *clflushopt* is ordered only with respect to memory fences (including *sfence* and *mfence*) and with respect to older loads/stores to the same cache line address. Similar to stores, *clflushopt* accesses the cache after it is retired from the CPU pipeline. Moreover, *clflushopt* becomes durable and the instruction completes when the dirty cache block has been written back to the buffer in the memory module.

Table 2. Real System Used in Evaluation

Component	Configuration
Processor	32 CPUs, AMD Opteron(TM) Processor 6272, 2.1GHz
L1I and L1D	16KB and 64KB, respectively. 64B block
L2	2MB, 64B block
L3	6MB, 64B block
Main Memory	32GB DRAM

Table 3. Multiple Approaches Applied to the Tiled Matrix Multiplication to Achieve Write Durability of the Resultant Matrix

Variant	Description
tmm	Tiled matrix multiplication
tmm+CP	Checkpointing (1 checkpoint)
tmm+R_ii	Recompute (ii granularity)
tmm+R_jj	Recompute (jj granularity)
tmm+L_ii	Transaction with Logging (ii granularity)
tmm+L_jj	Transaction with Logging (jj granularity)
tmm+HR_ii_x32	Hybrid Recompute (ii granularity), checkpoint interval 32× of tmm+CP
tmm+HR_jj_x32	Hybrid Recompute (jj granularity), checkpoint interval 32× of tmm+CP
tmm+HR_ii_x64	Hybrid Recompute (ii granularity), checkpoint interval 64× of tmm+CP
tmm+HR_jj_x64	Hybrid Recompute (jj granularity), checkpoint interval 64× of tmm+CP

The matrix is 1024×1024 in size, and the tile size is 16 in order to align them to the cache block size. Thus, to persist one stride, only one *clflushopt* is required.

6.2 Real Machine Configuration

In addition to the evaluation on the Gem5 simulator, we evaluated Naive Checkpointing and Recompute scheme on a real system (shown in Table 2). As shown in the table, the machine used is DRAM-based (i.e., not NVMM-based) because NVMM-based systems are not commercially available yet. Thus, we replaced cache persistency instructions (e.g., CLFLUSHOPT, CLWB, etc.)³ with legacy CLFLUSH instruction. CLFLUSH has different ordering rules compared to CLFLUSHOPT; hence, the absolute performance results would be different than what our simulation experiments would report. However, we think that it is useful for comparing schemes (i.e., Naive Checkpointing and Recompute scheme) because the instruction is applied consistently across all the schemes. We used the originally available SFENCE instruction for persistency barrier.

6.3 Benchmarks

Table 3 shows multiple approaches applied to the tiled matrix multiplication that we evaluated. *tmm* is the baseline tiled matrix multiply with no persistence or checkpointing at all. *tmm_CP* is the *Checkpointing* approach. *tmm+R*, *tmm+L*, and *tmm+HR* are our *Recompute*, *Logging*, and *Hybrid Recompute* schemes, respectively. We evaluated two persistence granularities, namely *ii* loop granularity and *jj* loop granularity, with the *ii* granularity being the default (e.g., in Figure 4(a)). For most experiments, we run the matrix multiplication with eight worker threads plus one master thread. They run on nine cores.

³PCOMMIT instruction was not needed anyway because it has been deprecated even in the Gem5 simulation.

Table 4. Summary of All the Benchmarks We Evaluated

Benchmark	Description
TMM	1k-squared input tiled matrix multiplication
FFT	1-million nodes vector FFT
LU	1k-square input matrix LU decomposition
Gauss	4k-square input matrix Gauss elimination
2D-Conv	1k-square input matrix 2D convolution
Cholesky	1k-square input matrix Cholesky factorization

Other benchmarks that we evaluated include LU decomposition (LU), Fast Fourier Transform (FFT), Gaussian elimination (Gauss), and two-dimensional convolution (2D-conv). These benchmarks constitute kernels that are frequently used in the high performance computing domain and beyond. They are taken from Refs. [2, 56, 58] and the Splash-2 benchmark suite [57]. Table 4 shows the input that we use.

In the simulation-based evaluation, we simulate all the schemes over the same number of loop iterations to ensure that each of the designs performs the same amount of work during the simulation window. Approximately, the simulation window includes 250 million instructions, on average, to warm up the caches and other structures, and we simulate and report the timing for an additional 300 million instructions, on average.

For the real machine part, we ran all the benchmarks from the beginning to the end of the kernel, which is far longer than the simulation window used for the Gem5 simulation part. Overall, the number of instructions in the real machine experiments is about 40–60 $\times$ higher than the number of instructions simulated in the Gem5 simulation part.

7 EVALUATION

We evaluate the execution time overhead of *Checkpointing* and compare it against *Logging* and our *Recompute* and *Hybrid Recompute* schemes. We modeled *Checkpointing* optimistically by only counting the time to store all three matrices (*a*, *b*, and *c*) to NVMM, while ignoring the time to write to the file system, context switching, register file saving, and the precise interrupt. In addition, we further reduced the time it takes to write the checkpoint to the NVMM in our *Checkpointing* model by using the x86–64 SSE quadword store instruction, which is an SIMD vector store.

7.1 Execution Time and Number of Writes

Table 5 shows the execution time and number of writes for all of the schemes we study, normalized to the base tiled matrix multiplication, which is not failure safe. The number of writes represents the number of L2 writebacks plus cache line flushes.

As can be seen in the table, *Checkpointing* (tmm+CP) more than triples the execution time (3.07 $\times$) and quadruples NVMM writes (4.3 $\times$). This is because a snapshot of the matrices are copied to another memory location in the NVMM. The number of writes *Checkpointing* incurs is troublesome since NVMM often has limited write endurance. *Logging* (tmm+L) incurs acceptable small execution time overheads: 8% and 9% for the *ii* and *jj* granularities, respectively. However, *Logging* causes a significant increase in the number of writes: 2.11 $\times$ & 2.23 $\times$ for the *ii* and *jj* granularities, respectively. Although lower than those in *Checkpointing*, the number of writes overheads are still problematic for NVMM write endurance. Note that the *ii* granularity yields lower execution time and write overheads compared to the *jj* granularity. This is expected as the *ii* loop envelopes the inner *jj* loop. *Recompute* simultaneously achieves low execution time overheads (5% and 6% for the

Table 5. Execution Time and Number of Write Comparison between *Checkpointing* (tmm+CP) and Different Granularity of Our *Logging* (tmm+L), *Recompute* (tmm+R), and *Hybrid Recompute* (tmm+HR) Schemes, Normalized to the *Base Tiled Matrix Multiplication* (tmm)

Scheme	Exe Time	Num Writes
base (tmm)	1.00	1.00
tmm+CP	3.07	4.30
tmm+L_ii	1.08	2.11
tmm+L_jj	1.09	2.23
tmm+R_ii	1.05	1.07
tmm+R_jj	1.06	1.15
tmm+HR_ii_x32	1.08	1.13
tmm+HR_ii_x64	1.07	1.08
tmm+HR_jj_x32	1.09	1.20
tmm+HR_jj_x64	1.08	1.16

ii and *jj* granularities, respectively) and low write overheads (7% and 15% for *ii* and *jj* granularities, respectively). This shows that for loop-based code, it is possible to achieve failure safety without incurring much execution time or write overheads.

Finally, we also evaluated the *Hybrid Recompute* scheme with 32 $\times$ and 64 $\times$ the checkpoint interval of tmm+CP (both denoted by the tmm+HR prefix). As expected, less frequent checkpointing reduces both execution and write overheads: 7% with tmm+HR_ii_x64 vs. 8% for tmm+HR_ii_x32. The same observation applies for the *jj* granularity as well. Compared to *Recompute*, *Hybrid Recompute* with 64 $\times$ checkpointing interval incurs 1–2% slightly higher execution time and write overheads. However, the recompute effort during failure recovery is much lower now and is bounded by 64 iterations. Compared to *Checkpointing*, the overhead is much lower since only a small part of matrix *c* is copied at each iteration.

7.2 Sensitivity to Checkpointing Frequency

The high execution time and number of writes overheads for *Checkpointing* are closely related to the frequency of taking checkpoints. On a system that takes frequent checkpoints, execution time and write overheads will be higher than on a system that takes checkpoints less frequently. In contrast to *Checkpointing*, the overheads of *Logging*, *Recompute*, and *Hybrid Recompute* are constant (i.e., we incur this overhead every loop iteration). Thus, a question arises: can the frequency of taking checkpoints be reduced such that the execution time overhead for *Checkpointing* is equal to or lower than *Hybrid Recompute*?

To answer this question, we plot the execution time of *Checkpointing* normalized to *Hybrid Recompute* at *ii* granularity at 64 $\times$ the checkpointing frequency, as we vary the number of *kk* loop iterations. This is shown in Figure 7. We note that the execution time speedup ratio of *Checkpointing* decreases *inversely proportionally* as the number of *kk* loop iteration increases, as the checkpoint creation overhead is amortized across more iterations, whereas it remains constant for tmm+HR. While we have not collected results for a large number of *kk* iterations, we extrapolated the tmm+CP curve using a least square method. The intersection of the two curves is at 42. Note, however, that this number is not exact as we relied on extrapolation. The key point, though, is that

Fig. 7. Extrapolation of execution time overheads normalized to *base* for *Checkpointing* (tmm+CP) and our ii-granularity *Hybrid Recompute* with matrix saving frequency of 64x.

Table 6. Normalized Execution Time as the Thread Count Varies from 1 to 12 Threads, for Various Schemes

Scheme	1T	4T	8T	12T
base (tmm)	1.00	1.00	1.00	1.00
tmm+CP	1.88	2.85	3.07	3.27
tmm+L_ii	1.30	1.25	1.08	1.08
tmm+L_jj	1.36	1.26	1.09	1.10
tmm+R_ii	1.15	1.12	1.05	1.04
tmm+R_jj	1.18	1.14	1.06	1.05
tmm+HR_ii_x32	1.21	1.17	1.08	1.07
tmm+HR_ii_x64	1.21	1.16	1.07	1.06
tmm+HR_jj_x32	1.26	1.17	1.09	1.07
tmm+HR_jj_x64	1.26	1.16	1.08	1.06

there is a loop iteration count N such that checkpointing may become cheaper than *Recompute*, and N is likely quite high.

Repeating the same methodology for the number of writes, the parity point between the write overhead of *Checkpointing* and *Hybrid Recompute* occurs at 155 *kk* loop iterations, which is significantly higher than the ratio we observed for the execution time overhead.

Overall, system designers need to take these intersection points into account when choosing an approach to achieve failure safety. If they are high (especially when they approach or exceed MTTF), our *Recompute* or *Hybrid Recompute* are more attractive. However, when the intersection points occur at a low iteration count, *Checkpointing* is acceptable.

7.3 Sensitivity to Thread Count

Tables 6 and 7 show the execution time and number of writes overhead, respectively, for running the different schemes with different number of threads (thread count). These overheads are normalized to the base tiled matrix multiplication (tmm) running with the respective thread count. We evaluated each scheme's sensitivity to thread count for 1, 4, 8, and 12 threads, while adjusting the number of cores to scale with the thread count. All runs in Tables 6 and 7 are for the same amount of work (i.e., running the same number of iterations of the tiled matrix multiplication).

Table 6 shows that *Checkpointing* (tmm+CP) incurs higher execution time as thread count increases. This is mainly because it requires all threads to synchronize before it can take the checkpoint. While the base execution time decreases as thread count increases due to increasing

Table 7. Normalized Number of NVMM Writes as the Thread Count Varies from 1 to 12 Threads, for Various Schemes

Scheme	1T	4T	8T	12T
base (tmm)	1.00	1.00	1.00	1.00
tmm+CP	4.10	4.09	4.03	3.65
tmm+L_ii	2.12	2.05	2.04	1.71
tmm+L_jj	2.04	2.12	2.23	1.43
tmm+R_ii	1.03	1.03	1.07	1.02
tmm+R_jj	1.03	1.10	1.15	1.09
tmm+HR_ii_x32	1.09	1.09	1.13	1.06
tmm+HR_ii_x64	1.04	1.05	1.08	1.04
tmm+HR_jj_x32	1.09	1.17	1.21	1.19
tmm+HR_jj_x64	1.05	1.12	1.16	1.12

parallelism, the checkpoint creation time remains the same, and the synchronization time increases slightly. Thus, relative to the base execution time, the overhead from checkpoint creation increases.

The execution time overheads for *Logging* (tmm+L), *Recompute* (tmm+R), and *Hybrid Recompute* (tmm+HR) slightly decrease as the number of threads increases. This is true for both *ii* and *jj* granularities. Comparing across schemes, *Recompute* holds on to its execution time overhead advantage compared to other schemes. For 12 threads, its execution time overhead is only 4%, vs. 8% for *Logging* and 6% for the *Hybrid Recompute*. All other previous observations remain, i.e., the *ii* granularity produces less overheads vs. the *jj* granularity, and 64× *Hybrid Recompute* incurs less overheads compared to its 32× counterpart.

Regarding the write overheads, Table 7 does not show any strong trend affected by the thread count.

7.4 Other Benchmarks

We demonstrate similar results on the other workloads we studied. Figure 8 shows the normalized execution time and number of writes for four other benchmarks and their geometric means, with each benchmark running with eight threads. Figure 8(a) shows *Recompute* incurring execution time overheads between 1–7% (averaging 3%), compared to a range of 31–507% (averaging 91%) for *Checkpointing*. Figure 8(b) shows *Recompute* incurring NVMM write overheads between 2–30% (averaging 8%), compared to a range of 8–71% (averaging 38%) for *Checkpointing*. The benefit from *Recompute* differs from one benchmark to another, depending on the nature of the benchmark. For *Gauss*, the savings in execution time from *Recompute* is significantly higher than the savings in the number of writes. The reason for this is that *Gauss* has decreasing computation effort per iteration as the program runs. This makes the execution time overhead of *Checkpointing* higher compared to *Recompute*.

7.5 Evaluation on Real Hardware

In addition to the simulation-based evaluation, we evaluated the work on a real machine with configuration shown in Table 2 (Section 6). Using a real machine allows us to run the benchmarks from the beginning to the end, which is a far longer execution window compared to what is feasible with simulation evaluation. Investigating application behavior from start to completion made it

Fig. 8. Execution time (top) and number of writes (bottom) overheads for different benchmarks. For each benchmark, we compare *Checkpointing* and our *Recompute* schemes normalized to *base* (base is the model without persistence or checkpointing). The figure also shows the geometric mean (gmean) for each scheme.

possible to get clearer insights. On top of that, if the results from real machine evaluation are consistent with the results from the simulator, the simulation evaluation is validated.

7.5.1 Checkpointing Frequency Analysis. The discussion in Section 7.2 raised an important insight into the comparison of the overheads of Naive Checkpointing and Recompute Scheme. Understanding this comparison is crucial for deciding which scheme is better for a specific situation. This decision can be made by finding the checkpoint frequency where the overheads for the two schemes intersect. If the checkpointing frequency is higher than the intersection point, *Recompute* is cheaper than checkpointing. On the other hand, if the checkpointing frequency is lower than the intersection point, the traditional checkpointing is cheaper than *Recompute*.

Figure 9 shows the execution time overheads of *Recompute* versus checkpointing, the latter plotted as a function of checkpointing frequency that varies from 1 to 256 checkpoints for the entire run of each application. Such results are not feasible to get using simulations because that would take too much time, as described earlier. In contrast, running on a real machine allows the results to be collected due to our ability to run applications from start to completion. Note that these curves rely on DRAM-based memory; we expect that the execution time overheads will be higher for checkpointing on actual NVMM-based systems due to the limited write bandwidth available in such systems. Thus, the overhead difference between checkpointing and *Recompute* is underestimated on DRAM-based systems because *Recompute* requires much lower write bandwidth than checkpointing.

As expected, Figure 9 shows that increasing the checkpointing frequency causes an increase in the execution time overheads of checkpointing due to the time spent in creating the checkpoints. The execution overheads of *Recompute* do not depend on checkpointing frequency as no checkpoint is created.

The figure shows that the impact of changing the checkpointing frequency on the execution time overhead is not identical across the benchmarks. Some benchmarks are very sensitive to checkpointing frequency; even creating one checkpoint incurs higher execution time overheads than *Recompute* (e.g., FFT and Gauss). At the other extreme, one benchmark (LU) does not incur much execution time overheads from checkpointing; only when 16 checkpoints are created does the execution time overhead from checkpointing exceed that of *Recompute*. All the remaining benchmarks

Fig. 9. Execution time overheads for *Recompute* and Naive Checkpointing, as affected by checkpointing frequency for Naive Checkpointing. Both of the curves are normalized to the base version of the corresponding benchmark where no failure safety schemes are implemented (lower is better).

(e.g., TMM, 2D Convolution, and Cholesky) show execution time overheads of checkpointing that are slightly lower if very few checkpoints are created, but show significantly higher overheads as the checkpointing frequency increases.

Several factors contribute to how *Recompute* compares vs. checkpointing as checkpointing frequency changes. First, the time it takes to create a checkpoint is determined by the amount of data that needs to be checkpointed. In a benchmark with a high memory footprint, creating a checkpoint takes a long time; hence, such a benchmark is more sensitive to the increase in checkpointing frequency. Second, how well *Recompute* does compared to the base matters. Even if the time for creating a checkpoint is relatively modest, it is difficult for Naive Checkpointing to compete with *Recompute* if the latter incurs negligible overheads compared to the base (e.g., 2D-Convolution). On the other hand, when *Recompute* suffers from high overheads compared to the base, Naive Checkpointing can enjoy higher checkpointing frequency without making it more expensive than *Recompute* (e.g. LU). Table 8 summarizes the benchmarks studied and categorizes each benchmark based on the the intersection point between *Recompute* and Naive Checkpointing.

To sum up our observations, the decision of which scheme to use is not a straightforward one; it requires analysis of the benchmarks and the minimum number of checkpoints based on the MTTf of the studied system. However, as shown in Figure 9, *Recompute* outperforms Naive

Table 8. Summary of the Studied Benchmarks

Category: When CP overhead is higher?	Benchmarks
Always	FFT
	2D-Convolution
	Gauss
After a SMALL number of checkpoints (shown in parentheses)	TMM (3) Cholesky (2)
After a LARGE number of checkpoints (shown in parentheses)	LU (16)

Categorizing the benchmarks according to the intersection point between the curves for Naive Checkpointing and Recompute Scheme. This defines at which checkpointing frequency Naive checkpointing will become more expensive than the Recompute scheme.

Checkpointing in most of the studied benchmarks. Furthermore, for half of these benchmarks (the first category in Table 8), *Recompute* is a better choice regardless of the checkpointing frequency. For these benchmarks, the decision is relatively easy to arrive at because creating even one checkpoint will be more expensive than the execution time overheads with *Recompute*.

7.5.2 Thread Count Analysis. Another benefit of experimenting on a real machine is that it allows a better understanding of the scalability of these benchmarks as the number of threads increases in the whole kernel's lifetime. Even though the analysis in Section 7.3 gave important insights about the scalability of the proposed schemes, it was still limited by the small execution window feasible for simulation studies. In this section, we show scalability results for all benchmarks running in parallel from start to finish.

Figure 10 illustrates the speedup comparison for all the studied benchmarks when the number of threads varies from 1 to 32. As can be seen in the figure, *Recompute* scales almost as effectively as base for all the benchmarks. This confirms the observation from the simulation evaluation described in Section 7.3 for TMM. Furthermore, the figure confirms that *Recompute* is just as effective and scalable in other benchmarks as well.

The Gauss benchmark (Figure 10(e)) is one of the benchmarks that does not show good scaling with an increasing number of threads. This is because it has a decreasing amount of computation per iteration as the program progresses, causing significant load imbalance between threads. This behavior is observed only as the evaluation covers the final stages of the workload where the amount of computation becomes very small, which couldn't be reached without the real-hardware evaluation.

8 RELATED WORK

Checkpoint/Restart (C/R) is a traditional approach to achieve fault tolerance. In C/R, the computational state of the machine that constitutes a full checkpoint, such as the PC, register file, and the address space are saved periodically to stable storage. When a machine crashes and subsequently restarts, it restores the checkpoint by copying the saved structures back into memory before it can resume execution. Since the failure usually occurs between checkpoints, some work must be re-run to recover the state of the system back to the point of the failure. A key disadvantage of the C/R approach is its significant performance overhead [52]. C/R overhead can be reduced in many ways. For example, checkpoints can be compressed [26] to reduce the time to write them to storage. Faster secondary storage options, like NVM [22, 33], can reduce the overhead of copying

Fig. 10. Impact of the number of parallel threads on speedup for base (non-failure safe parallel version) vs. our *Recompute* scheme, on a real machine. All speedup numbers are normalized to the base case with a single thread of the corresponding benchmark (higher is better).

checkpoints. Multi-level checkpointing [46] can reduce the frequency and overhead of copying checkpoints to slower disk-based storage. However, C/R still poses significant overhead.

Application-Level Checkpointing (ALC) provides an improvement over C/R. It exploits the observation that most iterative scientific applications have certain key data structures or variables from which the computational state of the program can be recovered and resumed. For example, in order to restore an *n-body* application, we only need to save the positions and velocities of all the particles; therefore, ALC would checkpoint only these key data structures in each checkpoint [12]. Programmer instrumentation is needed to determine good points in the program to save a checkpoint of the key data structures or variables. This approach has two main advantages over C/R. First, it is not machine or OS specific. Second, it can significantly reduce the amount of checkpointed data by saving only the necessary ones. Reducing the data copied to secondary storage significantly improves the execution time overhead and reduces the memory needed to take the checkpoint compared to the C/R approach [12].

Persistency Models for NVMM. Because of their attractive characteristics, Non-Volatile Memories have attracted many researchers to investigate better ways of utilizing their features. Studies in NVMM include important aspects such as security [7, 8, 49] and memory organization and

acceleration [5, 51, 54]. Benefiting from NVMM for failure safety requires defining proper rules for ordering when store values reach the NVMM (i.e., become durable). To achieve that, several persistency models were proposed in the literature. Strict persistency [48] is the most conservative model where durability ordering of stores is identical to their program order. Other, more relaxed, persistency models have also been proposed, including epoch persistency [16, 31], buffered epoch persistency [16, 31], and strand persistency [48]. Logging is usually used to provide atomicity for a transaction so that all stores in a transaction persist together or none at all [15, 32, 53, 55]. *Lazy Persistency* (LP) [4], a recent persistency technique, provides a different persistency model, where durability ordering is completely relaxed in normal execution, resulting in substantial speedups. Memory updates are allowed to reach the NVMM through natural cache evictions. Execution is split into regions protected with checksums capable of detecting persistency failure when recovery is triggered. During recovery, LP relies on the *Recompute* scheme for fixing any regions that did not fully persist prior to failure.

Recovery by Resumption. Several works in the literature have investigated exploiting some features in the running application to facilitate recovering after a crash. One of the key features used in many of these works is *idempotency*. A region, or sequence of instructions, is considered idempotent if running the region once will have an identical effect as running it multiple times. The works proposed by Mahlke et al. [44] and Bershard et al. [10] are the first papers that exploited the idempotency characteristic in their solutions. Kruijf et al. proposed compiler and architectural techniques that exploit idempotency [19–21]. In particular, they proposed a compiler technique [21] that identifies idempotent regions in a program with minimum overhead, and it is the basis for recent work that supports failure-atomic sections and recovery for NVM [41].

Liu et al. proposed Bolt [42], a compiler-directed soft error recovery scheme that provides two advantages over the prior idempotent-based solutions: first, it guarantees recovery without expensive hardware support. Second, it reduces the performance overhead introduced by prior works. Bolt makes this possible with two techniques. (1) Eager Checkpointing guarantees recoverability by *eagerly* checkpointing the values for the region’s input registers immediately after they are defined. (2) Checkpoint Pruning reduces the checkpointing overhead by identifying checkpoints that are safe to remove and that can be reconstructed using other checkpoints.

Moreover, JUSTDO Logging [29] is a recent failure atomicity technique that supports fine-grained concurrency through lock-based Failure-Atomic SEctions (FASE). In contrast to regular REDO and UNDO logging mechanisms, JUSTDO logging doesn’t discard any changes made during FASEs when recovering from a crash. Instead, JUSTDO simply resumes execution from the last store instruction, which exploits the idempotence of a single store operation. JUSTDO logging requires that each store be logged and the log persisted before the corresponding store is persisted. Such logging is necessary to keep track of the last store instruction that will need to be resumed in recovery. Maintaining such a requirement would be very expensive in a conventional system with volatile caches. Because of that, JUSTDO assumes a system with persistent caches, which is something not required in prior works [13, 25] and unlikely to be included in commercial processors in the near future.

To broaden the applicability of JUSTDO logging, Liu et al. proposed iDO [41]. iDO also supports concurrency using lock-based FASEs, but iDO persists store instructions in a coarser granularity as compared to JUSTDO. This was made possible because iDO doesn’t log information at each store instruction. Instead, iDO divides the FASE into several idempotent code regions and logs information at the beginning of each idempotent region within the FASE. This significantly reduced the logging overhead and made JUSTDO logging practical on a conventional system with volatile caches.

Our Recompute scheme is similar to the aforementioned techniques in that it also recovers by resuming from the point of failure. However, there are several points that distinguish our Recompute scheme. (1) Even though the Recompute scheme requires choosing a persistency region, it doesn't require this region to be idempotent, which is unlike the case for Refs. [29], [41], and [42]. As discussed in Section 4.2, the chosen persistency region (*ii* region) is not idempotent, but it can be reconstructed on recovery from the original input. This allows more flexibility in forming persistency regions. (2) This reconstruction technique might look similar to the Checkpoint Pruning technique used in Bolt [42]. However, the solution in Bolt can discover reconstruction opportunities with limited backtracking depth, which will likely miss many opportunities in scientific applications since the data necessary for reconstruction usually goes back to the program's input. On the other hand, the Recompute scheme sees these opportunities since it relies on the algorithm-level idempotency. In fact, it was shown in Ref. [42] that Bolt is not very effective in the cases of loops-based workloads. In contrast, these are the best workloads for the Recompute scheme. (3) The Recompute scheme focuses on scientific applications, which are heavily used in HPC systems. For example, 80% of the execution time in the deep neural network is spent in the convolution layer [30, 47]. Recovery by resumption on scientific applications is not thoroughly studied in the context of NVM. Scientific applications are usually loop-based workloads. Because of that, this article focused on solving problems related to loops. (4) All the other works required some compiler-level or hardware-level support. We believe that NVM can allow programmers to exploit their algorithm-level understanding for providing more relaxed persistency without the need for any other underlying support. Although we investigated providing compiler support for the Recompute scheme in Section 5, we emphasize that this compiler support is *not required* for using the Recompute Scheme. Instead, it is only one of the possible ways to make writing programs with the Recompute scheme more convenient for programmers.

9 CONCLUSION

In this work, we studied several approaches for checkpointing on loop-based codes on NVMM. We observed that *Logging* applied to a tiled loop increases the number of writes to NVMM significantly, hence, reducing write endurance of NVMM.

Our new approach is based on the novel observation that inconsistent state can be tolerated to gain both performance and reduce the number of writes to NVMM for loop-based code. Rather than logging all of the state modified during a transaction, which incurs large overheads, we only log enough state to enable recomputation. We also optimize our recompute-based approach to avoid recomputing from the beginning, and we show this incurs little overhead.

We compare our new approach against *Logging* and *Checkpointing* on five scientific workloads, including tiled matrix multiplication, by running on the gem5 simulator with support for the Intel PMEM instruction extension. For tiled matrix multiplication, we find that our *Recompute* approach has an execution time overhead of only 5% compared to an 8% overhead with *Logging* and 207% overhead with *Checkpointing*. Furthermore, *Recompute* incurs only 7% additional NVMM writes, compared to 111% and 330% more with *Logging* and *Checkpointing*, respectively. Other workloads show similar trends. Hence, *Recompute* simultaneously achieves good execution time performance and does not adversely affect NVMM write endurance. The evaluation section ended with showing results when experimenting on real hardware. The provided results are consistent with the overall findings in the article. Moreover, the real hardware experiments provided deeper analysis on the impact of changing the frequency of checkpointing. As a result, we were able to suggest the best Checkpoint/Restart scheme to use for each workload.

REFERENCES

- [1] 2016. Ruby Memory System. Retrieved from <http://gem5.org/Ruby>.
- [2] Song Ho Ahn. 2005. Convolution. Retrieved from <http://www.songho.ca/dsp/convolution/convolution.html>.
- [3] Hiroyuki Akinaga and Hisashi Shima. 2010. Resistive random access memory (ReRAM) based on metal oxides. *IEEE J.* 98, 12 (2010), 2237–2251.
- [4] M. Alshboul, J. Tuck, and Y. Solihin. 2018. Lazy persistency: A high-performing and write-efficient software persistency technique. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 439–451.
- [5] Amro Awad, Sergey Blagodurov, and Yan Solihin. 2016. Write-aware management of NVM-based memory extensions. In *Proceedings of the 2016 International Conference on Supercomputing (ICS'16)*.
- [6] Amro Awad, Brett Kettering, and Yan Solihin. 2015. Non-volatile memory host controller interface performance analysis in high-performance I/O systems. In *Proceedings of International Symposium on Performance Analysis of Systems and Software (ISPASS)*.
- [7] Amro Awad, Pratyusa Manadhata, Stuart Haber, Yan Solihin, and William Horne. 2016. Silent shredder: Zero-cost shredding for secure non-volatile main memory controllers. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*.
- [8] Amro Awad, Yipeng Wang, Deborah Shands, and Yan Solihin. 2017. ObfusMem: A low-overhead access obfuscation for trusted memories. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.
- [9] F. Bedeschi, et al. 2004. An 8Mb demonstrator for high-density 1.8V phase-change memories. In *Proceedings of the International Symposium on VLSI Circuits*.
- [10] Brian N. Bershad, David D. Redell, and John R. Ellis. 1992. Fast mutual exclusion for uniprocessors. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*.
- [11] N. Binkert, et al. 2011. The GEM5 simulator. *ACM SIGARCH Computer Architecture News (CAN)* (2011).
- [12] G. Bronevetsky, D. Marques, K. Pingali, P. K. Szwed, and M. Schulz. 2004. Application-level checkpointing for shared memory programs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
- [13] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA'14)*.
- [14] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. 2015. REWIND: Recovery write-ahead system for in-memory non-volatile data-structures. *Proc. VLDB Endow.* 8, 5 (Jan. 2015), 497–508.
- [15] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-heaps: Making persistent objects fast and safe with next-generation non-volatile memories. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
- [16] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*.
- [17] Intel Corp. 2016. Intel 64 and IA-32 Architectures Developer's Manual: Vol. 3A.
- [18] Marc de Kruijf and Karthikeyan Sankaralingam. 2011. Idempotent processor architecture. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*.
- [19] Marc de Kruijf and Karthikeyan Sankaralingam. 2011. Idempotent processor architecture. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. ACM, New York, NY, 140–151. DOI: <https://doi.org/10.1145/2155620.2155637>
- [20] Marc de Kruijf and Karthikeyan Sankaralingam. 2013. Idempotent code generation: Implementation, analysis, and evaluation. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (CGO'13)*.
- [21] Marc A. de Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. 2012. Static analysis and compiler design for idempotent processing. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*.
- [22] 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 International Conference on High Performance Computing Networking, Storage and Analysis (SC)*.
- [23] 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.
- [24] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. 2018. Persistency for synchronization-free regions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 46–61.

- [25] Terry Ching-Hsiang Hsu, Helge Brügger, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. 2017. NVthreads: Practical persistence for multi-threaded applications. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys'17)*.
- [26] 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 (2015), 387–402.
- [27] Intel. 2016. Persistent Memory Programming. Retrieved from <http://pmem.io>.
- [28] Intel and Micron. 2015. Intel and Micron Produce Breakthrough Memory Technology.
- [29] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-atomic persistent memory updates via JUSTDO logging. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*.
- [30] Yangqing Jia. 2014. *Learning Semantic Image Representations at a Large Scale*. Ph.D. Dissertation.
- [31] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2015. Efficient persist barriers for multicores. In *Proceedings of International Symposium on Microarchitecture (Micro)*.
- [32] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra. 2017. ATOM: Atomic durability in non-volatile memory through hardware logging. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.
- [33] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic. 2013. Optimizing checkpoints using NVM as virtual memory. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*.
- [34] T. Kawahara, R. Takemura, K. Miura, J. Hayakawa, S. Ikeda, Y. Lee, R. Sasaki, Y. Goto, K. Ito, T. Meguro, F. Matsukura, H. Takahashi, H. Matsuoka, and H. Ohno. 2007. 2Mb spin-transfer torque RAM (SPRAM) with bit-by-bit bidirectional current write and parallelizing-direction current read. In *Proceedings of the International Solid-State Circuits Conference (ISSCC)*.
- [35] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. 2016. NVWAL: Exploiting NVRAM in write-ahead logging. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*. ACM, New York, NY, 385–398.
- [36] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. 2016. High-performance transactions for persistent memories. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
- [37] Emre Kultursay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. 2013. Evaluating STT-RAM as an energy-efficient main memory alternative. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*.
- [38] Benjamin C. Lee. 2010. Phase change technology and the future of main memory. *IEEE Micro* (2010).
- [39] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building durable transactions with decoupling for persistent memory. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*. ACM, New York, NY, 329–343.
- [40] 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 *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.
- [41] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L. Scott, Sam H. Noh, and Changhee Jung. 2018. iDO: Compiler-directed failure atomicity for nonvolatile memory. *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2018).
- [42] Q. Liu, C. Jung, D. Lee, and D. Tiwari. 2016. Compiler-directed lightweight checkpointing for fine-grained guaranteed soft error recovery. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.
- [43] Youyou Lu, Jiwu Shu, Long Sun, and Onur Mutlu. 2014. Loose-ordering consistency for persistent memory. In *Proceedings of the International Conference on Computer Design (ICCD)*.
- [44] Scott A. Mahlke, William Y. Chen, Wen-mei W. Hwu, B. Ramakrishna Rau, and Michael S. Schlansker. 1992. Sentinel scheduling for VLIW and superscalar processors. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*.
- [45] C. Mohan, D. Haderle, B. Lindsay, et al. 1992. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)* (1992).
- [46] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R. de Supinski. 2010. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*.
- [47] K. Osawa, A. Sekiya, H. Naganuma, and R. Yokota. 2017. Accelerating matrix multiplication in deep learning by using low-rank approximation. In *2017 International Conference on High Performance Computing Simulation (HPCS)*.
- [48] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory persistency. In *Proceedings of International Symposium on Computer Architecture (ISCA)*.

- [49] M. K. Qureshi. 2011. Pay-as-you-go: Low-overhead hard-error correction for phase change memories. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.
- [50] Raghunath Rajachandrasekar, Sreeram Potluri, Akshay Venkatesh, Khaled Hamidouche, Md. Wasi ur Rahman, and Dhabaleswar K. (DK) Panda. 2014. MIC-check: A distributed checkpointing framework for the Intel many integrated cores architecture. In *Proceedings of the International Symposium on High-performance Parallel and Distributed Computing (HPDC)*.
- [51] Mohit Saxena and Michael M. Swift. 2010. FlashVM: Virtual memory management on flash. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (USENIXATC'10)*.
- [52] B. Schroeder and G. A. Gibson. 2007. Understanding failures in petascale computers. *Journal of Physics* 78, 1 (2007), 12–22.
- [53] 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 Annual IEEE/ACM International Symposium on Microarchitecture*.
- [54] Seunghye Shin, James Tuck, and Yan Solihin. 2017. Hiding the long latency of persist barriers using speculative execution. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*.
- [55] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight persistent memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
- [56] M. E. Wolf and M. S. Lam. 1990. A data locality optimizing algorithm. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*.
- [57] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of International Symposium on Computer Architecture (ISCA)*.
- [58] S. C. Woo, J. P. Singh, and J. L. Hennessy. 1994. The performance advantages of integrating block data transfer in cache-coherent multiprocessors. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

Received December 2018; revised February 2019; accepted February 2019