Algorithm-Directed Crash Consistence in Non-Volatile Memory for HPC

Shuo Yang
ys14@mails.tsinghua.edu.cn

Kai Wu†
kwu42@ucmerced.edu

Yifan Qiao
qian-yf15@mails.tsinghua.edu.cn

Dong Li†
dli35@ucmerced.edu

Jidong Zhai
zhaijidong@tsinghua.edu.cn

Tsinghua University

University of California, Merced†

***Abstract*—Fault tolerance is one of the major design goals for HPC. The emergence of non-volatile memories (NVM) provides a solution to build fault tolerant HPC. Data in NVM-based main memory are not lost when the system crashes because of the non-volatility nature of NVM. However, because of volatile caches, data must be logged and explicitly flushed from caches into NVM to ensure consistence and correctness before crashes, which can cause large runtime overhead.**

In this paper, we introduce an algorithm-based method to establish crash consistence in NVM for HPC applications. We slightly extend application data structures or sparsely flush cache blocks, which introduce ignorable runtime overhead. Such extension or cache flushing allows us to use algorithm knowledge to *reason* data consistence or correct inconsistent data when the application crashes. We demonstrate the effectiveness of our method for three algorithms, including an iterative solver, dense matrix multiplication, and Monte-Carlo simulation. Based on comprehensive performance evaluation on a variety of test environments, we demonstrate that our approach has very small runtime overhead (at most 8.2% and less than 3% in most cases), much smaller than that of traditional checkpoint, while having the same or less recomputation cost.

I. Introduction

Fault tolerance is one of the major design goals for HPC. Because of hardware and software faults and errors, HPC applications can crash or have incorrect computation results during the execution. The most common strategy to enable fault tolerant HPC is to periodically store a consistent and correct application state in persistent storage, such that there is always a resumable state throughout the application execution. Such application state is often characterized as the data values of critical data objects within the application. If the application crashes or an error is detected during the application execution, the application can go back to the last consistent and correct state, and restart. The application-level checkpoint/restart mechanism is an implementation of such strategy.

However, there is a problem with the strategy of periodical checkpoint. If the application state to checkpoint is large, the application has to suffer from a large data copy overhead. This fact is especially pronounced in HPC systems based on remote storage nodes for checkpoint. Although there is a large body of work to reduce the checkpoint overhead, such as hierarchical checkpoint to save checkpoint in local compute nodes [1], [2], [3], incremental checkpoint that only checkpoints modified data to reduce checkpoint size [4], [5], [6], [7], and disk-less checkpoint [8], [9], [10], [4], the checkpoint overhead remains one of the major scalability challenges for future extreme-scale HPC systems [11].

The emergence of non-volatile memories (NVM), such as phase change memory (PCM) and STT-RAM, provide an alternative solution to build fault tolerant HPC. Unlike regular DRAM, those memories are *persistent*, meaning that data are not lost when the system crashes because of the non-volatility nature of NVM. Furthermore, short access latency and large memory bandwidth of NVM makes the performance of NVM close to that of DRAM. In fact, with hardware simulation, the existing work has demonstrated that using NVM as the main memory to run sophisticated scientific applications may not have large performance loss [12], because of memory level parallelism and the overlap between computation and memory accesses. Hence, using NVM as the main memory to build fault tolerant HPC is promising.

However, leveraging the non-volatility of NVM to establish a consistent and correct state, which is called *crash consistence*, throughout the application execution in NVM is challenging. Because of volatile hardware caches widely deployed in HPC systems, there is no guarantee that the application state in NVM is correct and usable by the recovery process to restart applications. Ideally, if the application state in NVM is the same as a one established by the checkpoint mechanism, then the existing restart mechanism can be seamlessly integrated into the NVM-based HPC.

To maintain a consistent and correct state in NVM throughout the application execution, the most common software-based approaches are redo-log (storing new data updates) or undo-log (storing old data values) [13], [14], and enable a transaction scheme for relatively small workloads (e.g., hash table searching, B-tree searching, and random swap). Those approaches are often based on a programming model with the support of persistent semantics [13], [14], [15]. Those approaches, unfortunately, can impose large overhead, because they have to log memory update intensively and maintain the corresponding metadata. Such overhead is especially pronounced when the data objects are frequently updated by

the application. In fact, our preliminary work with CG and dense matrix multiplication based on a undo-log [15] has 4.3× and 5.5× performance loss, respectively. While such large overhead is tolerable in specific domains (e.g., database) with data persistence prioritized over performance, this overhead is not acceptable in HPC. To leverage NVM as persistent memory and build a consistent and correct state, we must introduce a lightweight mechanism with minimum runtime overhead.

In this paper, we introduce a new method to establish a consistent and correct state for critical data objects of HPC applications in NVM. The goal is to replace checkpoint for HPC fault tolerance based on the non-volatility of NVM, while introducing ignorable runtime overhead.

Our method is based on the following observation. Given a relatively small cache size (comparing with the main memory), most of data in an HPC application are not in caches, and should be in consistent state in NVM, because HPC applications are characterized with a large memory footprint. However, how to detect which data in NVM is consistent and can be reused for recomputation is challenging. The existing logging-based approaches explicitly establish data consistence and correctness with logs, but at the cost of data copy. If we can reason the consistent state of data in NVM, then we do not need logs, and remove expensive data copy.

Based on the above observation, we propose a novel method to analyze data consistence and correctness in NVM when the application crashes. In particular, instead of frequently tracking and maintaining data consistence and correctness in NVM at runtime, we slightly extend the application data objects or selectively flush cache blocks, which introduces ignorable runtime overhead. Such application extension or cache flushing allow us to use algorithm knowledge to reason data consistence and correctness when the application crashes.

In essence, we leverage invariant conditions inherent in algorithms, and decide if the invariant conditions still hold when the crash happens. We study using numerical algorithm knowledge to detect consistence and correctness for restart from three perspectives. First, we leverage the orthogonality relationship between data objects to detect consistence and correctness. We use the conjugate algorithm from sparse linear algebra as an example to study the feasibility of this method.

Second, we leverage the invariant conditions established by the algorithm-based fault tolerance method (ABFT) to detect data consistence and correctness. It has been shown that ABFT introduces ignorable runtime overhead by slightly embedding extra information (e.g., checksum) into data objects. Using the extra information, we can determine data consistence and correctness when the application crashes, and even correct inconsistent data. We use an algorithm-based matrix multiplication from dense linear algebra as an example to study the feasibility of this method.

Third, we study the Monte-Carlo (MC) method. MC is known for its statistics nature and error tolerance. In a sense, the inconsistency data is an "error". Hence MC can restart from the crash without knowing the consistent state of the critical data objects in NVM. However, contrary to the common intuition, we find that some critical intermediate results in MC could be lost and have big impact on computation result correctness. We must ensure the consistence and correctness of those critical intermediate results. Based on the above algorithm knowledge, we only flush the data of the critical intermediate results out of caches. This brings ignorable overhead while ensuring the computation correctness when restarting MC.

Algorithm knowledge has been used to address the problems of fault tolerance [16], [17], [18], [19], [20], performance optimization [21], [22], [23], and energy efficiency [24], [25]. In this paper, we extend the usage of numerical algorithm knowledge into a new territory: by using algorithm-inherent invariant relationship between data objects or algorithm-inherent statistics, we determine consistence and correctness of data objects without expensive data copy or cache flushing.

Our main contributions are summarized as follows.

- We introduce an algorithm-directed approach to address crash consistence in NVM for HPC. This approach has very small runtime overhead (at most 8.2% and less than 3% in most cases), much smaller than that of traditional checkpoint, while providing the same or less recomputation cost.
- We reveal that with the algorithm-directed approach, the recomputation cost varies because of caching effects. With a sufficiently large input problem, most of data objects in HPC applications can be consistent and correct in NVM. Hence, the recomputation cost can be small.
- We demonstrate that based on the algorithm-directed approach, leveraging the non-volatility of NVM to enhance or even remove the traditional checkpoint is feasible for future HPC.

II. Background

Recent progresses on NVM techniques have implemented NVM with different performance characteristics. Some reports have shown that certain NVM techniques (such as ReRAM and STTRAM) can achieve very similar latency and bandwidth as DRAM [26], and some NVM techniques (such as PCM) may have less than an order of magnitude reduction in performance (up to 4× higher latency and 8× lower bandwidth [26], [27]).

Leveraging byte addressability, better scalability, and excellent performance of NVM, using NVM as the main memory is promising. In this NVM usage model, NVM may be built as NVDIMM modules, and physically attached to high-speed memory bus and managed by a memory controller [28]. Furthermore, for those NVM techniques with a performance gap between NVM and DRAM (such as PCM), the likely deployment of NVM is to build a heterogeneous NVM and DRAM system. In such system, most of the memory is NVM to exploit their low cost and scalability benefits, and a small fraction of the total memory is DRAM. A large body of work has explored NVM as the main memory, including those software-based solutions [27], [29], [30], [31] and hardware-based solutions [32], [33], [34], [35]. In this paper, we assume

that NVM is used as the main memory, and explore a software-based solution to enable resilient HPC on NVM.

To build a consistent and correct state for critical data objects in NVM, the data objects in NVM must be updated with the most recent data in caches. This can be achieved by using cache flushing instructions to flush cache blocks of data objects out of caches. Because there is no mechanism to track which cache block is dirty and whether a specific cache block is in caches, we have to flush all cache blocks of the data objects, as if those cache blocks are in caches. Flushing clean cache blocks in caches and flushing cache blocks not in caches have performance cost at the same order as flushing dirty cache blocks. Hence, depending on the size of the data objects, flushing cache blocks of the data objects can be expensive.

We use `CLFLUSH`, the most common cache flush instruction, in this paper. Other cache flush instructions include `WBINVD`, `CLFLUSH_OPT`, and `CLWB`. However, `WBINVD` is a privileged instruction used by operating system (OS); `CLFLUSH_OPT` is only available in the most recent Intel processor (skylake) and `CLWB` is not available in the commercial hardware yet. Hence we do not use them in the paper. But considering them should further improves performance of our proposed approach.

III. Algorithm-Directed Crash Consistence

In this section, we explain our approach in details for three representative and popular algorithms. We first explain our performance evaluation methodology, commonly used to study the three algorithms.

A. Evaluation Methodology

To study data consistence between caches and NVM-based main memory when the application crash happens, we develop a "crash emulator" that allows us to examine the values of remaining data in caches and main memory. To study application performance in NVM, we use an NVM performance emulator. We explain the crash emulator and test environment as follows.

Crash emulator. We develop a PIN [36] based crash emulator. In essence, the crash emulator intercepts memory read and write instructions from the application, and emulates a configurable LRU cache. But different from the traditional PIN-based cache emulator, our crash emulator records the most recent data values in caches and main memory. The crash emulator also allows the user to specify when to trigger application crash. When a user-specified crash point is triggered, the crash emulator will output the values of data in caches and main memory.

The user can specify the application crash point in two ways. In the first way, the user can ask the crash emulator to output the data values after a specific statement is executed. This is achieved by inserting an API (particularly *crash_sim_output()*) right after the statement in the application. In the second way, the user can ask the crash emulator to output the data values after a specific number of instructions have been executed. To implement the second way, the crash emulator first profiles the application to collect total number of instructions, and then reports it to the user. The user chooses an instruction number to trigger application crash, and then re-runs the crash emulator. The data values will be output by the crash emulator when the user-specified number of instructions is executed.

Test environment. To measure runtime performance, we use a system with two Xeon E5606 (2.13GHz) and 28GB memories. To emulate NVM performance, we use Quartz emulator [37], a lightweight DRAM-based emulator that allows us to change DRAM bandwidth and latency. Quartz has low overhead and good accuracy (with emulation errors 0.2% - 9%) [37]. More importantly, it allows us to emulate HPC workloads with large data sets within reasonable time.

Since NVM techniques may have inferior performance than DRAM (e.g., 4× higher latency and 8× lower bandwidth [27]), we configure NVM bandwidth as 1/8 of DRAM bandwidth with Quartz. We change memory bandwidth, because our evaluation tests include cache flushing and memory copying, which are sensitive to memory bandwidth. Since NVM has inferior performance with the lower bandwidth, we introduce a DRAM cache to bridge performance gap between NVM and DRAM. Such heterogeneous NVM/DRAM-based main memory is common [27], [29], [38], [33], [39]. The DRAM cache size is 32MB, the same as a recent algorithm-based work for NVM [40]. With this heterogeneous NVM/DRAM-based main memory, we must decide data placement between NVM and DRAM. We decide data placement in this memory system based on a recent work [27].

Besides the above NVM emulation based on Quartz, we also study an NVM configuration with the same bandwidth and latency as DRAM. With this configuration, NVM is the same as DRAM. Hence we use an *NVM-only system* without DRAM cache, and do not need Quartz. Such NVM configuration is based on an optimistic assumption on NVM performance, and eliminates any performance impact of data placement in a heterogeneous NVM/DRAM on our study.

We thoroughly evaluate the performance of our algorithm-based approach with seven test cases. (1) Native execution: the execution without any checkpoint or algorithm-based approach; (2) Checkpoint based on a local hard drive; (3) Checkpoint based on the NVM-only system; (4) Checkpoint based on the heterogeneous NVM/DRAM system; (5) Using the Intel PMEM library [15] on the NVM-only system; (6) Algorithm-based approach on the NVM-only system; (7) Algorithm-based approach on the heterogeneous NVM/DRAM system.

Among the seven test cases, checkpoint is the most common method to establish a consistent and correct state on non-volatile storage in HPC; The Intel PMEM library represents the state of the art approach to establish a consistent and correct state in NVM. We compare our algorithm-based approach with checkpoint and PMEM to study the performance benefit of the algorithm-based approach.

With the memory-based checkpoint (i.e., checkpoint based on the NVM-only system or the heterogeneous NVM/DRAM system), checkpoint is equivalent to perform data copying plus cache flushing to ensure data consistence between NVM and

1 $r \leftarrow b - A \cdot x, z \leftarrow 0, p \leftarrow 0, q \leftarrow 0, \rho \leftarrow r^{\mathrm{T}} \cdot r$;
2 for $i \leftarrow 1$ to n
3 $q \leftarrow Ap$
4 $\alpha \leftarrow \rho/(p^{\mathrm{T}} \cdot q)$
5 $z \leftarrow z + \alpha p$
6 $\rho_0 \leftarrow \rho$
7 $r \leftarrow r - \alpha p$
8 $\rho \leftarrow r^{\mathrm{T}} \cdot r$
9 $\beta \leftarrow \rho/\rho_0$
10 $p \leftarrow p + \beta p$
11 Check $r = b - A \cdot z$.
12 end for

Fig. 1. Pseudo-code for CG. Capital letters such as A represent matrices; lowercase letters such as x, y, z represent vectors; Greek letters α, ρ represent scalar numbers.

caches. The cache flushing in the NVM-only system includes using `CLFLUSH` instruction to flush CPU caches; The cache flushing in the heterogeneous NVM/DRAM system includes flushing both CPU caches (using `CLFLUSH`) and the DRAM cache (using memory copy).

We explain our algorithm-directed crash consistence in details as follows.

B. Algorithm-Directed Crash Consistence for Iterative Method

Conjugate Gradient (CG) is one of the most commonly used iterative methods to solve the sparse linear system $Ax = b$, where the coefficient matrix A is symmetric positive definite. Figure 1 lists the algorithm pseudocode. In CG, three vectors p, q, and z can be checkpointed for resuming other variables and restarting. In the rest of this section, we use notation p^i, r^i and z^i to specify p, r, and z in the iteration i before they are updated in Lines 10, 7, and 5 respectively; q^i specify q in the iteration i.

In CG, there are implicit relationships between multiple data objects, shown in Equations 1 and 2. In particular, Equation 1 shows that at each iteration i, the vectors $p^{(i+1)}$ and $q^{(i)}$ satisfy an orthogonality relationship. Equation 2 shows that at each iteration i, the vectors $r^{(i+1)}$, $z^{(i+1)}$, b, and the matrix A satisfy an equality relationship.

$$p^{(i+1)^T} \cdot q^{(i)} = 0 \tag{1}$$

$$r^{(i+1)} = b - A \cdot z^{(i+1)} \tag{2}$$

Algorithm extension. Instead of using the checkpoint method (in which at least three arrays p, q, and z should be explicitly saved) to achieve the crash consistence, we rely on the existing hardware-based caching mechanism to evict data out of caches and opportunistically build the crash consistence. We extend CG and leverage the above implicit relationships between the data objects to reason the crash consistency of p, q, and z in NVM. This method removes runtime checkpoint and frequent cache flushing, hence improving performance.

In particular, if a crash happens at an iteration i, we examine the data values of $p^{(i+1)}$, $q^{(i)}$, $z^{(i+1)}$, and $r^{(i+1)}$ in NVM, and decide if the above implicit relationships are held. If not, then p, q, z, and r are not consistent and valid, and we cannot restart from the iteration i. We then check $p^{(i)}$, $q^{(i-1)}$, $z^{(i)}$ and $r^{(i)}$ and examine the implicit relationship for the iteration $i-1$. We continue the above process, until we find an iteration j ($j < i$) where the four data objects satisfy the above implicit relationship. This indicates that $p^{(j+1)}$, $q^{(j)}$, $z^{(j+1)}$, and $r^{(j+1)}$ are consistent and valid. We can restart from the iteration j.

To implement the above idea, we need to extend the original implementation shown in Figure 1. In the figure, p, q, r and z are one-dimensional arrays overwritten in each iteration. We add another dimension into the four arrays, such that each array has the data values of each iteration. We also flush the cache line containing the iteration number i at the beginning of each iteration. This makes the iteration number consistent between caches and NVM, which is helpful for the examination of the data values in NVM after the crash. Note that we only flush one single cache line at every iteration. This brings ignorable performance overhead. Figure 2 shows our extension to the original implementation.

1 $r \leftarrow b - A \cdot x, z \leftarrow 0, p \leftarrow 0, q \leftarrow 0, \rho \leftarrow r^{\mathrm{T}} \cdot r$;
2 for $i \leftarrow 1$ to n
3 flush the cache line containing i
4 $q[i+1] \leftarrow Ap[i]$
5 $\alpha \leftarrow \rho/(p^{\mathrm{T}} \cdot q)$
6 $z[i+1] \leftarrow z[i] + \alpha p$
7 $\rho_0 \leftarrow \rho$
8 $r[i+1] \leftarrow r[i] - \alpha p$
9 $\rho \leftarrow r^{\mathrm{T}} \cdot r$
10 $\beta \leftarrow \rho/\rho_0$
11 $p[i+1] \leftarrow p[i] + \beta p$
12 Check $r = b - A \cdot z$.
13 end for

Fig. 2. Extending CG to enable algorithm-directed crash consistence. Our extension to CG is highlighted with red color.

Performance characterization. The above algorithm-based approach does not use checkpoint or frequent cache flushing to explicitly build consistent and valid data state for those critical data objects in NVM. This greatly reduces runtime overhead, as shown in our performance evaluation.

The success of this approach heavily relies on the memory access patterns in CG. The two-dimensional arrays in the new CG (Figure 2) have a "streaming-like" memory access pattern: the data values generated in any iteration are used in at most two iterations. If the working set size of CG is much larger than the last level cache size, the data values of p, q, r, and z from the previous iterations before the crash happens have a very good chance to be evicted out of caches and consistent in NVM. In fact, CG is common to be applied on large sparse linear systems that have large data sizes. For those systems, our approach can effectively leverage the existing hardware caching management to evict data out of caches and build consistent and valid data states. In theory, given sufficiently large linear systems, the data values of the four arrays for a specific iteration i can be evicted out of caches in the iteration $i+1$. If the cache happens at the iteration $i+1$, we can restart from the iteration i, which limits recomputation cost to only one iteration. This literally achieves the same recomputation cost as checkpointing at every iteration.

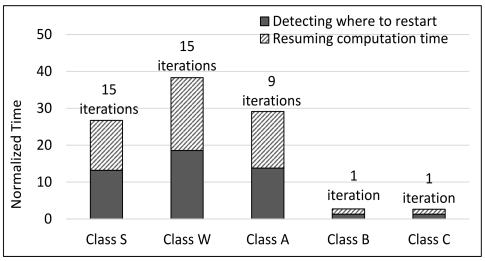


Fig. 3. Recomputation cost (execution time) for CG with our algorithm-based approach. The recomputation cost is normalized by the average execution time of individual iterations of CG. The input problem size is ranked in the increasing order in the x axis.

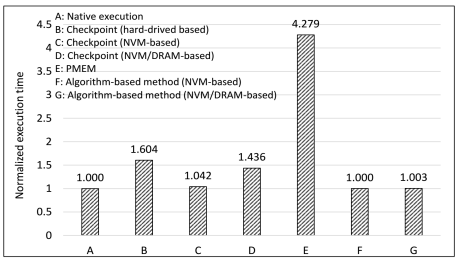


Fig. 4. Runtime performance (execution time) with various mechanisms and our algorithm-based approach to enable crash consistence. Performance is normalized by the performance of native execution without any checkpoint or algorithm extension.

However, if the linear system to solve by CG is small, the data values of the four arrays from multiple iterations are in caches and get lost when the crash happens. Depending on how many iterations of the data are lost, our approach could result in larger recomputation cost than the traditional checkpoint. In the worst case, our approach has to restart from the very beginning of CG. However, for those small systems with data sizes smaller than the last level cache size, the recomputation cost is small, and hence may not be a problem.

Performance evaluation. We evaluate the performance of our approach from two perspectives, recomputation cost and runtime overhead. Ideally, we want to minimize recomputation cost after crashes, and minimize runtime overhead.

To measure recomputation cost, we use the crash emulator to trigger a crash at a specific program execution point, particularly Line 10 (Figure 2) in the 15th iteration of the main loop in NPB CG (one benchmark in NAS parallel benchmark suite [41]). Figure 3 shows the performance on the heterogeneous NVM/DRAM system with different input problems of CG. The recomputation time in the figure is broken down, and it includes the time to detect from which iteration CG is resumable (labeled as "Detecting where to restart") and the time to resume from the resumable iteration to the crashed iteration (labeled as "Resuming computation time" in the figure). The recomputation time is normalized by the average execution time of individual iterations of the main loop in CG. The number of iterations on the top of each column is the number of iteration we lose because of the crash.

Figure 3 reveals that the recomputation cost becomes smaller when we use a larger input problem size. When the input problem size is small (Classes S and W), the recompaution time is relatively large. We lose all of the iterations (15 iterations) when the crash happens. However, when the input problem size is large (Classes B and C), we lose only 1 iteration and the recomputation time is very small. This result is aligned with our performance characterization: in particular, a larger input problem tends to lose smaller computation when a crash happens.

We further compare runtime overhead between traditional checkpoint, Intel PMEM library [15], and our approach. With Class C as the input problem, recomputation with our approach is limited to one iteration, already shown in Figure 3. Hence, we make checkpoint at the end of each iteration of the main loop in CG. This frequent checkpoint enables a fair performance comparison, because checkpoint at the end of each iteration results in the same recomputation cost as our algorithm-based approach. For the PMEM library, we use its transaction mechanism and enable transactional updates on the three arrays (i.e., p, r, and z). Each iteration of the main loop of CG is a transaction, which makes the recomputation cost with PMEM also limited to one iteration. The transaction mechanism in the PMEM library is based on undo log. Figure 4 shows the results.

Figure 4 reveals that the traditional checkpoint based on hard drive has very large overhead (60.4%), comparing with the native execution), even if the hard drive is local. Assuming that NVM has equivalent performance as DRAM, NVM-based checkpoint leads to ignorable overhead (4.2%). However, if NVM performance is not as good as DRAM, frequent checkpoint causes large overhead (43.6%). Our further study reveals that 51.9% of the overhead comes from data copying and 48.1% comes from cache flushing (including DRAM cache flushing). The PMEM library also causes large overhead (329%), because of frequent and expensive data logging. Our approach, on the other hand, has ignorable runtime overhead (less than 3%, no matter whether on NVM-only or NVM/-DRAM system) because our approach does not come with any extra data copying and only flushes one single cache line per iteration.

Conclusions. Our evaluation results demonstrate that CG with our algorithm extension achieves superior runtime performance with close to zero runtime overhead. Furthermore, when the input problem size is large enough, the recomputation cost is limited to only one iteration. Our approach performs better than the traditional checkpoint or PMEM library for NVM.

C. Algorithm-Directed Crash Consistence for Matrix Multiplication

Matrix multiplication ($C = A \times B$) is a very common numerical computation. Matrix multiplication (MM) has been extended to handle fail-continue errors which have the failed process to continue working when errors occur. To detect and correct fail-continue errors, MM is extended to add checksum within matrices A, B, and C. Such method is based on numerical algorithm (called *algorithm-based fault tolerance or ABFT*) has shown success within 5% performance loss while detecting and correcting certain number of errors in the matrices.

1 **for** $s = 1$; $s \leq n + 1$; $s \leftarrow s + k$
2 Verify the checksum relationship of C^f
3 $C^f \leftarrow C^f + A_c(1 : n + 1, s : s + k - 1) \times B_r(s : s + k - 1, 1 : n + 1)$
4 **end for**

Fig. 5. A practical implementation of ABFT for matrix multiplication.

How MM is extended to add checksums in the existing method [19] is as follows. The input matrices A and B are encoded into a new form with checksums, shown in Equations 3 and 4. Equation 3 defines a column checksum matrix of A (assuming the size of A is $m \times k$), denoted by A_c, where the vector v is a checksum vector. v is typical set with all elements as 1. With v, the last row of A_c ($a_{m+1,j}$, $1 \leq j \leq k$) is also shown in Equation 3. Equation 4 defines a row checksum matrix of matrix B (assuming the size of B is $k \times n$), where the vector w is a checksum vector. w is typically set with all elements as 1. With w, the last column of B_r ($b_{j,n+1}$, $1 \leq j \leq k$) is also shown in Equation 4.

$$A_c = \begin{pmatrix} A \\ v_c^{\mathrm{T}} A \end{pmatrix}, \quad a_{m+1,j} = \sum_{i=1}^{m} a_{i,j} \tag{3}$$

$$B_r = \begin{pmatrix} B & Bv_r \end{pmatrix}, \quad b_{j,n+1} = \sum_{j=1}^{n} b_{i,j} \tag{4}$$

With the encoded A and B, instead of computing $C = A \times B$, we compute their checksum versions, shown in Equation 5. C^f is the result matrix with checksums. In particular, in C^f, the summations of each row of C are stored in the extra column of C^f, and the summation of each column of C are stored in the extra row of C^f, shown in Equation 6. If one element of C is corrupted, using the checksum relationship shown in Equation 6, we can detect and correct errors.

$$C^f = A_c \times B_r = \begin{pmatrix} AB & ABv_r \\ v_c^{\mathrm{T}} AB & v_c^{\mathrm{T}} ABv_r \end{pmatrix} \tag{5}$$

$$c_{m+1,j} = \sum_{i=1}^{m} c_{i,j}, \quad c_{i,n+1} = \sum_{j=1}^{n} c_{i,j} \tag{6}$$

The above ABFT is commonly implemented based on Figure 5. This implementation detects errors at every iteration of the loop (Line 2 in Figure 5) and makes better use of cache system based on a rank k update (Line 1 in Figure 5) [1]. Within each iteration of the loop, this implementation does a submatrix multiplication and accumulates the result into C^f. Our following discussion is based on this implementation.

Algorithm extension. We leverage the checksum information to detect crash consistence and correct inconsistent data in the output matrix C in NVM. To do so, a naive idea is to flush checksums at the end of each iteration of the loop in Figure 5. Then, when a crash happens, we check checksums to detect the validness of matrix rows and columns in C. However, the above idea does not work for the following two reasons.

[1]For brevity, we assume A and B are n by n square matrix, and $(n + 1)$ is divisible by k.

1 //submatrix multiplication $C_s^{temp}(.) = A_c(.) \times B_r(.)$
2 **for** $s = 1$; $s \leq (n + 1)/k$; $s \leftarrow s + 1$
3 $C_s^{temp} \leftarrow A_c(1 : n + 1,\ (s - 1) \times k + 1 : s \times k)\ \times$
4 $B_r((s - 1) \times k + 1 : s \times k,\ 1 : n + 1)$
5 flush row and column checksums in C_s^{temp}
6 **end for**
7
8 //submatrix addition
9 **for** $i = 1$; $i \leq n + 1$; $i \leftarrow i + k$
10 $C_{temp}(i : (i + k - 1),\ 1 : n + 1) \leftarrow$
11 $C_{temp}(i : (i + k - 1),\ 1 : n + 1)$ +
12 $\sum_{s=1}^{n/k} C_s^{temp}(i : (i + k - 1),\ 1 : n + 1)$
13 flush k rows of row checksums in C_{temp}
(particularly, $C_{temp}(i : (i + k - 1), n + 1)$)
14 **end for**
15 $C^f \leftarrow C^f + C_{temp}$

Fig. 6. A new version of ABFT for MM to facilitate the detection and correction of crash consistence in NVM.

First, we cannot detect consistence in the middle of an iteration based on the checksums. Checksums are only useful to detect consistence at the beginning of each iteration. In the middle of each iteration, the checksum row and column in C may be partially updated by computation, and Equation 6 is not held. Second, the matrix C^f is completely overwritten in each iteration, hence restart will be difficult. Even if we find inconsistence in C^f (i.e., some elements of C_f come from the iteration i while other elements come from the iteration j ($j \neq i$)), we do not have a consistent copy of C^f (i.e., all elements of C^f come from the same iteration) to restart. Checksums may be able to correct some of inconsistent elements based on Equation 6 at the beginning of the iteration, but such checksums can only correct limited number of inconsistent elements.

To address the above problems, we extend the above naive idea and introduce a new algorithm shown in Figure 6. The new algorithm decomposes the loop in the original ABFT into two loops. One loop performs the submatrix multiplication, and the other loop performs the addition of the submatrix multiplication results.

In the first loop, the submatrices are still the same as those in the original ABFT with embedded checksums. But different from the original ABFT, the first loop saves submatrix multiplication results into temporal matrices C_s^{temp} ($s = 1, ..., (n + 1)/k$) with row and column checksums, and flushes those checksums to make sure they are consistent (Line 5 in Figure 6). If a crash happens in the first loop, then using those checksums we can detect which temporal matrix (C_s^{temp}) is inconsistent in NVM. Any inconsistent and uncorrectable temporal matrix by checksums will be recomputed.

In the second loop, we perform the addition (matrix addition) of C_s^{temp} ($s = 1, ..., (n+1)/k$), and save the result with row checksum embedded in a temporal matrix C_{temp}. Note that we perform matrix addition rows by rows. Hence the row checksums, once established and consistent in NVM (Line 13 in Figure 6), will not be overwritten. If the crash happens in the second loop, then the row checksums in C_{temp} can decide which rows are not consistent and should be recalculated.

The new algorithm solves the two problems in the original

algorithm, because checksums in the result matrices (C_s^{temp} and C_{temp}), once established and consistent in NVM, are not overwritten, hence can be used reliably to detect inconsistence of matrix data at any moment. Furthermore, a set of temporal matrices enable easy recomputation and easy detection of inconsistence.

The downside of the above algorithm extension is that we increase memory consumption. But with the deployment of NVM with much higher capacity than DRAM, we expect this problem is alleviated. Also, the memory consumption relines on a choice of k. A Smaller k results in larger number of temporal matrices (more memory consumption) and smaller recomputation cost. We can manage memory consumption by selecting a good k and exploiting the tradeoff between memory consumption and recomputation cost.

The above algorithm extension also increases the working set size which could cause extra cache misses and lose performance. However, we do not see big performance loss (no bigger than 8.2%) in our evaluation. The reason is as follows. Given a large matrix size $n \times n$, the matrix multiplication based on the submatrix multiplication in the original code causes similar cache misses as the new algorithm, because both of the original code and the new algorithm fetch different submatrices for multiplication and save the results in either C^f or C_s. Those submatrices multiplications, which dominates the computation time, have "streaming-like" memory access patterns — we need to fetch submatrices one by one for multiplication. Such memory access patterns cause similar cache misses in the original code and the new algorithm. Furthermore, the regular memory access patterns in matrix multiplication allows prefetching to take effect and further alleviate the effects of cache misses.

Performance evaluation. We evaluate the performance of our approach from the perspective of recomputation cost and runtime performance.

Figure 7 shows the recomputation cost on the heterogeneous NVM/DRAM system. We use four different input matrix sizes. For each matrix size, we do two crash tests. One crash test triggers a crash at the end of the 4th iteration of the first loop in Figure 6. The second crash test triggers a crash at the end of the 4th iteration of the second loop in Figure 6. Hence, within Figure 7, we have two columns representing the recomputation cost for the two crash tests for each matrix size.

The recomputation time for the first crash test is normalized by the average execution time of one iteration (i.e., submatrix multiplication) of the first loop in Figure 6. The recomputation time for the second crash test is normalized by the average execution time of one iteration (i.e., submatrix addition) of the second loop in Figure 6. Such normalization can quantify how many submatrix multiplications or additions are lost when a crash happens. Similar to the recomputation result for CG, Figure 7 breaks down the recomputation cost into "detecting where to restart" and "resuming computation time".

Figure 7 shows that using different matrix sizes as input we lose different numbers of submatrix multiplication in the first crash test. With $n = 2000$ as input, we lose about

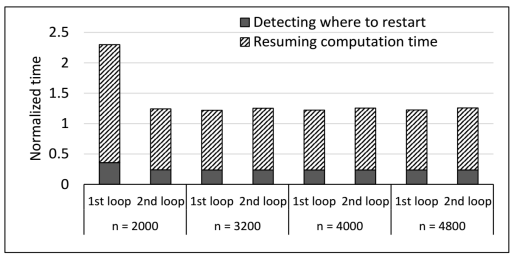


Fig. 7. Recomputation cost (execution time) for ABFT-based matrix multiplication for two crash tests happened in the first and second loops. The x axis is the matrix size. Rank $k = 400$ in all tests. The recomputation cost is normalized by the average execution time of one iteration in the first loop or the second loop in Figure 6.

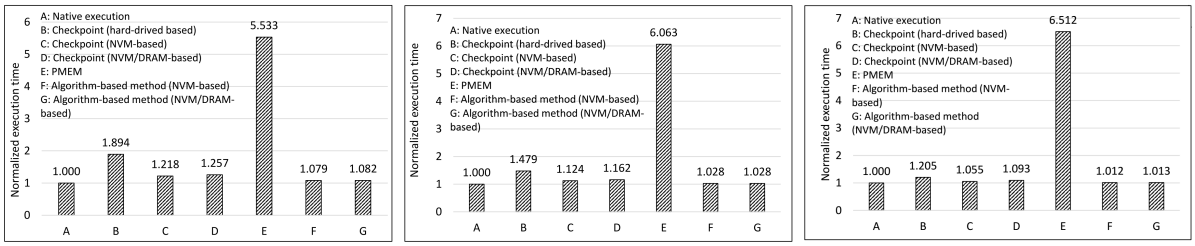
two submatrix multiplications, but with larger inputs, we lose only one submatrix multiplication. Using a larger input, our algorithm-based approach limits the recomputation to at most one submatrix multiplication because the submatrices are large and eliminate computation results from other iterations out of caches. For the second crash test, we always lose one submatrix addition, even if we use a relatively small matrix (i.e., $n = 2000$). The reason is that each iteration of the second loop has larger memory footprint than that of the first loop, which eliminates the results of submatrix addition from other iterations out of caches.

Note that during our crash tests, our algorithm-based approach cannot correct those inconsistent data in C_s^{temp} and C_{temp}, because there are too much inconsistent data in the same row or column, which are not correctable by the checksums. Hence we claim in the above discussion that the submatrix multiplication or addition is lost. However, for some cases, it is possible that our algorithm-based approach can directly correct those inconsistence data based on checksums. In those cases, the submatrix multiplication or addition is not lost, and the recomputation cost is even smaller.

We further study runtime performance. Similar to CG, we compare runtime performance with traditional checkpoint, Intel PMEM library [15], and our approach. We use the matrix size as $n = 8000$. Given such large matrix size, the recomputation cost with our approach is limited to a submatrix multiplication or a submatrix addition. Hence, we perform the traditional checkpoint at the end of each submatrix multiplication, such that the recomputation cost for the traditional checkpoint is a submatrix multiplication. For the PMEM library, each submatrix multiplication (the first loop of Figure 6) is a transaction and we enable transaction update on the submatrix multiplication result, which makes the recomputation cost also limited to a submatrix multiplication. Figure 8 shows the runtime performance.

The figure reveals that our algorithm-based approach has the smallest runtime overhead among all cases (no bigger than 8.2% in all cases, depending on the rank size). Also, a larger rank size results in a smaller runtime overhead, because the algorithm does not need to frequently flush checksum cache blocks. With a large rank size (e.g., $rank = 1000$), the runtime overhead is only 1.3%.

The runtime overhead of our algorithm-based approach is much smaller than the NVM-based checkpoint. For example,



(a) rank = 200 (b) rank = 400 (c) rank = 1000

Fig. 8. Runtime performance (execution time) with various checkpoint mechanisms and our algorithm extension for ABFT-based matrix multiplication. The matrix size is $n = 8000$. Performance is normalized to the native execution with neither checkpoint nor our algorithm extension.

when $rank = 200$, NVM-based checkpoint has at least 21.8% overhead, while our approach has only 8.2%. This is due to the fact that our approach selectively flushes cache blocks of the checksums, instead of copying all data blocks of critical data objects.

Conclusions. Treating inconsistent data as data corruption and using algorithm-based fault tolerance to detect and correct data inconsistence in NVM save expensive runtime overhead to establish crash consistence. Selectively flushing cache blocks to maintain crash consistence of checksums is the key. Given a large matrix size, our approach can effectively limit the recomputation cost to one submatrix multiplication or addition.

D. Algorithm-Directed Crash Consistence for Monte Carlo Transport Simulation

Monte Carlo method (MC) has been applied to a broad range of scientific simulations, such as nuclear reactor physics and medical dosimetry. In essence, MC employs repeated random sampling to obtain numerical results and solve problems that are deterministic in principle. Given MC simplicity, MC provides significant advantages compared to deterministic methods. Leveraging MC's random nature, we explore how to use MC algorithm knowledge to build crash consistence in NVM without losing scientific simulation accuracy.

We focus on a specific MC benchmark, XSBench. This benchmark models the calculation of macroscopic neutron "cross sections" [42] within a nuclear reactor, which is the most computationally intensive part of a typical MC transport algorithm [43]. Listing 9 shows the major computation of XSBench.

XSBench has two large, read-only data arrays, which are a nuclide grid and an energy grid. The two arrays account for most of the memory footprint of XSBench. XSBench has a main computation loop, and each iteration of the loop performs a lookup of some data (particularly "cross section" data) from the nuclide grid with the assist of searching the energy grid. The lookup result of each iteration accumulates to an array (particularly, macro_xs_vector) with five elements (Line 7 in Figure 9). Each element of the array macro_xs_vector is the value of a macroscopic cross section. Those five values correspond to five different particle interaction types in the nuclear reactor. In each iteration, the lookup happens based on two randomly chosen inputs (particularly, neutron energy and material, shown in Line 2 in Figure 9).

XSBench is only a benchmark for performance study, hence its result (particularly macro_xs_vector) does not have sufficient physical meaning. From one run to another, the result can be different due to the random nature of the benchmark. It is difficult to know if the benchmark result remains correct for our crash consistence evaluation. Based on the domain knowledge, we slightly extend the benchmark such that the benchmark result has physical meaning. In particular, at the end of each iteration, we apply a cumulative distribution function (CDF) to the five elements of macro_xs_vector, and then normalize the CDF result by the largest element. Then we generate a uniformly distributed random number x ($0 < x < 1$). This random number represents a computation result in a full-featured simulation of the nuclear reactor. Based on the random number, we find which interaction type (i.e., which element of macro_xs_vector) should be chosen based on the normalized CDF result.

For example, suppose macro_xs_vector = $\{0.9, 0.1, 0.3, 0.6, 0.05\}$ in one iteration. We create a CDF of this macro_xs_vector, which is $\{0.9, 1.0, 1.3, 1.9, 1.95\}$. Then we normalize the CDF result by the largest element of macro_xs_vecotr (i.e., 1.95). The normalization result is $\{0.462, 0.513, 0.667, 0.974, 1.0\}$. Then, we choose a random number, for example, 0.65. According to the normalization result, 0.65 falls between the second (0.513) and third elements (0.667), which means the second interaction type is chosen.

We perform the above process for each iteration of the XSBench loop, and introduce five counters to count how many times each interaction type is chosen for all iterations. Given the sufficient number of lookups (i.e., iterations of the main computation loop), the number of times an interaction type is chosen is roughly the same for all interaction types. This method gives us a deterministic and meaningful way to quantify the validness of the benchmark result.

Algorithm extension. Because of the random nature of XSBench, we expect that directly restarting from remaining data in NVM after a crash does not result in incorrect results, assuming that macro_xs_vector and the two arrays (nuclide grid and energy grid) in NVM are accessible after the crash. In particular, at every iteration, we flush the cache block containing the loop index variable i, such that we can know which iteration the crash happens. After the crash happens, we restart from the beginning of the iteration i. Except flushing the single cache block, we do not use any mechanism to establish data consistence and validness in NVM during the execution.

```
1  for (i = 0; i < total number of lookups; i++)
2    Generates two randomly sampled inputs
     (neutron energy and material);
3    Binary search on the energy grid;
4    for each nuclide in the input material
5      Look up two bounding nuclide grid points
       from the nuclide grid;
6      Interpolate the two points to give microscopic
       cross sections;
7      Microscopic cross sections accumulate into
       macroscopic cross section (i.e., macro_xs_vector);
8    end for
9  end for
```

Fig. 9. Pseudo code for XSBench.

Using the above approach, we may lose the binary search result (Line 3 in Figure 9), the lookup result in the nuclide grid (Line 5 in Figure 9), the interpolate result (Line 6 in Figure 9), and the accumulation result (Line 7 in Figure 9). However, we expect the impact of those results losses is limited, because the binary search and the lookup in the nuclide grid need to load the energy grid and the nuclide grid into caches. Those two grids are two large arrays, and Loading them can evict the results of most of the previous iterations out of caches, and implicitly enable data consistence. We may only lose the results of a few iterations, and those results will not be accumulated into macro_xs_vector (Line 7 in Figure 9). However, losing a few iterations of the results may not impact the accuracy of counting the number of times each interaction type is chosen, because XSBench uses a sampling-based approach and takes a number of samples (i.e., the number of lookups shown in Line 1). As the number of samples is large, losing a few samples is not expected to impact the counting accuracy.

To verify the above basic idea, we run XSBench with an input problem of 34 fuel nuclides in a Hoogenboom-Martin reactor model. With such input problem, the energy grid and nuclide grid take about 246MB memory. There are 1.5×10^7 lookups in the main computation loop. We use our crash simulator to run the benchmark and trigger a crash when the benchmark is in the 1.5×10^6th lookup (10% of all lookups).

Figure 10 shows how many times each interaction type is counted for two tests. In one test, we do not have crash (labeled as "No crash"); in the other test, we have the crash but immediately restart based on the above basic idea (labeled as "Crash and restart based on the basic idea"). The numbers of times counted for the five interaction types are normalized by the total number of lookups and shown as percentage in the y axis. These two tests use the same randomly sampled inputs (Line 2 in Figure 9) for each lookup, such that we enable a fair comparison of the XSBench results of the two tests.

From Figure 10, we notice that the five interaction types in the case of no crash have almost the same counting result. However, the case with crash and restart based on the basic idea has obviously different counting results for the five interaction types. For example, there is 8% difference in the counting result between the interaction types 1 and 2.

To investigate the reason why there is such result difference

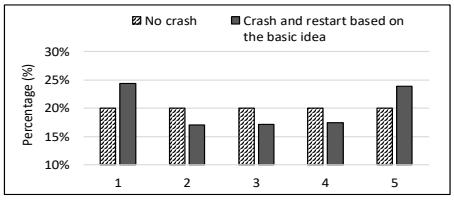


Fig. 10. Comparing XSBench results of two cases (one case without crash and the other case with crash and restart based on the basic idea).

```
1  for (i = 0; i < total number of lookups; i++)
2    Generates two randomly sampled inputs
     (neutron energy and material);
3    Binary search on the energy grid;
4    for each nuclide in the input material
5      Look up two bounding nuclide grid points
       from the nuclide grid;
6      Interpolate the two points to give microscopic
       cross sections;
7      Microscopic cross sections accumulate into
       macroscopic cross section (i.e., macro_xs_vector);
8      if (every 0.01% of total number of lookups)
9        flushing macro_xs_vector, the five counters and i;
10   end for
11 end for
```

Fig. 11. Selectively flush cache blocks for XSBench based on the algorithm knowledge. Our extension to XSBench is highlighted in red.

between the two cases, we examine the five counters in NVM at the crash trigger point, and compare the values of those counters in the two cases. We found that the values of the five counters in the two cases are very different. For the crash case, the counting results from a number of iterations before the crash trigger point are still in caches. They are not evicted out of caches as expected. For the crash case, the counting results are not consistent between caches and NVM.

Further investigation reveals that although the two grids are two large arrays, binary search on the energy grid (Line 3 in Figure 9) and lookup operation in the nuclide grid (Line 5 in Figure 9) do not necessarily access the whole grids, and hence the five counters and macro_xs_vector are not evicted out of caches in many iterations as expected. Also, the five counters and macro_xs_vector are frequently updated from one iteration to another. Such frequent updates keep those variables in caches and make them inconsistent between caches and NVM. Hence, when a crash happens, we lose the results of many iterations.

To address the above data inconsistence problem, we can flush the five counters and macro_xs_vector whenever there is any update happened to them at every iteration. However, such frequent cache flushing causes 16% performance loss. Hence, we flush caches lines every n iterations ($n = 0.01\%$ of total number of lookups), shown in Figure 11.

How often we should flush cache blocks is a challenging problem. For XSBench, if the sizes of the two grids are big and a large portion of the two grids are touched at each iteration, then the five counters and macro_xs_vector can be evicted out of caches automatically and frequently, and we do not have to frequently flush cache blocks. However, due to the randomness of sampled inputs (i.e., neutron energy and

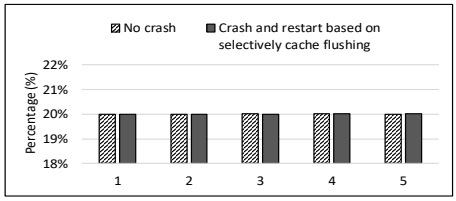


Fig. 12. Comparing XSBench results of two cases (one case without crash and the other case with crash and restart based on selectively cache line flushing).

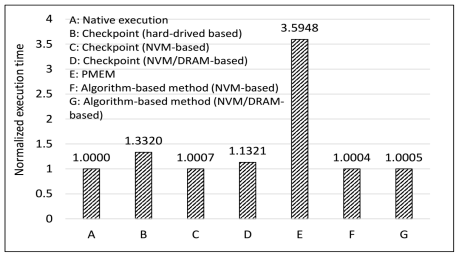


Fig. 13. Runtime performance (execution time) with various checkpoint mechanisms and our algorithm-based approach.

material at Line 2 in Figure 11), it is difficult to quantify the memory footprint size of each iteration. We empirically determine the frequency as every 0.01% of total number of lookups in our tests. Such frequency of cache flushing has ignorable performance overhead shown in Figure 13. Also, using such frequency for flushing cache blocks, we bound the result loss when a crash happens by 0.01% of total number of iterations, which is small.

Performance evaluation. We first compare the result correctness between our approach and no-crash case. We trigger the crash at the same point as the one in Figure 10, i.e., the 1.5×10^6th lookup (10% of all lookups). Figure 12 shows the result. With our approach, the number of times an interaction type is chosen is almost the same for all interaction types, which is the same result as the one with no crash.

We further compare the performance of the seven cases. For those cases with checkpoint, we checkpoint macro_xs_vector and five counters at every 0.01% of total number of iterations. This checkpoint frequency is the same as that in our algorithm. Figure 13 shows the results. Our selective cache block flushing (labeled as "algorithm-based approach") has ignorable overhead (at most 0.05%). NVM-based checkpoint based on the NVM-only system also has ignorable overhead. However, when we use the NVM/DRAM system, checkpoint overhead is as large as 13%, much larger than our runtime overhead.

Conclusions. To ensure result correctness, MC simulation must flush a few cache blocks to enable crash consistence. Different from CG which has large data objects frequently evicting critical data objects out of caches, XSBench with large data objects may have small memory footprint at each iteration and cannot evict critical data objects. This is due to the random nature of MC simulation.

IV. Related Work

Crash consistence in NVM. Leveraging persistent extensions from ISA (e.g., `CLFLUSH`), some work introduces certain program constructs to enable crash consistence in NVM. Mnemosyne [14], Intel NVM library [15], [44], NV-heaps [13], and REWIND [45] provide transaction systems optimized for NVM. NVL-C [46] introduces flexible directives and runtime checks that guard against failures that corrupt data consistence. SCMFS [47] provides a PM-optimized file system based on the persistent extensions from ISA. Atlas [48] uses those extensions for lock-based code. To use the existing efforts for HPC applications, we may have to make extensive changes to applications or operating systems. The application can suffer from large runtime overhead because of frequent runtime checking or data logging. Our evaluation with the Intel NVM library shows such large overhead.

Some work introduces persistent cache, such that stores become durable as they execute [49], [50], [51]. Those existing efforts eliminate the necessity of any cache flushing operation, by not caching NVM accesses, or by ensuring that a batter backup is available to flush the contents of caches to NVM upon power failure. However, those existing efforts need extensive hardware modification. It is not clear if integrating NVM into processors has any manufacturing challenges.

Some work divides program execution into epochs. In the epoch, stores may persistent concurrently by flushing cache lines or bypassing caches. Pelley et al. [52] introduce a couple of variation of epoch, and demonstrate potential performance improvement because of a relaxation of inter-thread persist dependencies. Joshi et al. [53] propose a buffered epoch persistency by defining efficient persist barriers. Delegated ordering [54] decouples cache management from the path persistent writes take to memory to allow concurrent writes within the same epoch to improve performance. Those existing efforts can be complementary to our work to improve the performance of cache flushing (especially for algorithm-directed crash consistence based on ABFT for matrix multiplication).

Algorithm-based program optimization. Leveraging algorithm knowledge is an effective approach to improve performance [21], [22], [23], application fault tolerance [16], [17], [18], [19], [20], and energy efficiency [24], [25]. Different from the existing efforts, this paper uses algorithm knowledge to achieve crash consistence in NVM. This is a fundamentally new approach to explore the usage of algorithm knowledge.

V. Conclusions

Leveraging the emerging NVM to establish crash consistence is a promising while challenging approach to enable resilient HPC. HPC has high requirement for performance. We must minimize runtime overhead when building crash consistence. This paper introduces a fundamentally new methodology to do so. Based on the algorithm knowledge and comprehensive performance evaluation, we show that we can significantly reduce runtime overhead while enabling detectable crash consistence for future NVM-based HPC.

Acknowledgment

We thank anonymous reviewers for their valuable feedback. This work is partially supported by U.S. National Science Foundation (CNS-1617967 and CCF-1553645). In China, this work is partially supported by National Key Research and Development Program 2016YFB0200100, NSFC Project 61472201, and Tsinghua University Initiative Scientific Research Program. Dong Li and Jidong Zhai are corresponding authors.

REFERENCES

[1] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2010.

[2] X. Dong, N. Muralimanohar, N. Jouppi, R. Kaufmann, and Y. Xie, "Leveraging 3D PCRAM Technologies to Reduce Checkpoint Overhead for Future Exascale Systems," in *International Conference on High Performance Computing Networking, Storage and Analysis (SC)*, 2009.

[3] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "FTI: High performance Fault Tolerance Interface for hybrid systems," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[4] J. S. Plank and K. Li, "Faster checkpointing with N+1 parity," in *International Symposium on Fault-Tolerant Computing*, 1994.

[5] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira, "Adaptive Incremental Checkpointing for Massively Parallel Systems," in *International Conference on Supercomputing (ICS)*, 2004.

[6] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott, "Hybrid Checkpointing for MPI Jobs in HPC Environments," in *International Conference on Parallel and Distributed Systems (ICPADS)*, 2010.

[7] G. Bronevetsky, D. Marques, K. Pingali, S. A. McKee, and R. Rugina, "Compiler-enhanced incremental checkpointing for OpenMP applications," in *International Symposium on Parallel and Distributed Processing (IPDPS)*, 2009.

[8] J. S. Plank, K. Li, and M. A. Puening, "Diskless Checkpointing," *IEEE Transactions on Parallel and Distributed System*, vol. 9, no. 10, 1998.

[9] C.-D. Lu, "Scalable Diskless Checkpointing for Large Parallel Systems," Ph.D. dissertation, 2005.

[10] X. Tang, J. Zhai, B. Yu, W. Chen, and W. Zheng, "Self-Checkpoint: An In-Memory Checkpoint Method Using Less Space and Its Practice on Fault-Tolerant HPL," in *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2017.

[11] F. Cappello and G. Bosilca, "DSN 2016 Tutorial: Resilience for Scientific Computing: From Theory to Practice," in *International Conference on Dependable Systems and Networks Workshops*, 2016.

[12] D. Li, J. Vetter, G. Marin, C. McCurdy, C. Cira, Z. Liu, and W. Yu, "Identifying Opportunities for Byte-Addressable Non-Volatile Memory in Extreme-Scale Scientific Applications," in *International Parallel and Distributed Processing Symposium*, 2012.

[13] J. Coburn, A. Caulfield, A. Akel, L. Grupp, R. Gupta, R. Jhala, and S. Swanson, "NV-heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[14] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight Persistent Memory," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[15] Intel NVM Library. http://pmem.io/nvml/libpmem/.

[16] Z. Chen, "Online-ABFT: An Online Algorithm based Fault Tolerance Scheme for Soft Error Detection in Iterative Methods," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2013.

[17] T. Davies and Z. Chen, "Correcting Soft Errors Online in LU Factorization," in *HPDC*, 2013.

[18] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra, "Algorithm-based Fault Tolerance for Dense Matrix Factorizations," in *PPoPP*, 2012.

[19] P. Wu, C. Ding, L. Chen, T. Davies, C. Karlsson, and Z. Chen, "Online Soft Error Correction in Matrix-Matrix Multiplication," *Journal of Computational Science*, vol. 4, no. 6, pp. 465–472, 2013.

[20] D. Li, Z. Chen, P. Wu, and J. S. Vetter, "Rethinking Algorithm-Based Fault Tolerance with a Cooperative Software-Hardware Approach," in *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

[21] M. Faverge, J. Herrmann, J. Langou, B. R. Lowery, Y. Robert, and J. Dongarra, "Designing LU-QR Hybrid Solvers for Performance and Stability," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2014.

[22] M. S. Lam, E. E. Rothberg, and M. E. Wolf, "The Cache Performance and Optimizations of Blocked Algorithms," in *ASPLOS*, 1991.

[23] S. Williams *et al.*, "Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2007.

[24] R. Dorrance *et al.*, "A Scalable Sparse Matrix-Vector Multiplication Kernel for Energy-Efficient Sparse-Blas on FPGAs," in *FPGA*, 2014.

[25] E. Garcia *et al.*, "Optimizing the LU Factorization for Energy Efficiency on a Many-Core Architecture," in *LCPC*, 2013.

[26] K. Suzuki and S. Swanson, "The Non-Volatile Memory Technology Database (NVMDB)," Department of Computer Science & Engineering, University of California, San Diego, Tech. Rep. CS2015-1011, 2015, http://nvmdb.ucsd.edu.

[27] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan, "Data Tiering in Heterogeneous Memory Systems," in *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys)*, 2016.

[28] R. Chen, Z. Shao, and T. Li, "Bridging the i/o performance gap for big data workloads: A new nvdimm-based approach," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[29] M. Giardino, K. Doshi, and B. Ferri, "Soft2LM: Application Guided Heterogeneous Memory Management," in *International Conference on Networking, Architecture, and Storage (NAS)*, 2016.

[30] W. Wei, D. Jiang, S. A. McKee, J. Xiong, and M. Chen, "Exploiting Program Semantics to Place Data in Hybrid Memory," in *International Conference on Parallel Architecture and Compilation (PACT)*, 2015.

[31] S. Gao, B. He, and J. Xu, "Real-Time In-Memory Checkpointing for Future Hybrid Memory Systems," in *International Conference on Supercomputing*, 2015.

[32] B. Wang, B. Wu, D. Li, X. Shen, W. Yu, Y. Jiao, and J. S. Vetter, "Exploring Hybrid Memory for GPU Energy Efficiency through Software-Hardware Co-Design," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013.

[33] H. Yoon, J. Meza, R. Ausavarungnirun, R. Harding, and O. Mutlu, "Row Buffer Locality Aware Caching Policies for Hybrid Memories," in *International Conference on Computer Design (ICCD)*, 2012.

[34] L. Ramos, E. Gorbatov, and R. Bianchini, "Page Placement in Hybrid Memory Systems," in *International Conference on Supercomputing (ICS)*, 2011.

[35] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson, "Mojim: A Reliable and Highly-Available Non-Volatile Memory System," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[36] C.-K. Luk *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, Chicago, Illinois, June 2005, pp. 190–200.

[37] H. Volos, G. Magalhaes, L. Cherkasova, and J. Li, "Quartz: A Lightweight Performance Emulator for Persistent Memory Software," in *Annual Middleware Conference (Middleware)*, 2015.

[38] A. Bivens, P. Dube, M. Franceschini, J. Karidis, L. Lastras, and M. Tsao, "Architectural Design for Next Generation Heterogeneous Memory Systems," in *International Memory Workshop*, 2010.

[39] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable High-Performance Main Memory System Using Phase-Change Memory Technology," in *ISCA*, 2009.

[40] P. Wu, D. Li, Z. Chen, J. Vetter, and S. Mittal, "Algorithm-Directed Data Placement in Explicitly Managed No-Volatile Memory," in *ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2016.

[41] D. H. Bailey, L. Dagum, E. Barszcz, and H. D. Simon, "Nas parallel benchmark results," in *ACM/IEEE conference on Supercomputing*, 1992.

[42] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, "XSBench the Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis," in *International Conference on Physics of Reactors*, 2014.

[43] J. Tramm, "XSBench: The Monte Carlo Macroscopic Cross Section Lookup Benchmark," 2014, https://github.com/jtramm/XSBench.

[44] A. Rudoff, "Programming Models for Emerging Non-Volatile Memory Technologies," *;login: The USENIX Magazine*, vol. 38, no. 3, 2013.

[45] A. Chatzistergiou, M. Cintra, and S. D. Viglas, "REWIND: Recovery Write-ahead System for In-Memory Non-Volatile Data Structures," *Proceedings of the VLDB Endowment*, vol. 8, no. 5, 2015.

[46] J. E. Denny, S. Lee, and J. S. Vetter, "NVL-C: Static Analysis Techniques for Efficient, Correct Programming of Non-Volatile Main Memory Systems," in *International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2016.

[47] X. Wu and A. L. N. Reddy, "SCMFS: A File System for Storage Class Memory," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.
[48] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging Locks for Non-volatile Memory Consistency," in *International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, 2014.
[49] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: closing the performance gap between systems with and without persistence support," in *International Symposium on Microarchitecture*, 2013.
[50] T. Wang and R. Johnson, "Scalable Logging Through Emerging Non-volatile Memory," *Proceedings of the VLDB Endowment*, vol. 7, no. 10, 2014.
[51] D. Narayanan and O. Hodson, "Whole-system Persistence," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.
[52] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory Persistency," in *ISCA*, 2014.
[53] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Efficient Persist Barriers for Multicores," in *International Symposium on Microarchitecture*, 2015.
[54] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch, "Delegated persist ordering," in *International Symposium on Microarchitecture (MICRO)*, 2016.