Algorithm-Directed Crash Consistence in NVM for High Performance Computing

Jie Ren
jren6@ucmerced.edu

Kai Wu
kwu42@ucmerced.edu

Dong Li
dli35@ucmerced.edu

University of California, Merced

I. INTRODUCTION

Fault tolerance is one of the major design goals for high performance computing (HPC). Because of hardware and software faults and errors, HPC applications can crash during the execution. The most common strategy to make application survive execution crash and enable fault tolerant HPC is the checkpoint/restart mechanism. However, checkpoint is not scalable. If the application state to checkpoint is large, the application has to suffer from large data copy overhead.

The emergence of non-volatile memories (NVM), provides an alternative solution to build fault tolerant HPC. Those memories are persistent, meaning that data are not lost when the system crashes. Hence, using NVM as main memory to build fault tolerant HPC is promising. However, there is no guarantee that the application state in NVM is correct and usable by the recovery process to restart applications, because of volatile hardware caches and out-of-order processors widely deployed in HPC systems. 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 application execution, the common software-based approaches include redo-log or undo-log. Those approaches can 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. Those approaches, unfortunately, can impose large runtime overhead, because they have to log memory update intensively and maintain the corresponding metadata. While such large overhead may be 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 reduce the necessity of using checkpoint for HPC fault tolerance based on the non-volatility of NVM (as main memory), while introducing ignorable runtime overhead.

Our method is based on the following observation. Given a relatively small cache size, 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 reduce runtime overhead.

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 application data structures or selectively flush cache blocks, which introduces ignorable runtime overhead. Such application extension or cache flushing allows us to use algorithm knowledge to reason data consistence and correctness when the application crashes. For HPC applications with a large input problem, data objects critical for restart tend to be consistent and correct in NVM when the application crashes. Using NVM and algorithm knowledge to reason data consistence and correctness, we avoid any logging, and greatly reduce the necessity of using checkpoint.

We study using numerical algorithm knowledge to detect consistence and correctness from three perspectives discussed as following.

II. ALGORITHM-DIRECTED CRASH CONSISTENCE FOR CONJUGATE GRADIENT ALGORITHM

We use conjugate gradient algorithm (CG) from sparse linear algebra as an example to study the feasibility of our method. We leverage the orthogonality relationship between data objects of CG to detect consistence and correctness.

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 of the CG main loop (Lines 3-12

*The full paper of this abstract was published on IEEE International Conference on Cluster Computing (CLUSTER), 2017. http://ieeexplore.ieee.org/document/8048960/

```
1  r ← b − A · x, z ← 0, p ← 0, q ← 0, ρ ← r^T · r;
2  for i ← 1 to n
3     q ← Ap
4     α ← ρ/(p^T · q)
5     z ← z + αp
6     ρ0 ← ρ
7     r ← r − αp
8     ρ ← r^T · r
9     β ← ρ/ρ0
10    p ← p + βp
11    Check r = b − A · z.
12 end for
```

Fig. 1. Pseudocode for CG. Capital letters such as A represent matrices; lowercase letters such as p, q, r represent vectors; Greek letters α, ρ represent scalar numbers.

in Figure 1), before p, r, and z are updated in Lines 10, 7, and 5 in the iteration i; q^i specifies 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 (which should explicitly save the three arrays p, q, and z) for restart, 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 x, we examine the data values of $p^{(x+1)}$, $q^{(x)}$, $z^{(x+1)}$, and $r^{(x+1)}$ 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 x. We then check $p^{(x)}$, $q^{(x-1)}$, $z^{(x)}$ and $r^{(x)}$ and examine the implicit relationship for the iteration $x-1$. We continue the above process, until we find an iteration j ($j < x$) 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 ← b − A · x, z ← 0, p ← 0, q ← 0, ρ ← r^T · r;
2  for i ← 1 to n
3     flush the cache line containing i
4     q[i + 1] ← Ap[i]
5     α ← ρ/(p^T · q)
6     z[i + 1] ← z[i] + αp
7     ρ0 ← ρ
8     r[i + 1] ← r[i] − αp
9     ρ ← r^T · r
10    β ← ρ/ρ0
11    p[i + 1] ← p[i] + βp
12    Check r = b − A · z.
13 end for
```

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

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

We compare runtime overhead between traditional checkpoint, Intel NVML library [1], and our approach. Our evaluation results demonstrate that CG (using Class C as input problem from NPB benchmark suite) with our algorithm extension achieves very good runtime performance: the runtime overhead is less than 3%, while NVM-based checkpoint and PMEM have 43.6% and 329% overhead, respectively. Furthermore, when the input problem size is large enough, the recomputation cost of our approach is limited to only one iteration of the main loop of CG.

III. Algorithm-Directed Crash Consistence for Matrix Multiplication

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.

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

We further study the Monte-Carlo (MC) method. MC is known for its statistics nature and error tolerance. In a sense, the inconsistent 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.

References

[1] Intel NVM library. [Online]. Available: http://pmem.io/nvml/libpmem/