Understanding Application Recomputability without Crash Consistency in Non-Volatile Memory

Jie Ren
University of California, Merced
jren6@ucmeced.edu

Kai Wu
University of California, Merced
kwu42@ucmerced.edu

Dong Li
University of California, Merced
dli35@ucmerced.edu

ABSTRACT

Emerging non-volatile memory (NVM) is promising to be used as main memory, because of its good performance, density, and energy efficiency. Leveraging the non-volatility of NVM as main memory, we can recover data objects and resume application computation (recomputation) after the application crashes. The existing work studies how to ensure that data objects stored in NVM can be recovered to a consistent version during system recovery, a property referred to as crash consistency. However, enabling crash consistency often requires program modification and brings large runtime overhead.

In this paper, we use a different view to examine application recomputation in NVM. Without taking care of consistency of data objects, we aim to understand if the application can be recomputable, given possible inconsistent data objects in NVM. We introduce a PIN-based simulation tool, NVC, to study application recomputability in NVM without crash consistency. The tool allows the user to randomly trigger application crash and then perform postmortem analysis on data values in caches and memory to examine data consistency. We use NVC to study a set of applications. We reveal that some applications are inherently tolerant to the data inconsistency problem. We perform a detailed analysis of application recomputability without crash consistency in NVM.

ACM Reference Format:
Jie Ren, Kai Wu, and Dong Li. 2018. Understanding Application Recomputability without Crash Consistency in Non-Volatile Memory. In *MCHPC'18: Workshop on Memory Centric High Performance Computing (MCHPC'18), November 11, 2018, Dallas, TX, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3286475.3286476

1 INTRODUCTION

Emerging byte-addressable non-volatile memory (NVM) technologies, such as memristors [21] and spin-transfer torque MRAM (STT-MRAM) [12], provide better density and energy efficiency than DRAM. Those memory technologies also have the durability of the hard drive and DRAM-like performance. Those properties of NVM allow us to use NVM as main memory, which blurs the traditional divide between byte-addressable, volatile main memory and block-addressable, persistent storage.

Leveraging the non-volatility of NVM as main memory, we can recover data objects and resume application computation (recomputation) after the application crashes. However, with write-back caching, stores may reach NVM out of order, breaking data object recoverability. Also, data objects cached in the cache hierarchy and stored in NVM may not be consistent. Such inconsistence persists after the application restarts and may impact application execution correctness. Consequently, many existing work [2, 3, 7, 15, 17, 28] studies how to ensure that data objects stored in NVM can be recovered to a consistent version during system recovery, a property referred to as *crash consistency*.

To ensure crash consistency, the programmer typically invokes ISA-specific cache flushing mechanisms via inline assembly or library calls (e.g., `CLFLUSH`) to ensure persist order. To enable crash consistency, log-based mechanisms and the checkpoint mechanism [3, 6, 23, 29] are often employed to make a copy of critical data objects. However, frequent cache flushing, data logging and checkpointing can cause program stalls and large runtime overhead [33].

In this paper, we use a different view to examine application recomputation in NVM. Without taking care of consistency of data objects, we aim to understand if the application can be recomputable, given possible inconsistent data objects in NVM. We define application recomputability regarding *application outcome correctness*. In particular, we claim an application is recomputable after a crash if the application outcome is correct. If an application is recomputable without crash consistency of data objects, then we do not need to employ any cache flushing or logging mechanisms, which improves performance. Having the performance improvement is especially attractive to applications in the field of high-performance computing (HPC).

Evaluating application recomputability without crash consistency is not trivial, because of the following reasons. *First*, to evaluate application recomputation, we must collect data object values in NVM for recomputation, when the crash happens. Without available NVM hardware, the traditional DRAM-based main memory, although often used to emulate NVM [27], can lose data when the crash happens. NVDIMM-N provide a possible solution to address the problem. In particular, when a power failure happens, NVDIMM-N copies the data from the volatile traditional DRAM to flash storage and copies it back when power is restored. In the solution of NVDIMM-N [24], the traditional DRAM is used to emulate NVM, and a small backup power source (e.g., a large battery) is used to make the data copy during the power failure. However, NVDIMM-N is not suitable for our evaluation, because our evaluation involves a large amount of application crash tests. For those tests, a machine with NVDIMM-N has to repeatedly stop and restart, which is time-consuming and impacts the machine reliability.

MCHPC'18, November 11, 2018, Dallas, TX, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6113-2/18/11...$15.00
https://doi.org/10.1145/3286475.3286476

Second, we must determine data consistency when the crash happens. This requires that we compare data in caches and its counterpart in memory. This indicates that we must track data dirtiness of each cache line in caches. To quantify how much inconsistency there is between two data copies, we must also record data values of dirty cache lines. The real hardware does not allow us to track data values and dirtiness of cache lines. The existing simulators usually do not store data values in caches and main memory for simulation.

To evaluate application recomputability without crash consistency, we introduce a PIN [19]-based simulation tool, named *NVC* (standing for *N*on-*V*olatile memory *C*rash tester). In essence, the tool is a PIN-based cache simulator plus rich functionality for crash tests. The tool allows the user to randomly trigger application crash and then perform postmortem analysis on data values in caches and memory to examine data consistency. The tool associates rich data semantics with data values, such that the user can determine which data objects are critical to application recomputatbility. The tool is highly customizable, allowing the user to configure cache hierarchy, cache coherence, and the association between data values and data semantics. The tool also allows the user to test the impact of different cache flushing mechanisms (e.g., `CLFLUSH`, `CLWB` and `CFLUSHOPT`) on data consistency. The tool also integrates the functionality of restarting the application with postmortem data in memory to determine application recomputability.

NVC is useful for several scenarios. Beyond being used to study application recomputability, NVC can be used as a debugger tool. As a debugger tool, NVC can be used to examine if the persist order enforced by the programmer is correct. It can also be used to detect if the data value of a specific variable is consistent as expected by the programmer when the application crashes.

We use NVC to study recomputability of several representative applications from the fields of HPC and machine learning. Using thousands of crash tests, we statistically reveal that some applications do not need crash consistency on critical data objects and are highly recomputable after crashes. We study the reasons that account for the application's inherent tolerance to crash consistency, including memory access pattern, data size, and application algorithm.

The major contributions of the paper are summarized as follows.

- We introduce a tool, NVC, to study application recomputability in NVM without crash consistency, which is unprecedented.
- We use NVC to study a set of applications. Different from the existing work that relies on enforcing crash consistency for application recomputation, we reveal that some applications are inherently tolerant to crash consistency. We perform a detailed analysis of the reasons.

2 PROBLEM DEFINITION AND BACKGROUND

Definition of data objects. Data objects considered in the paper refer to large data structures that store computation results. An example of such data structures is multi-dimensional arrays that represent large matrices. When running an application in a large-scale parallel system, those data objects are often the target to apply checkpoint. In a programming model for NVM (e.g., PMDK [14]), those data objects can be placed into a memory (NVM)-mapped file for the convenience of application restart.

In this paper, we do not consider memory address-related crash consistency. The memory address-related crash consistency problem can cause dangling pointers, multiple frees, and memory leaks. Those problems can easily cause memory segmentation fault when the application restarts. Many existing efforts (e.g., [6, 29]) can address the memory address-related crash consistency. Those efforts are complementary to our work.

In addition, we do not consider those applications with strong demand for *memory transactions*. Those applications include transactional key-value store and the relational database. For those applications, losing data consistency has a severe impact on the functionality of those applications, although some of them usually do not have any problem to restart after crashes.

Application recomputability. We define application recomputability in terms of application outcome correctness. In particular, we claim an application is recomputable after a crash, if the application can restart and the final application outcome remains correct.

The application outcome is deemed correct, as long as it is acceptable according to application semantics. Depending on application semantics, the outcome correctness can refer to precise numerical integrity (e.g., the outcome of a multiplication operation must be numerically precise), or refer to satisfying a minimum fidelity threshold (e.g., the outcome of an iterative solver must meet certain convergence thresholds).

We distinguish restart and recomputability in the paper. After the application crashes, the application may resume execution, which we call *restart*, but there is no guarantee that the application outcome after the application restarts is correct. *If the application outcome is correct, we claim application is recomputable.*

Application restart. When the application crashes, data objects that are placed into a memory (NVM)-mapped file are persistent and usable to restart the application. Other data objects in NVM, either being consistent or inconsistent, are not used for application restarting.

Typically it is the programmer's responsibility to decide which data objects should be placed into the file. Those data objects are critical to application execution correctness. We name those data objects as *critical data objects* in the paper. In many applications, non-critical data objects are either read-only or can be recomputed based on the critical data objects.

In our study, we focus on applications with iterative structures. In those applications, there is a main computation loop dominating computation time. We choose those applications because they are promising to be recomputable after crashes: The iterative structures of those applications may allow the computation of those applications to amortize the impact of corrupted critical data objects. There are a large amount of those applications, including most HPC applications and many machine learning training algorithms.

3 NVC: A TOOL FOR STUDYING APPLICATION RECOMPUTABILITY

NVC is a PIN-based crash simulator. NVC simulates a multi-level cache hierarchy with cache coherence and main memory; NVC also includes a random crash generator, a set of APIs to support the configuration of crash tests and application restart, and a component to examine data inconsistency for post-crash analysis. For the simulation of cache and main memory, different from the traditional PIN-based cache simulator, NVC not only captures microarchitecture level, cache-related hardware events (e.g., cache misses and cache invalidation), but also records the most recent value of data objects in the simulated caches and main memory.

NVC is highly configurable and supports a range of crash tests with different configurations, summarized as follows.

- **Cache configuration**, including the selection of a cache coherence protocol and typical microarchitecture configurations (e.g., cache associativity and cache size);
- **Crash configuration**, including when to trigger the crash and which data objects are critical;
- **Cache flush configuration**, including specifying which cache flushing instruction will be used to ensure data consistency;
- **Recomputation configuration**, including specifying a point within a program for restarting.

We describe the main functionality of NVC as follows.

Cache simulation. Besides supporting the simulation of multi-level, private/shared caches with different capacities and associativity, our cache simulation supports the simulation of cache coherence, which allows us to study the impact of cache coherence on data consistency. With the deployment of a cache coherence protocol, it is possible that a private cache has a stale copy of a cache block, while NVM has the most updated one, causing data inconsistency. NVC can capture such data inconsistency and ignore it if configured to do so. In our evaluation, we use data in NVM to restart and does not count such data inconsistency, because NVM has the most updated data values.

Using PIN to intercept every memory read and write instructions from the application, NVC can get memory addresses and corresponding data values associated with memory accesses. NVC also records cache line information, such as data values in each cache line, cache line dirtiness, and validness.

Our cache simulation supports different cache flushing mechanisms. In particular, we provide three APIs: *flush_cache_line()*, *cache_line_write_back()* and *write_back_invalidate_cache()*. Table 1 contains more details.

Main memory simulation. Different from the traditional microarchitectural simulation for main memory, the main memory simulation in NVC aims to record data values. In particular, NVC uses a hashmap with memory addresses as keys and data values as values. Using the hashmap enables easy updates of data values: whenever the cache simulation in NVC writes back any cache line, the main memory simulation can easily find the corresponding record in the simulated main memory.

Random crash generation. NVC emulates the occurrence of a crash by stopping application execution after a randomly selected instruction. To allow the user to limit crash occurrence to a specific code region (e.g., a function call or a loop), NVC introduces two functions, *start_crash()* and *end_crash()*, to delineate the code region. NVC intercepts the invocations of the two functions to determine where to trigger a crash. To statistically quantify application recomputability, we perform a large number of crash tests (thousands of tests) per benchmark.

To enforce random crash generation, NVC profiles the total number of instructions (specified as N), before crash tests. For each crash test, NVC generates a random number n ($1 \leq n \leq N$). After n instructions are executed, NVC stops application execution. Furthermore, NVC has functionality to report call path information when a crash happens. This is implemented by integrating CCTLib [1] into NVC. CCTlib is a PIN-based library that collects calling contexts during application execution. The call path information is useful for the user to interpret crash results. In particular, the call path information introduces the program context information for analyzing crash results. Having the context information is useful to distinguish those crash tests that happen in the same program position (i.e., the same program statement), but with different call stacks.

Data inconsistent rate calculation. NVC reports data inconsistent rate after a crash happens. The data inconsistent rate is defined in terms of either all data in main memory or specific data objects. If the data inconsistent rate is for all data in main memory, then the data inconsistent rate is the ratio of the number of inconsistent data bytes to the size of whole memory footprint of the application. If the data inconsistent rate is for specific data objects, then the data inconsistent rate is the ratio of the number of inconsistent data bytes of the specific data objects to the size of the data objects.

We use the following method to calculate the data inconsistent rate. We distinguish cache line and cache block in the following discussion. The cache line is a location in the cache, and the cache block refers to the data that goes into a cache line. When a crash happens, NVC examines cache line status in the simulated cache hierarchy. If a cache line in a private cache has "invalidate" status, this cache line is not considered for the calculation of data inconsistence rate, because either another private cache or main memory has an updated version of the cache line data and inconsistent data rate will be based on the new version of the cache line data. If a cache line has "dirty" status, then NVM compares the dirty cache block of the cache line with the corresponding data in main memory to determine the number of dirty data bytes. Note that for a specific dirty cache block, we only consider it once, even if the cache block may correspond to multiple cache lines in the cache hierarchy.

To calculate the data inconsistent rate for the critical data objects, NVC must know memory addresses and data types of those data objects, such that we can determine if a cache line has data of the critical data objects. NVC relies on the user to use a dummy function, *critical_data()*, to pass memory address and data type information of a data object to NVC. This function is nothing but uses memory address and data type as function arguments. NVC intercepts them and associate them with a critical data object.

Application restart. When restarting the application, NVC reads the critical data objects, initializes other data objects using the initialization function of the application, and then resumes the

Table 1: APIs for using NVC.

Signature	Description
void start_crash(); void end_crash();	Define where a crash could happen. A crash could happen within the code region encapsulated by the two APIs.
critical_data(void const *p, char type[], int const size);	Collect the address, type and size information of a critical data object.
consistent_data(void const *p, char type[], int const size);	Collect the address, type and size information of a consistent data object.
void cache_line_write_back(void const *p);	Writes back a dirty cache line containing the address p, and marks the cache line as clean in the cache hierarchy. This API is used to emulate `CLWB`.
void flush_cache_line(void const *p);	Flush a cache line containing address p, invalidate this cache line from every level of the cache hierarchy in the cache coherence domain. This API is used to emulate `CLFLUSH` and `CLFLUSHOPT`.
void write_back_invalidate_cache()	Writes back all dirty cache lines in the processor's cache to main memory and invalidates (flushes) the cache hierarchy. This API is used to emulate `WBINVD`.

main computation loop. Note that when NVC restarts the application, except the critical data objects, other data objects are not usable, even though NVC has data values for those data objects. This is because data semantics for those data values are lost. NVC does not know which data values belong to which data objects.

In our crash tests, we restart the application from the iteration of the main loop where the crash happens, instead of recomputing the whole main loop. To know which iteration to restart, we flush the cache line that contains the iterator at the end of each iteration to make the iterator consistent.

Putting all together. Figure 1 generally depicts the workflow of using NVC. To use NVC, the user needs to insert specific APIs to specify critical data objects, the initialization phase of the application for a restart, and specific code regions for crash tests. The user also needs to configure cache simulation and crash tests. During the application execution, NVC leverages the infrastructure of PIN to instrument the application and analyze instructions for cache simulation. NVC triggers a crash as configured and then perform post-crash analysis to report data inconsistent rate and then restart the application.

An example case. We take MG as an example to explain how we perform a crash test. We use the same method to preform crash tests for other benchmarks.

Figure 2 shows how we add NVC APIs into MG. MG has two critical data objects and their information is passed to NVC in Line 7 and Line 8. The crash test happens in the main computation loop encapsulated by *start_crash()* and *end_crash()*. Right before the main computation loop, we flush whole cache hierarchy (Line 11) to ensure that all data is consistent before we start the crash test. Within the main loop, we flush the cache line containing the iterator at the end of each iteration (Line 15) for the convenience of application restart.

```
1  ...
2  static double u[NR];
3  static double r[NR];
4  ...
5  int main(int argc, char **argv) {
6    int it;
7    critical_data(&u[0], "double", NR);
8    critical_data(&r[0], "double", NR);
9    consistent_data(&it, "int", 1);
10   ...
11   write_back_invalidate_cache();
12   start_crash();
13   for (it = 1; it <= nit; it++) {
14         ...
15     cache_line_write_back(&it);
16   }
17   end_crash();
18   ...
19 }
```

Figure 2: Add NVC APIs into MG

4 RECOMPUTABILITY EVALUATION

4.1 Execution Platform and Simulation Configurations.

In this paper, we simulate a two-level, inclusive cache hierarchy, using the LRU replacement policy. The first level is a private cache (256KB per core and we simulate 8 cores). The second level is a shared cache (20MB). In addition, we use a write-back and no-write allocate policy for the first level cache, and a write-back and write allocate policy for the second level cache. The cache line size is 64 bytes for both caches.

4.2 Benchmark Background.

We use three benchmarks from NAS parallel benchmark (NPB) suite 3.3.1 and one machine learning code (Kmeans [4]) for our study. Those benchmarks are summarized in Table 2. For each benchmark, we use two different input problems, such that we can study the impact of different memory footprint sizes on application recomputability. We describe the benchmarks in details in this section.

Conjugate gradient method (CG). CG is used to compute an approximation to the smallest eigenvalue of a large sparse symmetric positive definite matrix. CG is an iterative method, in the sense that it starts with an imprecise solution and then iteratively converges towards a better solution. CG has a verification phase at the end of CG. The verification tracks the solution convergence towards

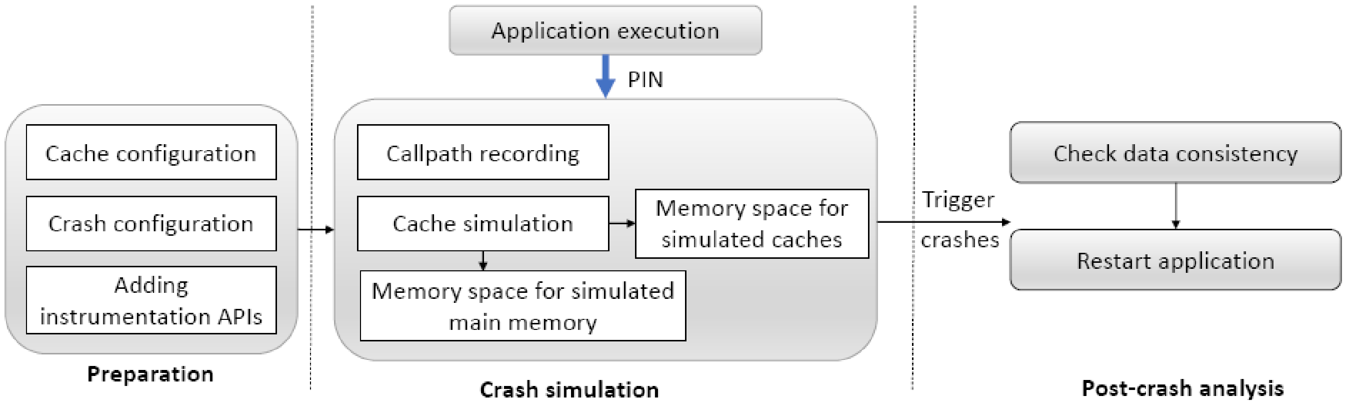
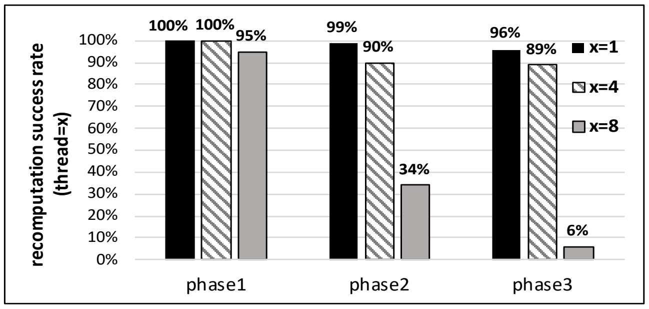
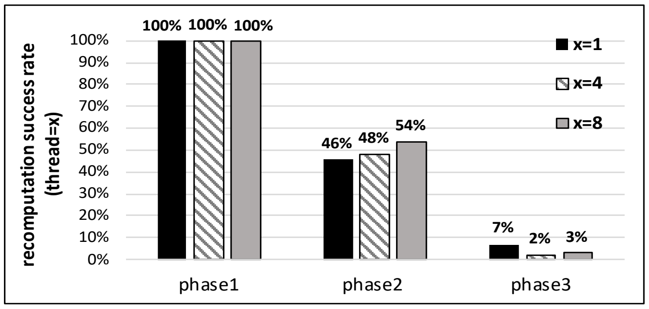


Figure 1: The general workflow of using NVC.

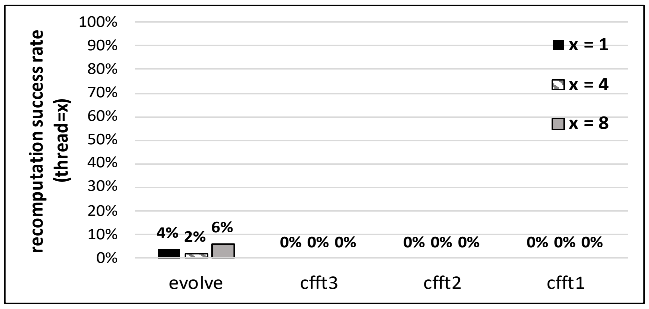


(a) CLASS=A

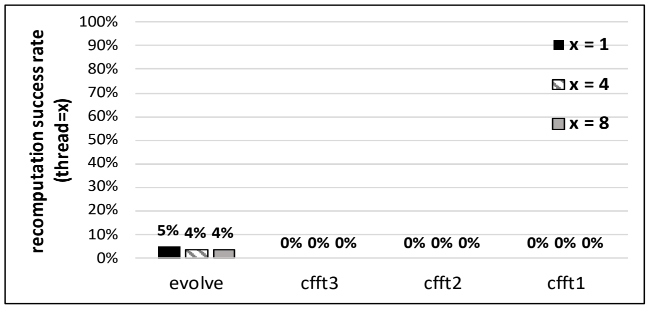


(b) CLASS=B

Figure 3: Recomputation success rate for CG.



(a) CLASS=A



(b) CLASS=B

Figure 4: Recomputation success rate for FT.

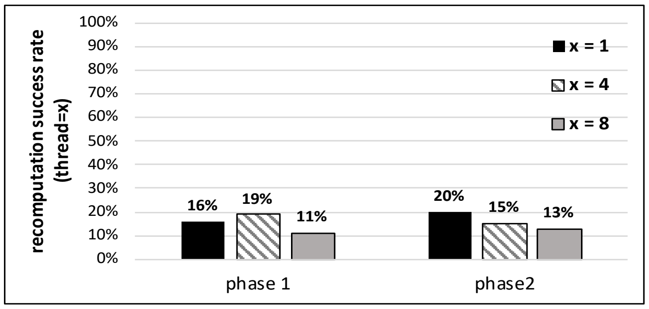
a precision solution. We determine if CG successfully recomputes based on the CG verification.

CG has sparse, unstructured matrix vector multiplication with irregular memory access patterns. In CG, we have five critical data objects (x, p, q, r and z), taking 1% of total memory footprint size.

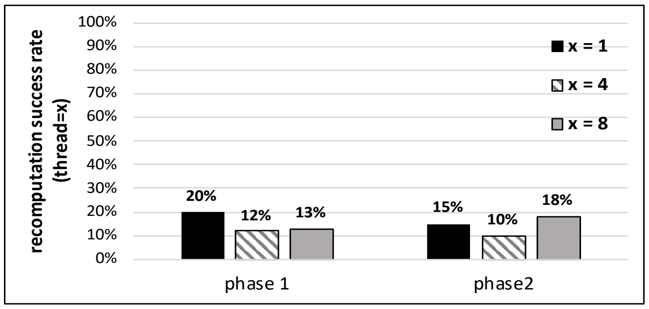
Fourier transform (FT). This benchmark solves a partial differential equation using forward and inverse fast Fourier transform (FFT). FT has a main loop repeatedly performing FFT. At the end of each iteration of the main loop, FT has a verification phase to examine the result correctness of each iteration. The verification phase compares some checksums embedded in data objects of FT with reference checksums to determine the result correctness. Since FT is not an iterative solver and has verification at each iteration of the main loop, we simulation one iteration for our study, in order to save simulation time. We determine if FT successfully recomputes based on the FT verification.

FT has strided memory access patterns. Depending on the input problem size of FT, the stride size can be large, causing intensive accesses to main memory. In FT, we have two critical data objects ($u0$ and $u1$), taking at least 80% of total memory footprint size.

Multigrid method (MG). MG is used to obtain an approximation solution to the discrete Poisson problem based on the multigrid method. MG is also an iterative method, in the sense that MG alternatively works at finer or coarser variants of the input problem

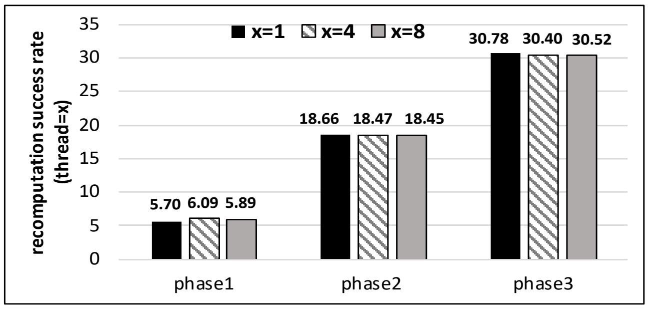


(a) CLASS=A

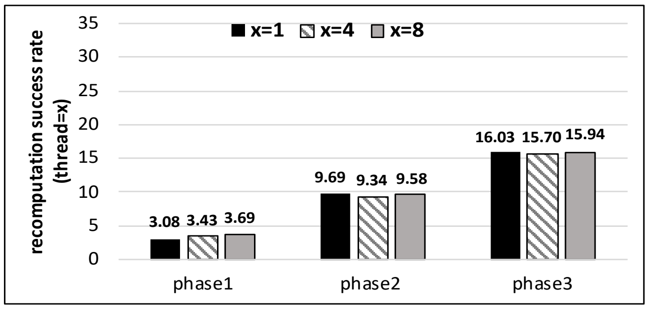


(b) CLASS=B

Figure 5: Recomputation success rate for MG.



(a) kdd as input



(b) 8192 as input

Figure 6: Recomputation success rate for Kmeans.

to compute the increasingly more accurate solutions. MG has a verification phase at the end of MG. The verification tracks the solution convergence towards a precision solution. We determine if MG successfully recomputes based on the MG verification.

MG has either short or long distance data movement, depending on whether MG works on finer or coarser granularity of the input problem. Working at the coarse granularity, MG can cause intensive main memory accesses. In MG, we have two critical data objects (u and r), taking a least 70% of total memory footprint size.

Kmeans. Kmeans is a clustering algorithm to classify a set of input data points into n clusters. Kmeans is an iterative algorithm: In each iteration, each data point is associated with its nearest cluster, and then a new cluster centroid for each cluster is formed. Kmeans iteratively determines the cluster centroid for each cluster. Kmeans will stop clustering when a convergence criteria is met. Given a set of data points as input, the algorithm identifies related points by associating each data point with its nearest cluster, computing new cluster centroids and iterating until converge.

Kmeans has streaming memory accesses (lacking temporal data locality). In Kmeans, we have *centers* as critical data objects, which is the position information of cluster centroids. *centers* takes less than 1% of total memory footprint.

Table 2: Benchmark information.

Benchmark	Description	Memory footprint size of two input problems
CG	Iterative solver	Class A: 55MB, Class B: 398MB
FT	Spectral method	Class A: 321MB, Class B: 1283MB
MG	Iterative solver	Class A: 431MB, Class B: 431MB
Kmeans	Clustering analysis	kdd_cup: 133MB, 819200: 222MB

4.3 Recomputability Summary.

We evaluate and characterize application recomputability by triggering crashes at different execution phases and changing the number of threads to run benchmarks. Each case of our study is a combination of using a specific number of threads and triggering crashes at a specific execution phase. Each case of our study includes 100 crash tests. For each crash test, we stop the benchmark in one iteration of the main loop, and then restart the benchmark and examine if the benchmark can complete successfully by running the remaining iterations of the main loop. For the benchmarks CG and MG, the number of iterations of the main loop from one run to another remains constant, but for Kmeans, the number of iterations is not constant from one run to another. In addition, Kmeans can always recompute (converge) successfully after the crash. To evaluate the recomputability of Kmeans, we use the number of iterations after the crash for Kmeans to converge as a metric. If Kmeans needs more iterations to converge after the crash than before the crash, then Kmeans has bad recomputability.

4.4 Recomputability at Different Execution Phases

We trigger crashes at different execution phases of each benchmark. For CG and Kmeans, we evenly divide the whole iteration space of the main loop into three parts, each of which corresponds to one phase. For MG, we evenly divide the whole iteration space into two parts (not three parts), because MG has a small number of iterations. For FT, we only have one iteration, as described in Appendix A. We divide the iteration into four phases (evolve, cfft3, cfft2, and cfft1) based on algorithm knowledge. Our following discussion focuses on the recomputation results of using one thread to run benchmarks. Figures 3a-6b shows the results.

CG with Class A as input shows strong recomputability: at least 96% of all crash tests can recompute successfully. However, when we use a larger input (Class B), CG shows weak recomputability in Phases 2 and 3 (the recomputation success rate is 46% and 7% respectively). FT shows very weak recomputability: the recomputability success rate is less than 6% in all cases. MG also shows relatively weak recomputability: the recomputability success rate is 15%-20%. Kmeans can recompute successfully in all cases, showing strong recomputability.

Observation 1. Different applications have large variance in recomputability. Some applications (e.g., CG and Kmeans) can recompute successfully in almost all cases, while some applications (e.g., FT) have close to zero tolerance to crash inconsistency.

Observation 2. Application recomputability is related to the input problem size. With different input problems, application recomputability can behave differently.

The observation 2 is aligned with our intuition. The application with a larger input problem can lead to a smaller portion of data objects in the cache hierarchy. This reduces the data inconsistent rate when a crash happens, and results in a better possibility to recompute.

When crashes happen at different execution phases of CG (Class B as input), CG shows different recomputability (100%, 46% and 7% for Phases 1, 2 and 3 respectively). CG does not show good recomputability at the late phase (Phase 3). Kmeans shows the similar results: when crashes happen at different execution phases, Kmeans uses a different number of iterations to converge. At the late phase (Phase 3), Kmeans needs a larger number of iterations.

Observation 3. Application recomputability is different across different execution phases.

The iterative structure of some applications has capabilities of tolerating approximate computation by amortizing approximation across iterations [11]. A crash and restart cause approximate computation, because of data inconsistence. A crash happening at the early execution phase has more iterations to tolerate approximation and has a higher possibility to recompute.

Implication 1. If we enforce crash consistency to improve application recomputability, we do not need to enforce it throughout application execution. Reducing the necessity of enforcing crash consistency is helpful to improve application performance. Some applications with specific input problems are naturally recomputable after crashes. They do not need to enforce crash consistency.

4.5 Recomputability with Different Numbers of Threads

We use 1, 4 or 8 threads to run applications. The results are shown in Figures 3a-6b. For CG with Class A, the number of threads has a significant impact on recomputability. As the number of threads increases, the recomputability goes worse. The recomputability for 1, 4 and 8 threads is 99%, 93%, and 45%, respectively. However, for CG (Class B), FT, MG and kmeans, the recomputability is not sensitive to the number of threads.

Observation 4. Application recomputability can be negatively impacted by the number of threads. However, applications with weak recomputability (e.g., FT and MG) remain to have weak recomputability when using different numbers of threads.

We attribute the above observation to the possible larger working set for critical data objects in caches when using a larger number of threads. When a crash happens, having a larger working set size for critical data objects in caches means more data is inconsistent. On the other hand, using a larger number of threads can cause higher data consistent rate, because of more cache line eviction. More cache line eviction implicitly causes more data to be consistent. We discuss the data inconsistent rate in Section 4.6.

Furthermore, Kmeans seems to be a special case. Kmeans has strong recomputability: the recomputation rate is always 100%, no matter how many threads we use to run Kmeans and trigger crashes. Different from CG (Class A) that also has strong recomputability, the recomputability of Kmeans is not impacted by the number of threads at all. We attribute such observation to the strong tolerance to data corruption of Kmeans.

Implication 2. When using the different number of threads, we must use different strategies to ensure application recomputability. When using a larger number of threads to run an application, there is often a need to enforce stronger crash consistency.

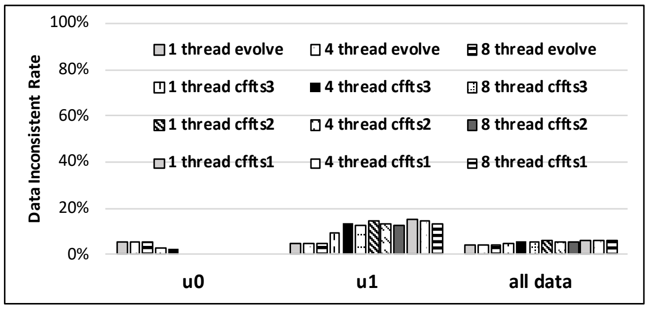
4.6 Analysis based on Data Inconsistent Rate

Figures 7a-10b show the data inconsistent rate for individual critical data objects as well all data objects. We define the data inconsistent rate in Section 3.

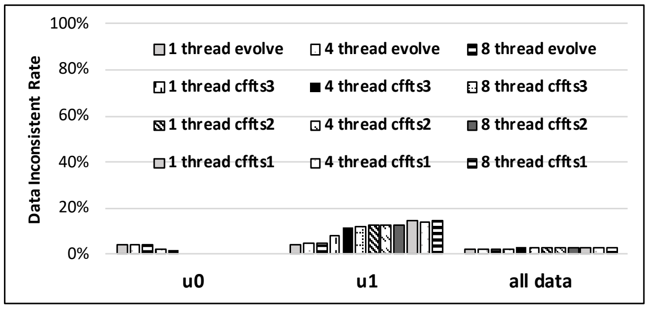
For CG, the data objects q and r have a high inconsistent rate in all cases. The variance of data inconsistent rate across cases is also small. We conclude that the recomputability of CG seems to be less correlated with the data inconsistent rate of these two data objects. We further notice that the data objects p and z has a large variance in data inconsistent rate across cases, when using Class A as input. For p, using a larger number of threads causes higher data inconsistent rate; for z, using a larger number of threads has opposite effects. We conclude that using a larger number of threads, accessing to p and z may have opposite impact on application recomputability. Given the fact that the recomputability of CG (Class A as input) is not sensitive to the number of threads, the impacts of p and z on application recomputability seem to be neutralized.

We have the similar observations for other benchmarks.

Observation 5. We cannot easily explain the variance of application recomputability based on the data inconsistent rate. There seems to be a small correlation between application recomputability and the data inconsistent rate.

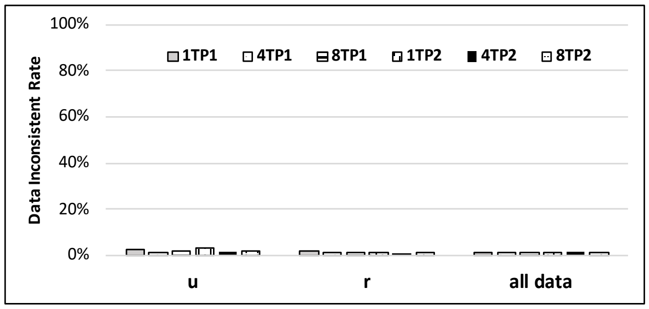


(a) CLASS=A

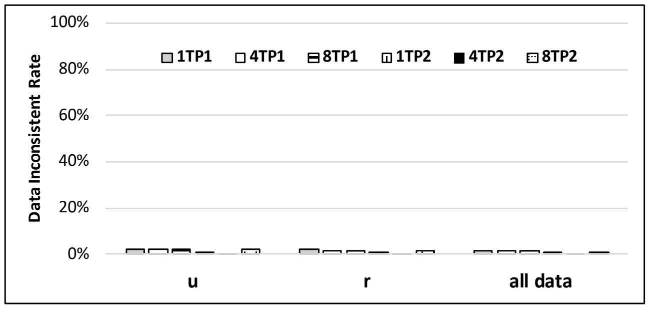


(b) CLASS=B

Figure 8: Data inconsistent rate for FT.



(a) CLASS=A



(b) CLASS=B

Figure 9: Data inconsistent rate for MG. In the legend, "T" stands for thread and "P" stands for phase. xTyP means using x threads and trigger crashes in Phase y.

The above observation may be because of the following reason. The data inconsistence rate only tells us that data is inconsistent, but cannot quantify the value difference between caches and main memory. Two crash tests may cause the same data inconsistent rate for a data object, but have quite different data values in the data object. Different data values can cause different application recomputability. Using the different number of threads and different input problems can cause a big difference in data values between caches and main memory when the crash happens. Such big differences cause different application recomputability.

Implication 3. Considering the cache effects to determine application inconsistent rate is not sufficient to understand application recomputability. We must also consider how different data values in caches and main memory are when the crash happens.

4.7 Discussions and Future Work

NVC is based on PIN that uses binary instrumentation. Running an application with a large memory footprint and intensive memory accesses with NVC can be time-consuming. In our experiments, using NVC to run the application with large input size can cause hundreds of times slowdown. Such long execution time brings challenges for running a large number of crash tests. We plan to extend our work by introducing a new technique to accelerate our analysis. In particular, we plan to proportionally scale down the application's data set and cache size for faster simulation without losing the result correctness for quantifying application recomputability.

NVC allows us to learn the application recomputability and the reason behind. We plan to learn more representative applications and introduce a mechanism to leverage application recomputability to avoid checkpoint or cache flushing for better performance.

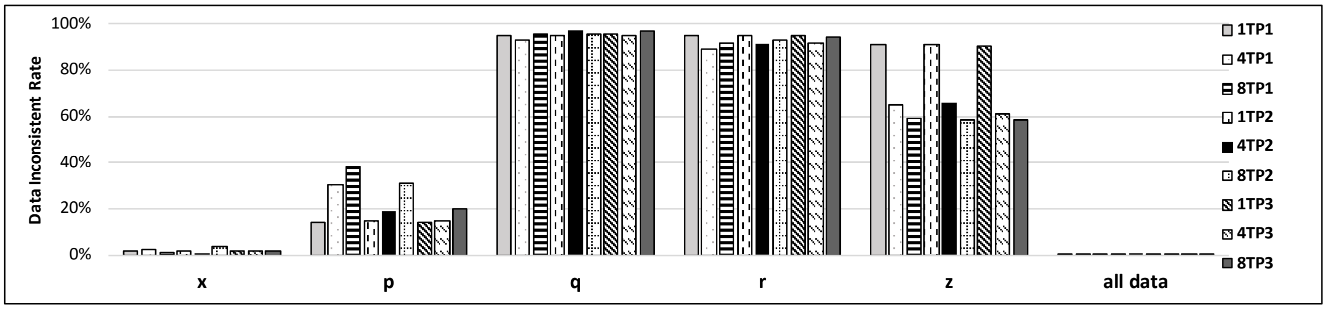
5 RELATED WORK

Many existing work focuses on enabling crashing consistency in NVM, using software- and hardware-based approaches. Different from the existing work, we study application recomputability without crash consistency. We review the existing work related to crash consistency as follows.

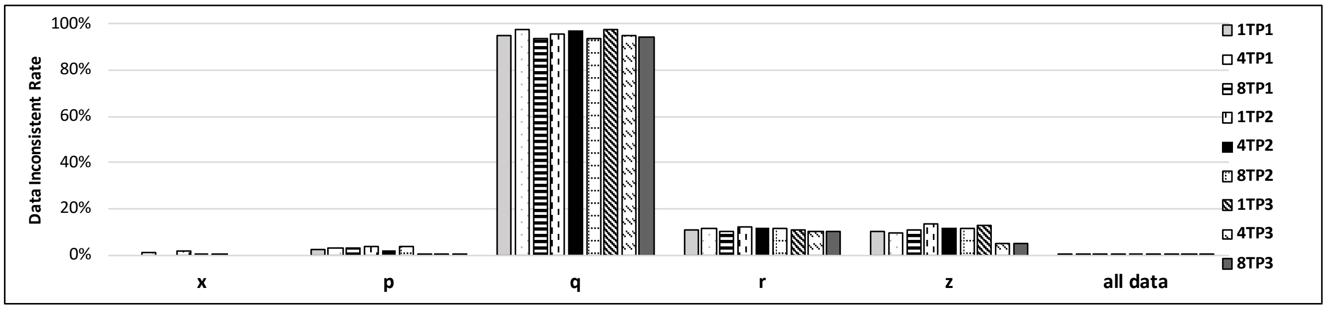
Software support for crash consistency. Enabling crash consistency in NVM with software-based approaches is widely explored. Undo logging and redo logging are two of the most common methods to enable crash consistency, often based on atomic and durable transactions. Using undo and redo logging, once a transaction fails or the application crashes, any uncommitted modifications are ignored, and the application rolls back to the latest version of data in the log.

Persistent Memory Development Kit (PMDK) [13] from Intel supports the transaction system in NVM by undo logging. Similarly, NV-Heaps [7], REWIND [3] and Atlas [2] adopt write-ahead undo logging in NVM. Kolli et al. [15] propose an undo logging that minimizes the write ordering constraint by delaying to commit the data modification.

Mnemosyne [28], a set of programming APIs and libraries for programming with persistent memory, uses redo logging. Lu et al. [17] optimize Mnemosyne to reduce the overhead of supporting transaction by delaying and minimizing the cache flushing. To



(a) CLASS=A



(b) CLASS=B

Figure 7: Data inconsistent rate for CG. In the legend, "T" stands for thread and "P" stands for phase. xTyP means using x threads and trigger crashes in Phase y.

achieve that, they maintain the correct overwrite order of data but do not write them back into memory immediately. A full processor cache flushing will be scheduled when they accumulate enough uncommitted data. Giles et al. [10] provide a redo logging based lightweight atomicity and durability transaction by ensuring fast paths to data in processor caches, DRAM, and persistent memory tiers.

Some existing work focuses on enabling crash consistency for specific data structures, including NV-Tree [32], FPTree [22], NVC-Hashmap [25], CDDS [26] and wBTree [5]. Those data structures support atomic and durable updates, and hence support crash consistency.

Our work can be very helpful for the above work. In particular, NVC can be used to check if crash consistency based on undo log and redo log enables data consistence as expected.

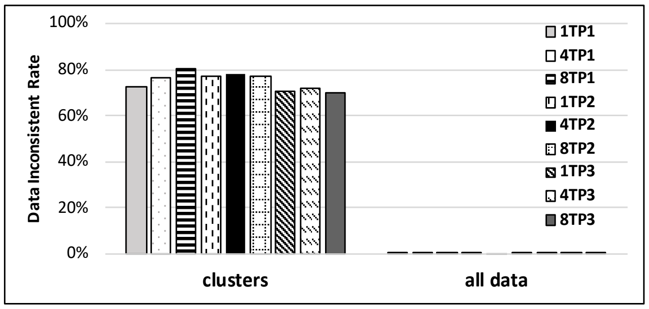
Some existing work considers crash consistency with the context of file system [8, 9, 30]. NOVA [31] is such a file system optimized for heterogeneous memory (DRAM and NVM) systems. It provides strong consistency guarantees and maintains independent logs for each inode to improve scalability. Octopus [16] is another example. Octopus is a RDMA-enabled distributed persistent memory file system. Octopus can have high performance when enforcing metadata consistency, by a "collect-dispatch" transaction. With the collect-dispatch transaction, Octopus collects data from remote servers for local logging and then dispatching them to remote sides by RDMA primitives.

Among the above software-based work, some of them [15, 17] in fact relaxes requirements on crash consistency and does not require crash consistency to be timely enforced, in order to have better performance. Since our work does not require crash consistency, our work also relaxes requirements on crash consistency.

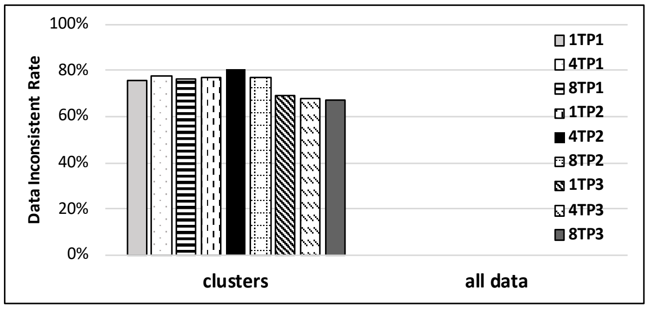
Hardware support for crash consistency. Lu et al. [18] use hardware-based logging mechanisms to relax the write ordering requirements both within a transaction and across multiple transactions. To achieve such goal, they largely modify the cache hierarchy and propose a non-volatile last level CPU cache. Ogleari et al. [20] combine undo and redo hardware logging scheme to relax ordering constraints on both caches and memory controllers for NVM-based systems. Meanwhile, to minimize the frequency of using write-back instructions, they add a hardware-controlled cache to implement a writeback cache. Our work is different from the above hardware-based work, because we do not require hardware modification for crash consistency.

6 CONCLUSIONS

Using NVM as main memory brings an opportunity to leverage NVM's non-volatility for application restarting and recomputing based on the remaining data objects in NVM, after the application crashes. Different from the existing work that enables crash consistency for application recomputation, we statistically quantify recomputability of a set of applications without crash consistency in NVM. We develop a tool (named NVC) that allows us to trigger random crash, examine data consistency and restart application for our study. Using the tool, we real that some applications have very good recomputability without crash consistency on critical data objects. Our work is the first one that studies application recomputability without crash consistency. Our work opens a door to remove runtime overhead of those crash-consistency mechanisms



(a) kdd_cup as input



(b) 819200 as input

Figure 10: Data inconsistent rate for kmeans. In the legend, "T" stands for thread and "P" stands for phase. xTyP means using x threads and trigger crashes in Phase y.

(e.g., logging and checkpoint). Our work makes NVM a more feasible solution for application recomputation in those fields with the high-performance requirement.

REFERENCES

[1] Milind Chabbi, Xu Liu, and John Mellor-Crummey. 2014. Call Paths for Pin Tools. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization.*

[2] 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).*

[3] 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.

[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC).*

[5] Shimin Chen and Qin Jin. 2015. Persistent B+-trees in Non-volatile Main Memory. *Proc. VLDB Endow.* 8, 7 (Feb. 2015), 786–797.

[6] Joel Coburn, Adrian Caulfield, Ameen Akel, Laura Grupp, Rajesh Gupta, Ranjit Jhala, and Steve Swanson. 2011. NV-heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proc. of 16th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS'11).*

[7] 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 the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[8] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09).*

[9] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14).*

[10] E. R. Giles, K. Doshi, and P. Varman. 2015. SoftWrAP: A lightweight framework for transactional support of storage class memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST).*

[11] Serge Gratton, Philippe L. Toint, and Anke Tröltzsch. 2011. How much gradient noise does a gradient-based linesearch method tolerate?

[12] Xiaochen Guo, Engin Ipek, and Tolga Soyata. 2010. Resistive Computation: Avoiding the Power Wall with Low-Leakage, STT-MRAM Based Computing. In *International Symposium on Computer Architecture (ISCA).*

[13] Intel. 2014. Persistent Memory Development Kit. https://pmem.io/. (2014).

[14] Intel. 2014. Intel NVM Library. http://pmem.io/nvml/libpmem/. (2014).

[15] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. 2016. High-Performance Transactions for Persistent Memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16).*

[16] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. 2017. Octopus: an RDMA-enabled Distributed Persistent Memory File System. In *2017 USENIX Annual Technical Conference (USENIX ATC 17).*

[17] Youyou Lu, Jiwu Shu, and Long Sun. 2016. Blurred Persistence: Efficient Transactions in Persistent Memory. *Trans. Storage* 12, 1, Article 3 (Jan. 2016), 29 pages.

[18] Y. Lu, J. Shu, L. Sun, and O. Mutlu. 2014. Loose-Ordering Consistency for persistent memory. In *2014 IEEE 32nd International Conference on Computer Design (ICCD).*

[19] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proc. 2005 ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '05).* Chicago, IL, 190–200.

[20] Ethan L. Miller Matheus Ogleari and Jishen Zhao. 2018. Steal but No Force: Efficient Hardware Undo+Redo Logging for Persistent Memory Systems. In *Proceedings of the 24th IEEE International Symposium on High-Performance Computer Architecture (HPCA).*

[21] P. Mazumder, S. M. Kang, and R. Waser. 2012. Memristors: Devices, Models, and Applications [Scanning the Issue]. *Proc. IEEE* (2012), 1911–1919.

[22] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *SIGMOD.*

[23] Andy Rudoff. 2013. Programming Models for Emerging Non-Volatile Memory Technologies. *The USENIX Magazine* 38, 3 (2013), 40–45.

[24] Arthur Sainio. 2016. NVDIMM: Changes are Here So WhatâĂŹs Next?. In *In-Memory Computing Summit 2016.*

[25] David Schwalb, Markus Dreseler, Matthias Uflacker, and Hasso Plattner. 2015. NVC-Hashmap: A Persistent and Concurrent Hashmap For Non-Volatile Memories. In *Proceedings of the 3rd VLDB Workshop on In-Memory Data Mangement and Analytics (IMDM '15).*

[26] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies.*

[27] Haris Volos, Guilherme Magalhaes, Ludmila Cherkasova, and Jun Li. 2015. Quartz: A Lightweight Performance Emulator for Persistent Memory Software. In *Proc. 16th Annu. Middleware Conference (Middleware '15).* Vancouver, Canada, 37–49.

[28] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

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

[30] X. Wu and A. L. N. Reddy. 2011. SCMFS: A file system for Storage Class Memory. In *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC).* 1–11.

[31] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST'16).*

[32] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15).*

[33] S. Yang, K. Wu, Y. Qiao, D. Li, and J. Zhai. 2017. Algorithm-Directed Crash Consistence in Non-volatile Memory for HPC. In *2017 IEEE International Conference on Cluster Computing (CLUSTER).*