Exploring Non-Volatility of Non-Volatile Memory for High Performance Computing Under Failures

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

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

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

***Abstract*—Hardware failures and faults often result in application crash in HPC. The emergence of non-volatile memory (NVM) provides a solution to address this problem. Leveraging the non-volatility of NVM, one can build *in-memory* checkpoints or enable crash-consistent data objects. However, these solutions cause large memory consumption, extra writes to NVM, or disruptive changes to applications. We introduces a fundamentally new methodology to handle HPC under failures based on NVM. In particular, we attempt to use remaining data objects in NVM (possibly stale ones because of losing data updates in caches) to restart crashed applications. To address the challenge of possibly unsuccessful recomputation after the application restarts, we introduce a framework *EasyCrash* that uses a systematic approach to automatically decide how to selectively persist application data objects to significantly increase possibility of successful recomputation. EasyCrash enables up to 30% improvement (20% on average) in system efficiency at various system scales.**

I. Introduction

The extreme-scale HPC systems face a grand challenge on system reliability. Hardware failures and transient hardware faults often result in application failures (application crashes). Application crashes lose application's work and decrease system efficiency. A typical HPC system nowadays has a mean-time between failure (MTBF) of tens of hours [1]–[4], even with hardware- and software-based protection mechanisms. It is expected that the failure rate could further increase in the future, as the complexity and scale of HPC systems increases. This indicates that a larger portion of computation cycles will have to be used to handle application failures [5], [6].

Byte-addressable non-volatile memory (NVM) technologies, such as Intel Optane DC persistent memory DIMM [7], are emerging. NVM provides higher density and power efficiency than DRAM while providing DRAM-comparable performance. Recent efforts have demonstrated the possibility of using NVM as main memory [8]–[10] with `load/store` instructions and for future HPC [11]–[13]. The emerging NVM provides new opportunities to handle HPC under failures.

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. Data objects cached in the cache hierarchy and stored in NVM may not be consistent during application crash. Such inconsistency persists after the application restarts and impacts application execution correctness. Consequently, many existing work [8], [9], [14] studies how to ensure that data objects stored in NVM can be recovered to a consistent version for successful recomputation, a property referred to crash consistency.

978-1-7281-6677-3/20/$31.00 © 2020 IEEE

To enable crash consistency, the existing solutions use *in-memory* checkpoint/restart (C/R) [10], [15] or build crash-consistent data objects [9]. However, those solutions have limitations when applied to NVM and HPC. (1) Using NVM as a fast persistent media to implement in-memory C/R creates bottleneck in memory capacity and worsens the endurance problem faced by NVM. In particular, creating checkpoints in NVM (used as main memory) can double or even triple memory footprint of the application. For those scientific simulations with large data sets, reducing the effective capacity of NVM constrains the simulation scale that scientists can study. In addition, NVM has limited endurance and can tolerate a limited number of writes (even with wear-leveling techniques in place [10], [15]). Since in-memory checkpoints are written to NVM, checkpointing causes a number of additional writes in NVM. (2) Building crash-consistent data objects can cause large modifications to applications. In order to enable crash consistency, the existing efforts record updates to data objects by creating undo/redo logs [8], [16] or metadata [17], which often introduce new data structures and memory synchronization. Such disruptive modifications are difficult to be adopted by legacy HPC applications, which are often large and dominate scientific simulations in HPC data centers.

In conclusion, high requirements of HPC on memory consumption, performance and code stableness call for a new solution to explore non-volatility of NVM to handle failures.

In this paper, we introduce EasyCrash, a framework that relaxes the requirement on crash consistency and leverages error resilience intrinsic to HPC applications to handle application crashes. EasyCrash employs a fundamentally new methodology: It does not create data copy or record modifications to data objects for high crash consistency; Instead, it attempts to use remaining data objects in NVM (possibly inconsistent ones because of losing data updates in caches) to restart crashed application, based on the prevalent characteristics of error resilience in HPC applications.

Relaxing the requirement on crash consistency raises a risk of unsuccessful recomputation. The random occurrence of crashes can leave data objects in NVM in any inconsistent state with no guarantee on successful application recomputation. To address this challenge, we perform crash tests and characterize how the success of application recomputation is sensitive to data consistency of data objects at various execution phases. Based on the study, we use analytical models to decide where to persist data objects to enable high application recomputability. To minimize runtime overhead of cache flushing, we use correlation analysis to decide which data objects are the most critical to successful application recomputation. EasyCrash only flushes cache blocks of those data objects at specific execution phases. Such selective cache flushing constrains

the relaxed crash consistency (but not too much), hence increasing application recomputability with high performance. To ensure 100% of application recomputation, EasyCrash is built upon the traditional checkpointing on *storage* to handle unsuccessful recomputation. However, with EasyCrash, we can reduce checkpoint frequency, because EasyCrash makes many crashes successfully recompute without rolling back to the last checkpoint. Reducing checkpoint frequency is critical to improve system efficiency. It was reported that up to 50% time in HPC data centers is spent in checkpointing [18], [19].

EasyCrash is based on three observations. First, many HPC applications are characterized with large data sets and most of them may not be in caches during application execution, because of limited cache capacity. This indicates that using cache flushing (instead of making data copy) to persist data objects can potentially reduce a large number of writes and memory consumption.

Second, many HPC applications have intrinsic error resilience, which indicates that computation inaccuracy because of relaxed consistency is tolerable by HPC applications. In particular, many popular HPC applications, such as iterative solvers (e.g., the preconditioned conjugate gradient method, Newton method, and multigrid method), Monte Carlo-based simulations [20] and some machine learning workloads (e.g., Kmeans and CNN training), have natural error resilience to localized numerical perturbations, because they require computation results to converge over time. As a result, they can intrinsically tolerate computation inaccuracy [21]–[23].

Third, many HPC applications have application-specific acceptance verification based on physical laws and math invariant. Leveraging the verification, the application can detect whether computation results are acceptable before delivering them to end users. For example, large-scale computational fluid dynamics simulations examine result correctness by making a comparison to exact analytical results [24]. Those applications with acceptance verification reduces the probability of producing incorrect results.

EasyCrash persists some data objects at certain execution phases of the application. Once a crash happens, the application immediately restarts using remaining consistent and inconsistent data objects in NVM. Application-specific acceptance verification checks if the recomputation result is correct. If the application cannot recompute successfully, then the application goes back to the last checkpoint.

EasyCrash meets high requirements of HPC to handle failures, and addresses the limitation in the existing efforts. First, EasyCrash does not create data copy, hence saving memory capacity and enabling scientific simulation with larger memory footprint. Second, EasyCrash flushes cache blocks using special instructions (e.g., `CLWB`), which do not write back cache lines [1] to main memory, if the corresponding cache blocks are clean or not resident in caches; Hence EasyCrash reduces unnecessary writes to NVM. Saving writes to NVM is beneficial for the performance of persisting data objects. Third, EasyCrash does not change data structures and involves few changes to the application; Hence, EasyCrash brings a feasible and highly beneficial solution to HPC.

In summary, this paper makes the following contributions:

- A methodology for HPC under failures, by leveraging NVM and error resilience intrinsic to many HPC applications;
- Characterization of HPC application recomputability after crashes; Theoretical analysis to provide guidance on persisting data objects with guarantee on high performance and system efficiency;
- EasyCrash transforms 54% of crashes that cannot correctly recompute into correct computation, while incurring 1.5% performance overhead (on average); 77% of crashes successfully recomputes. As a result, EasyCrash enables up to 30% improvement (20% on average) in system efficiency.

II. Background

A. Cache Flushing for Data Persistence

To ensure persistency and consistency of data objects in NVM, the programmer typically employs ISA-specific cache flushing instructions (e.g., `CLFLUSH`, `CLFLUSHOPT` and `CLWB`). To persist a large data object, the current common practice is to flush all cache blocks of the data object [16], even when some of them are not in the cache. This is because we do not have a performant and cost-effective mechanism to track dirty cache lines and whether a specific cache block is resident in the cache. However, flushing a clean cache block or a non-resident cache block is less expensive than flushing a dirty one resident in the cache, because there is no writeback.

B. Terminology and Problem Definition

Data objects. We focus on heap and global data objects, but not on stack data objects. Choosing those data objects is based on our survey on 60 HPC applications [25]: Major memory footprint and most important data objects (important to execution correctness) in HPC applications are heap and global ones. Our observation is aligned with the recent work [26], [27]. We study data objects (but not the whole system state) for recomputation study, because of two reasons: (1) The current main-stream NVM programming models [8], [14], [16] focus on persisting data objects for easy restart; (2) persisting the whole system state can cause large performance overhead.

Application recomputability. We define application recomputability in terms of application outcome correctness. In particular, we claim an application recomputes successfully after a crash, if the final application outcome is correct. The 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). Leveraging application-level acceptance verification, we can determine correctness of application outcome and execution. We define application recomputability with a high requirement on performance. In particular, for an HPC application with iterative structures, we claim that it recomputes successfully when its outcome is correct *and* it does not take extra iterations to finish.

Application recomputability quantifies the *possibility* that once a crash happens, the application recomputes successfully. To calculate application recomputability, one has to perform a number of crash tests to ensure statistical significance. Each test triggers a random crash and restarts the application. We use the ratio of the number of tests that successfully recompute to total number of tests as *application recomputability*. All

[1] We distinguish cache line and cache block. The cache line is a location in the cache, and the cache block refers to the data that goes into a cache line.

of the crash tests to calculate application recomputability form a *crash test campaign*. We distinguish "restart" and "recompute". After the application crashes, the application may resume execution, which we call *restart*. If the application outcome is correct and there is no need of extra iterations to finish, we claim the application *recomputes*.

System efficiency is defined as the ratio of accumulated useful computation time to total time spent on the HPC system. The total time includes time for useful computation, checkpoint, lost computation because of crashes, and recovery.

Application target. We focus on HPC applications. The effectiveness of EasyCrash is affected by the acceptance verification and error resilience characteristics of those applications.

The acceptance verification can happen at the end of the application [28] or during application execution [29], which detects whether the application state is corrupted before delivering results to users. Typically it is the programmer's responsibility to write the acceptance verification to ensure that computation results do not violate application-specific properties (e.g., convergence conditions or numeric tolerance for result approximation). The application-level acceptance verification is common in HPC applications, and increasingly common, because of the strong needs of increasing confidence in the results offered by HPC applications. The application-level acceptance verification has been commonly employed in the existing work to determine execution correctness of HPC applications [21]–[23], [30], [31].

A large number of HPC applications are characterized with an iterative structure (a main computation loop dominating computation time) and acceptance verification. Our comprehensive survey on 60 HPC applications from various scientific and engineering fields support the above conclusion [25]. Many of those HPC applications are known to be naturally resilient to computation inaccuracy [21], [32]. They are promising to be recomputable after crashes, because they work by improving the solution accuracy step by step, which is helpful to eliminate errors. For example, a convergent iterative method can tolerate data inconsistency during the convergence process. Because of the prevalence and importance of those applications, the recent work on approximate computing also focuses on those applications [33], [34]. We expect those applications become more common in the future, in order to enable higher performance and energy efficiency [35], [36].

Failure model. We focus on application failures which could be caused by power loss, hardware failures or faults. We do not consider application failures caused by software bugs, because those bugs can prevent recomputation. After application failure, NVM is still accessible for restart [8], [14].

Optane DC persistent memory module. The recent release of Intel Optane DC persistent memory module (DCPMM) is used as *main memory* and promising for future HPC [37]. We put our discussion in the context of this hardware to make our work more useful. DCPMM can be configured as either memory mode or app-direct mode. With the memory mode, DCPMM does not provide data persistency, hence not relevant to our work. We assume that DCPMM uses *app-direct* mode. With this mode, DCPMM is provisioned as persistent memory with byte addressability. The application can directly access it using load/store instructions, and flushing CPU caches makes data persistent in Optane DCPMM. To locate data objects in DCPMM after a failure, the user leverages a memory-mapped file-based mechanism. This mechanism is commonly used in the exiting NVM-aware programming models [8], [38].

```
1  static double u[NR];
2  static double r[NR];
3  void main(int argc, char **argv) {
4    int it;
5    initialize();
6    for (it = 1; it <= nit; it++) {//main comp loop
7      for () { // a first-level inner loop;
8           ...
9           for() {...}// a second-level inner loop
10     }
11     ...
12     for () { // a first-level inner loop;
13          ...
14           cache_block_flush(u, NR*sizeof(double));
15           cache_block_flush(r, NR*sizeof(double));
16     }
17     cache_block_flush(&it, sizeof(int));
18   }
19   //result verification
20   ...
21 }
```

(a) Persisting data objects during MG execution.

```
1  static double u[NR];
2  static double r[NR];
3  void main(int argc, char **argv) {
4      int it,it_old;
5      initialize();
6      load_value(u,NR*sizeof(double));
7      load_value(r,NR*sizeof(double));
8      load_value(&it_old,sizeof(int));
9      for (it = it_old; it <= nit; it++) {//main loop
10         ...
11         //flush cache blocks
12     }
13     //result verification
14     ...
15 }
```

(b) Restart MG.

Fig. 1: Study recomputability of MG with NVCT.

C. Study Application Recomputability

To study application recomputability in NVM, we use a PIN-based crash emulator, NVCT [39]. NVCT includes a simulated multi-level, coherent cache hierarchy and main memory, and a random crash generator. Different from the traditional cache simulator, NVCT not only captures microarchitecture level, cache-related hardware events such as cache misses and invalidation, but also records the most recent values of data objects in the simulated caches, which allows the user to calculate data inconsistent rate after a crash happens. To calculate the data inconsistent rate for a data object, NVCT counts the total number of dirty bytes in the data object and then divides the number by the data object size.

An example. Figure 1 gives an example of how we study application recomputability using NVCT. This is a multi-grid (MG) numerical kernel from the NAS benchmark suite [40] (NPB). Like many HPC applications, MG has a main computation loop, within which we persist two global data objects and a loop iterator [2] (Lines 14-15 and 17 in Figure 1a). After a crash happens, we restart MG using Figure 1b. To restart, the application re-initializes computation (Line 5 in Figure 1b), loads the values of the two data objects and old loop iterator (Lines 6-8) from NVM, and restarts the main computation loop from the iteration where the crash happens (Line 9). We run MG to completion and verify the application correctness.

[2]In the rest of the paper, we always persist a loop iterator to bookmark where the crash happens. This makes restart easier. Persisting just one iterator has almost zero impact on application performance.

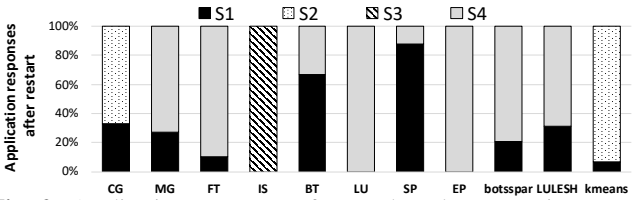


Fig. 2: Application responses after crash and restart. Figure annotation: S1 - successful recomputation without using extra iterations, S2 - successful recomputation with extra iterations, S3 - Interruption, and S4 - verification fails.

III. Characterization of Application Recomputability

A. Experiment Setup

Benchmarks for evaluation. We use all benchmarks from NPB. To enrich our benchmark collection, we add botsspar from SPEC OMP 2012 [41], kmeans from Rodinia [42] and LULESH [43]. These benchmarks are chosen, because of their representativeness and explicit code structures to verify application correctness. Table I summarizes them.

System configuration. We simulate a three-level cache (L1 cache: 32KB and 8-way; L2 cache: 1MB and 12-way; L3 cache: 19.25MB and 11-way), 64B cache line, write-back, write-allocation and LRU policy. We use both single and 4 threads to run benchmarks, and present the results of 4 threads.

Crash tests. To ensure statistical significance, for each benchmark, we run a sufficient number of crash and recomputation tests (usually 1000-2000 tests), such that further increasing tests does not cause big variation (less than 5%) in evaluation results. This method ensures that our evaluation is sufficient and results are statistically correct. During application execution, we randomly stop it for crash tests, and the time of stopping follows a uniform distribution. This method is common in the fault tolerance research [21], [30].

B. Experiment Results

We observe four application responses after a crash and restart. (1) Successful recomputation without performance overhead: the application successfully passes acceptance verification, and uses no extra iteration to finish; (2) Successful recomputation with performance overhead: the application successfully passes the acceptance verification, but takes at least one more iterations to finish; (3) Interruption: the application cannot run to completion; (4) Verification fails: the application cannot pass the acceptance verification, even after taking two times as many iterations as in the original execution.

Figure 2 and Table I show the results. We notice that some applications show strong recomputability (e.g., 88% and 67% for SP and BT respectively). Some (e.g., IS, LU, and EP) are the opposite: They cannot restart, or have segmentation faults.

Analysis. (1) SP and BT has high recomputability because they can isolate propagation of data inconsistency. In particular, SP and BT, aiming to solve 3-dimensional compressible Navier-Stokes equations [44], decouple computation along x, y and z dimensions. Each dimension employs an iterative numerical solver which tolerates data loss after crashes [45], and most of data inaccuracy is constrained to one dimension without propagation, because of the decoupling of dimensions. (2) LU has low recomputability, because it does not decouple computation along 3 dimensions. Although LU performs similar numerical simulation as SP and BT, data inaccuracy is propagated throughout the whole computation and fails the verification eventually. (3) IS has low recomputability, because it is characterized with rich pointer arithmetic. Crash and restart easily cause segfaults. (4) EP has low recomputability, because it has a small memory footprint. Crash and restart leave most of data objects in stale states, violating application requirement on data correctness.

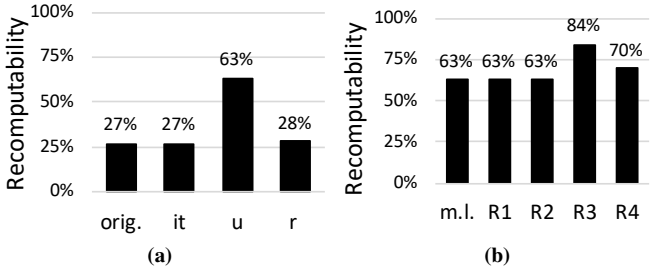


Fig. 3: The recomputability of MG after (a) persisting three different data objects in MG; and (b) persisting u in different code regions. "orig." stands for the recomputability without persisting anything. "m.l." stands for the case of persisting u at the end of each iteration of the main loop without considering code regions.

Conclusion 1. Applications have quite different recomputability, because of code structures (e.g., in IS and EP) and algorithms (e.g., in SP and LU).

To study how to improve application recomputability, we selectively persist data objects and examine its impact on application recomputability. We do not persist all data objects, because it can cause large performance overhead. Figure 3a shows the results for MG. We choose three data objects (it, u and r) for study. We persist them at the end of each iteration of the main computation loop. By persisting u, the recomputability is improved to 63%; However, persisting it and r, the recomputability is barely improved.

Analysis. (1) it is a single variable (4 bytes) and used only once in each iteration of the main loop. Without flushing it, it is highly likely that it is evicted out of the cache at the end of the iteration because of cache conflicts. Hence, persisting it is not helpful to improve recomputability. (2) u and r are large arrays; a set of stencils are applied to them over many iterations. r are dominated by read, while u are by intensive writes. Hence, persisting r is not helpful while persisting u is.

Conclusion 2. Persisting different data objects has different implications on application recomputability, because of data access patterns (e.g., data reuse and write/read pattern).

We further study the impact of where to persist data objects on application recomputability. MG has four first-level inner loops shown as R1-R4 in Figure 1a. They represent four execution phases. They all update u. We persist u at the end of an execution phase (code region). Figure 3b shows the result. Persisting u at R3, we have 21% improvement in recomputability, while persisting it at other code regions, we only have less than 7% improvement.

Analysis. MG is a hierarchical multi-grid method that approximates the solution to a discrete Poisson equation. R3 is the solving phase on a coarse grid to accelerate computation convergence [21]. Data inconsistency in R3 easily causes significant computation errors in multiple finer grids, leading to verification failure. Persisting u in R3 constrains data inconsistency caused by crashes, leading to higher recomputability. Other code regions work on a finer grid where the impact of data inconsistency on outcome correctness is limited. Hence persisting u at other code regions is not helpful.

TABLE I: Benchmark information. "DO" = "data object", "iter" = "iterations".

Benchmarks	Description	# of code regions	input	Memory footprint	Candi. of critical DO size	Critical DO size	Ave. # of extra iter. to restart (restart overhead)	Total # of iter. in the original app execution
CG	Sparse linear algebra	6	CLASS C	947MB	5.7MB	2.3MB	9.1	75
MG	Structured grids	4	CLASS C	3.4GB	2.3GB	1.2GB	0	20
FT	Spectral method	4	CLASS C	5.1GB	4.0GB	4.0GB	0	20
IS	Graph traversal (sorting)	8	CLASS C	1.0GB	264MB	4KB	N/A(segfault)	10
BT	Dense linear algebra	15	CLASS C	1.43GB	525.6MB	361.2MB	0	200
LU	Dense linear algebra	4	CLASS C	1.4GB	599MB	164MB	N/A (the verification fails)	250
SP	Dense linear algebra	16	CLASS C	1.47GB	561MB	394MB	0	400
EP	Monte Carlo	2	CLASS C	1MB	1MB	80B	N/A (the verification fails)	65535
botsspar	Sparse linear algebra	4	m=120, n=501 (ref)	3.74GB	3.36GB	3.36GB	0	200
LELUSH	Hydrodynamics modeling	4	s=100	1.41GB	251MB	20MB	0	3517
kmean	Data mining	1	100000_34.txt	222MB	20B	20B	18.2	36

Conclusion 3. The application shows different recomputability when persisting data objects at different code regions, because the execution correctness of those code regions has different impact on application outcome correctness.

Insight. Persisting all data objects throughout code regions may not be useful and efficient. Selectively persisting data objects at some code regions can effectively bound data errors caused by data inconsistency and lead to higher application recomputability, while paying less performance overhead.

IV. Design

Motivated by the above observations, we introduce EasyCrash, a framework that can increase application recomputability with an ignorable runtime overhead and offers higher system efficiency than C/R without EasyCrash. EasyCrash *automatically decides* which data objects should be persisted (Section IV-A) and where to persist them to maximize application recomputability (Section IV-B).

A. Selection of Data Objects

We name data objects selected to be persisted, *critical data objects* in the rest of the paper. To select data objects, we choose those data objects with the following properties as *candidates*: (1) Their lifetime is the main computation loop; and (2) They are not read-only. Except the candidates, the other data objects are either temporal or read-only, and not treated as the candidates. When the application restarts, the other data objects are not read from NVM. Instead, they are restored by either the initialization phase of the application or being re-computed based on the candidates of critical data objects. When the application restarts, the candidates are directly read from NVM. There is a large search space to select data objects out of the candidates: There could be hundreds or thousands of candidates in an HPC application. We use statistical correlation analysis to efficiently select data objects.

Our method is based on the following observation. When a crash happens, data objects in NVM can have different degrees of inconsistency. For example, a data object of 128MB could have 16MB of inconsistent data, giving an inconsistent rate of $16/128 = 12.5\%$, while some data object could have an inconsistent rate of 50%. Application recomputability correlates with the inconsistent rate of some data objects, meaning that if these data objects have high inconsistent rate, application recomputability is low. They should be selected as critical data objects. Application recomputability is not sensitive to the inconsistent rate of some data objects. Persisting them does not matter to application recomputability. Hence, the sensitivity of application recomputability to the inconsistent rate of data objects can work as a metric to select data objects.

We use Spearman's rank correlation analysis [46] to statistically quantify the correlation between the inconsistent rate of data objects and application recomputability. The analysis result is an coefficient (R_s), which quantifies how well the relationship between two input vectors can be described using a monotonic function. Furthermore, we use the p-value of R_s to ensure statistical significance of our analysis. The p-value is the probability of observing data that would show the same correlation coefficient in the absence of any real relationship between the input vectors.

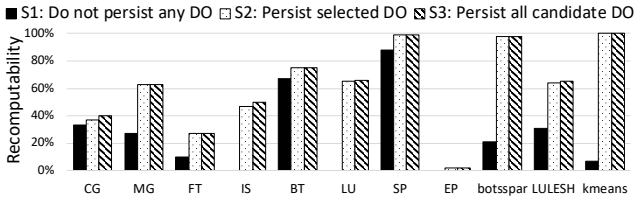


Fig. 4: Application recomputability under three strategies to persist data objects. Figure annotation: "DO" stands for data objects.

To use the Spearman's rank correlation analysis, we build two vectors for each candidate data object, using the results from a crash test campaign: One vector is composed of data inconsistent rates; The other is composed of Boolean values (i.e., whether the application recomputes successfully or not). Each component of the two vectors is collected in one crash test. The vectors are used as input to the correlation analysis.

Based on the Spearman's rank correlation analysis, we use two criteria to select data objects. (1) A critical data object should have a negative value of R_s, which indicates decreasing data inconsistent rate improves application recomputability. (2) The p-value of R_s should be smaller than a threshold. We use 0.01 as the threshold, because it is common [46] and less than it statistically shows a very strong correlation in our study.

Verification of the selection of data objects. We evaluate application recomputability with three strategies: (1) Do not persist any data object; (2) Persist selected data objects; (3) Persist all candidate data objects. Figure 4 shows the results. The figure shows that the difference in application recomputability between (2) and (3) is less than 3% in all cases. This verifies the effectiveness of our selection method.

B. Selection of Code Regions

In this section, we first introduce code regions in typical HPC applications. Then we formalize our problem of selecting code regions, and introduce an algorithm to solve it.

Application code regions. We characterize HPC applications as a set of iterative structures or loops. In particular, there is usually a main computation loop in an HPC application. Within the main loop, there are a number of inner loops typically used to update data objects iteratively. This code structure is commonly used in HPC applications. A number of existing efforts are based on this code structure [15], [30], [31], [47], [48]. Figure 1a shows an example from MG.

We model an application as a chain of code regions delineated by loop structures. A code region is either a first-level inner loop or a block of code between two adjacent, first-level inner loops. We use the above definition of code regions, because it easily represents computation phases. Persisting

data objects in a code region ensures that the most recent computation results in a phase are persistent in NVM, and can effectively improve application recomputability. A similar definition of code regions is in [30].

Problem formulation. Among all code regions, we want to select code regions to satisfy two performance goals. (1) *The runtime performance goal*: the application with critical data objects persisted at the selected code regions should have runtime overhead smaller than a threshold t_s. t_s is set by the user (in our study, $t_s = 3\%$ of application execution time without any crash). (2) *The system efficiency goal* for long-running applications: the system efficiency with EasyCrash (including successful and unsuccessful recomputation) should be better than that with traditional C/R without EasyCrash. Achieving this goal requires that the recomputability of the application should be high enough (higher than a threshold τ). Section VI discusses how to decide τ.

We name the selected code regions "critical code regions" in the rest of the paper. In the following discussion, we assume that there is only one critical code region, in order to make our formalization easy to understand. But our formalization can be easily extended to any number of critical code regions.

Assume that there are W code regions in an application and Y is the application recomputability without persisting any data objects. The recomputability of a code region i is c_i. **The recomputability of a code region** is the possibility that when a crash happens during the execution of the code region, the application is successfully recomputed. Based on the definition of recomputability (Section II-B), the application recomputability Y is a weighted sum of the recomputability of code regions; A weight for a code region is the ratio of execution time of the code region to total execution time of the application. We formulate Y based on the above discussion.

$$Y = \sum_{i=1}^{W}(a_i \times c_i) \tag{1}$$

where a_i is the weight of a code region i. In other words, a_i is the ratio of the accumulated execution time [3] of the code region i to total execution time of the application.

Assume the code region k ($1 \leq k \leq W$) is selected as a critical code region. After persisting critical data objects at k, the recomputability of the application and code region becomes Y' and c'_k respectively. We have performance loss l_k because of persisting critical data objects in k. l_k is the ratio of absolute performance loss to total execution time.

Y' is calculated based on c'_k for the code region k. c_i for the other code regions ($1 \leq i \leq W$ and $i \neq k$) remains the same. Y' is formulated in Equation 2.

$$Y' = \sum_{i=1}^{k-1}(a'_i \times c_i) + a'_k \times c'_k + \sum_{i=k+1}^{W}(a'_i \times c_i) \tag{2}$$

where a'_i and a'_k are new performance ratios (weights) with the consideration of the persistence overhead.

Our two performance goals are formulated as follows. We want to select a code region to meet the two goals.

$$l_k < t_s \tag{3}$$

$$Y' > \tau \tag{4}$$

Our algorithm to solve the problem. To determine if the selection of a code region meets Equation 3, we need to estimate the performance loss (l_k) caused by persisting critical data objects. Based on l_k, we easily get a'_i (the weight). l_k is estimated by measuring the overhead of flushing one cache block and the total number of cache blocks to flush. To determine if the selection of a code region k meets Equation 4, we first estimate c'_k without doing extensive crash tests (recall that c'_k is the recomputability of the code region k after persisting data objects). Then, based on Equation 2 and c'_k, we calculate Y' (recall that Y' is the application recomputability after persisting data objects at the code region k) and use Equation 4 to decide if we reach the system efficiency goal.

c'_k depends on how frequently we persist data objects in the code region. (1) If the code region k is a loop structure, we can persist data objects at every iteration of the major loop to maximize recomputability (c_k^{max}, and $c'_k = c_k^{max}$), or persist them every x iterations ($x > 1$) (the corresponding recomputability of the code region is c_k^x, and $c'_k = c_k^x$). If we do not persist data objects at all, then the recomputability of the code region is not changed (still c_k), and the code region is not selected. (2) If the code region is not a loop structure, we flush cache blocks at the end of the code region to reach c_k^{max}, or do not flush at all with no change of recomputability.

To measure c_k^{max} for a code region k, we persist data objects at every iteration of the major loop in k [4] to maximize recomputability of the code region. Then we trigger crashes during the execution of the code region k, and then measure the application recomputability as c_k^{max}. However, given W code regions to measure recomputability, this approach has to perform W crash test campaigns, which can be expensive. We use the following method to address this problem.

We use only one crash test campaign to measure best recomputability of all code regions (including the region k). In particular, we persist data objects at each iteration of major loop in *each* code region. This ensures best recomputability of each code region. In the crash test campaign, crash tests still randomly happen. We use those crash tests that trigger crashes during the execution of a code region to calculate the best recomputability of that code region.

To calculate c_k^x (recall that c_k^x is the recomputability of the code region k when we persist data objects every x iterations of major loop in the code region k), we use Equation 5.

$$c_k^x = (c_k^{max} - c_k) \times \frac{1}{x} + c_k \tag{5}$$

In essence, Equation 5 estimates c_k^x based on a linear interpolation between c_k^{max} and c_k (recall that c_k is the recomputability of the code region k without persisting any data object).

Using the above formulation, we are able to know the application recomputability (c'_k) for any code region. To illustrate the above modeling process, Figure 5 runs an example where we have three code regions, and the algorithm tries to decide if the code region 2 should be selected and how frequently to persist data objects there.

We can generalize our method to select any number of code regions (not just one as above) and decide how frequently to persist data objects in each code region. In particular, to meet the two performance goals, we choose those code regions in which persisting data objects with certain frequencies do not cause performance loss larger than t_s. Also, application recomputability after persisting critical data objects in the selected code regions with selected cache flushing frequencies

[3] A code region can be repeatedly executed. Hence we count the accumulated execution time.

[4] If there is no loop, we persist data at the end of the code region k.

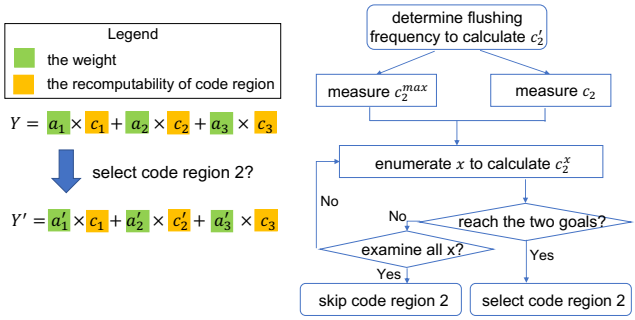


Fig. 5: An example of using our algorithm to decide whether a code region (code region 2) in a program should be selected. The program has three code regions.

is larger than τ (meeting the system efficiency goal). We also want to maximize application recomputability. This is a variant of the 0-1 knapsack problem [49] with each code region as an item, performance loss as the item weight, and application recomputability as the item value. This problem can be solved by the dynamic programming in pseudo-polynomial time.

Discussions. When estimating l_k, we assume every cache block of data objects is in the cache, which overestimates cache flushing overhead. However, overestimation is harmless, because it ensures that runtime overhead is smaller than t_s.

To calculate Y', we use one campaign of crash tests to measure the recomputability of *each* code region, by persisting critical data objects at *each* code region. However, to accurately measure the recomputability of a specific code region, we should persist critical data objects only at that code region (not each code region). Our method, although avoids massive crash tests, ignores the propagation of computation inaccuracy from one code region to another, which makes the measured recomputability smaller than the real recomputability. This means EasyCrash should result in larger recomputability and larger performance benefit in reality, which is good.

C. How to Use EasyCrash

To use EasyCrash, we need to know the performance loss l_k for each code region. Different frequencies of persisting data objects lead to different performance losses. We estimate the performance loss based on the overhead measurement of flushing one cache block, total number of cache blocks and flushing frequency. Note that certain cache flushing instructions (`CLFLUSH` and `CLFLUSHOPT`) invalidate cache lines after cache flushing. This means that cache blocks need to be reloaded into the cache when they are re-accessed, which causes extra performance loss. To account for such cases, we double our estimation on the overhead of flushing cache blocks. **The whole workflow of EasyCrash includes 4 steps**.

Step 1: Running a crash test campaign without persisting data objects. We collect the data inconsistent rate of candidates of critical data objects and calculate corresponding application recomputability. We also measure the recomputability of each code region (i.e., $c_k, 1 \leq k \leq W$) in this test campaign.

Step 2: Selection of data objects. We calculate the correlation between the inconsistent rate of data objects and application recomputability to decide critical data objects.

Step 3: Selection of code regions. We run another crash test campaign that persists critical data objects at each code region with highest frequency to measure the best recomputability of each code region ($c_k^{max}, 1 \leq k \leq W$). The output of this step is the selection of code regions and how frequently to persist data objects in the selected code regions.

Step 4: Production run. Just run the application, and EasyCrash automatically manages cache flushes.

Application preparation. The above steps introduce minor changes to the application. The application changes include two parts: (1) Allocating data objects that are updated in the main computation loop with an EasyCrash API. Those data objects are candidates of critical data objects, and their addresses are passed into EasyCrash for potential cache flushing during production runs. (2) Identifying the end of first-level inner loops with an EasyCrash API. Those places delineate code regions. We describe the APIs and show an example on how to use them in [25] and [50]. For (1) and (2), the compiler can annotate the application with the APIs, freeing the programmer from changing the application.

Ease of using EasyCrash. EasyCrash includes tools [50] to automate the selection of data objects and code regions and crash tests. Without crash tests, it usually takes a few tens of seconds; With crash tests, it only takes about 10 minutes or so in our evaluation, with a technique in Section VII.

V. Evaluation

We study whether EasyCrash can effectively improve application recomputability and what is the runtime overhead of EasyCrash. In Section VI, we evaluate system efficiency in large scale systems in the context of C/R mechanisms. We use benchmarks in Table I. To calculate application recomputability, we use the method in Section III-A for crash tests.

We use a platform with Optane DCPMM (1.5TB) and two Xeon 8260L processors. We set t_s as 3% in this section. We also use $t_s = 2\%$ and 5% for the *sensitivity study*. In all tests, the runtime overhead is effectively bounded by t_s. But a smaller t_s leads to less frequent persistence operations. As a result, a few benchmarks (e.g., FT) cannot meet the recomputation requirement imposed by τ. We show the results of $t_s = 3\%$ in this section. We do not present EP, because its inherent recomputability is 0. Even with EasyCrash, its recomputability is less than 3%, and EasyCrash cannot bring benefit in system efficiency according to our model (Equation 4).

Recomputability improvement. Figure 6 shows application recomputability after using EasyCrash. To reveal the effectiveness of selecting data objects and code regions, we first measure recomputability without using them, shown as "without EasyCrash" in the figure. Then we select data objects and persist them at the end of each iteration of the main computation loop, shown as "selecting data objects". We then select code regions to persist the selected data objects with selected frequencies, shown as "selecting code regions".

To show EasyCrash effectiveness, we also compare the *best recomputability* with the recomputability after using EasyCrash. The best recomputability is obtained by persisting critical data objects at each code region (if the code region has a loop structure, we persist critical data objects with the highest frequency, i.e., persisting them at the end of each iteration of the loop). Note that the method to get the best recomputability is very costly (see Table II), which is not a practical solution. In addition, we do not show results of persisting all data objects, because Section IV shows that persisting critical data objects can achieve very similar recomputability as persisting all data objects. Figure 6 leads to two observations.

(1) EasyCrash achieves high recomputability. Except for CG, the recomputability after applying EasyCrash is close to

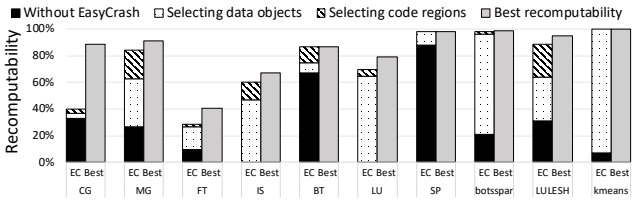


Fig. 6: Application recomputability with different techniques.

TABLE II: Normalized execution time. "Norm" = "normalized". "EC" = "EasyCrash". "best" = "the best recomputability"

	Time for persisting critical data for once	# of persistence operations	Norm. exe. time with EC	Norm. exe. time without EC	Norm. exe. time achieving the best.
CG	<0.001 s	75	1.004	1.20	1.24
MG	0.045 s	40	1.018	1.37	1.31
FT	0.043 s	80	1.023	1.42	1.36
IS	0.041 s	10	1.018	1.31	1.74
BT	0.042 s	100	1.018	1.31	1.63
SP	0.041 s	100	1.021	1.41	1.77
LU	0.049 s	125	1.021	1.42	1.80
botsspar	0.041 s	200	1.029	1.58	1.77
LULESH	0.039 s	293	1.027	1.54	1.73
kmeans	<0.001 s	36	1.000	1.00	1.00
Average	≈ 0.034 s	106	1.018	1.26	1.54

the best one, with a difference of only 5% on average. For CG, there is a big difference (49%), because many successful recomputation tests require extra iterations, which is not acceptable in EasyCrash due to the concerns on performance loss. Note that even with the difference, EasyCrash still brings 4% improvement in system efficiency for CG (Section VI).

(2) EasyCrash significantly improves application recomputability. This fact is especially pronounced in MG, botsspar and kmeans. We see 56%, 77%, and 93% improvement for them respectively. The average recomputability of all benchmarks after using EasyCrash is 75%, while it is 28% before using EasyCrash. EasyCrash is able to transform 47% of crashes that cannot correctly recompute into correct computation.

Performance study. We measure runtime overhead of persisting critical data objects at critical code regions with EasyCrash but with no crash triggered. We leverage `CLWB` for best performance of cache flushing. Table II summarizes execution time of persisting critical data objects for once (i.e., performing *one persistence operation*), the number of persistence operations with EasyCrash, and total execution time with persistence operations. *In the rest of this section, the total execution time is normalized by the execution time without any persistence operation.*

In general, the runtime overhead is no larger than 2.9% (bounded by $t_s = 3\%$). For comparison purpose, we show the overhead of persisting all candidate data objects at the end of each iteration of main computation loop (shown in the fifth column of Table II), which is a case without the selection of data objects and code regions. This case causes 26% overhead on average, much larger than EasyCrash. We also evaluate the overhead of achieving the best recomputability by persisting critical data objects with the highest frequency. The runtime overhead is 54% on average, much larger than EasyCrash.

Write Reduction. We compare EasyCrash and in-memory C/R mechanism in terms of number of extra writes. For EasyCrash, the extra writes come from persisting critical data objects at critical code regions. As discussed in Section II-A, when cache blocks of critical data objects are clean or not resident in the cache, flushing them does not cause any write in NVM. For C/R mechanism, the extra writes come from (1) making a copy of data objects and (2) cache line eviction because of loading checkpoint data into the cache when making data copy [15]. We use NVCT to measure the number of writes in NVM. Whenever a dirty cache block is written back to NVM, we count the number of writes by one.

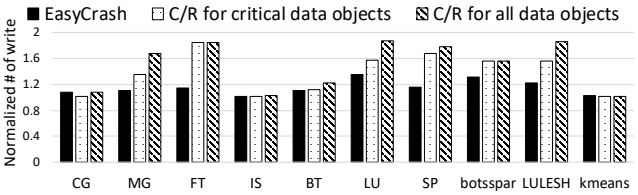


Fig. 7: Normalized number of NVM write.

To enable a fair comparison with EasyCrash, we perform C/R in two ways: (1) We checkpoint critical data objects, and (2) we checkpoint all data objects (excluding read-only ones). We assume that checkpoint happens only once. This is a conservative assumption favoring the checkpointing mechanism. The checkpoint could happen more often (depending on system failure rate and application execution time), causing more writes. We consider system failure rate and application execution time to evaluate checkpoint effects in Section VI.

Figure 7 shows the number of write normalized by total numbers of writes in NVM without EasyCrash and C/R. On average, EasyCrash adds 16% additional writes, while C/R adds 38% and 50% using the two checkpointing methods respectively. Also, for those benchmarks with large data objects (e.g., FT, SP and LU), EasyCrash is especially beneficial since the number of extra writes in a persistence operation is bounded by the last level cache size. A larger data object indicates that EasyCrash flushes more non-resident cache blocks or clean cache lines without causing actual writes. For benchmarks with small data objects (e.g., CG with data objects smaller than or similar to the last level cache size), EasyCrash is not beneficial to reduce writes, but writing those small data objects does not usually cause serious endurance problems.

VI. End-to-End Evaluation

We evaluate EasyCrash in the context of large-scale systems running time-consuming HPC applications with a C/R mechanism. To enable convincing evaluation, we need different system scales with various configurations, which is expensive to achieve. We develop an emulator based on performance models and Section V.

Basic assumptions. We assume that the checkpointing process does not have any corruption. This is a common assumption [22], [51]. We model coordinated checkpointing, which is the most common C/R commonly used in the recent work [22], [52], [53] (we model uncoordinated checkpionting in [25] for the completeness of our study. In general, EasyCrash improves system efficiency by 1% to 60% for uncoordinated checkpointing). With the coordinated checkpointing, all nodes take checkpoints at the same time with synchronization. The checkpoints are saved in *fast local storage and then asynchronously* moved to remote storage nodes. When a crash happens in one node and the application cannot successfully run to the completion or pass the acceptance verification after restarting using EasyCrash, all nodes will go back to the last checkpoint. Note that with EasyCrash, the application has a high probability to successfully recompute after restart. Hence, the checkpoint interval with EasyCrash is longer.

Performance modeling. Our emulator includes system and application related parameters. We summarize the *system related parameters* as follows.

1) ***MTBF***: Mean time between failures of the system without EasyCrash. $MTBF_{EasyCrash}$ is ***MTBF*** with EasyCrash.

Since the average application recomputability with EasyCrash is 77% (Section V), we have $MTBF_{EasyCrash} = MTBF/(1-77\%)$.

2) ***T_chk***: The time for writing a system checkpoint. The checkpoint on each node is written into local SSD (not in NVM main memory) and then gradually migrated to storage nodes (the data migration overhead is not in *T_chk*). This multi-level checkpoint mechanism is based on [52]. The checkpoint should not be written to main memory, because it greatly reduces memory space for applications.
3) ***T_r***: The time for recovering from the previous checkpoint. Similar to the existing work [51], we assume *T_r* = *T_chk*.
4) ***T_sync***: The time for synchronization across nodes. We use the assumption in [22]: The synchronization overhead is a constant value, and we use 50% of the checkpoint overhead as *T_sync*.
5) ***T***: The checkpoint interval, based on Young's formula [54], $T = \sqrt{2 \times T\_chk \times MTBF}$. This formula has been shown to achieve almost identical performance as in realistic scenarios [55].
6) ***T_vain***: The wasted computation time. When the application rolls back to the last checkpoint, the computation already performed in the checkpoint interval is lost. As proved by Daly [56], on average, half of a checkpoint interval for computation is wasted (i.e., $T\_vain = 50\% \times T$).

We summarize the *application related parameters*.

1) $\boldsymbol{R_{EasyCrash}}$: The application recomputatbility with EasyCrash.
2) $\boldsymbol{t_s}$: The runtime overhead introduced by EasyCrash because of persisting critical data objects (e.g., 3% in Section V).

Based on the above notations, we use performance models to evaluate system efficiency. The system efficiency is the ratio of the accumulated useful computation time (u) to total time spent on the system ($Total\_Time$), which is ($u/Total\_Time$). We assume that the accumulated useful computation takes checkpoints N times; and during the whole computation, the crash happens M times.

Equation 6 models the total time without using EasyCrash. The equation includes useful computation and checkpoint time ($N \times (T + T\_chk)$), and the cost of recovery using the last checkpoint ($M \times (T\_vain + T\_r + T\_sync)$). The number of crashes (M) is estimated using Equation 7.

$$Total\_Time = N \times (T + T\_chk) + M \times (T\_vain + T\_r + T\_sync) \quad (6)$$

$$M = \frac{Total\_Time}{MTBF} \quad (7)$$

EasyCrash improves system efficiency by avoiding recovery from the last checkpoint and increasing the checkpoint interval. Equation 8 models the total execution time with EasyCrash, where N' and T' are the number of checkpoints and their interval when using EasyCrash, and M' is the number of crashes that use the last checkpoint for recovery, and M'' is the number of crashes that use EasyCrash to recompute successfully.

$$\begin{aligned} Total\_Time = {} & N' \times (T' + T\_chk) + \\ & M' \times (T\_vain' + T\_r + T\_sync) + \\ & M'' \times (T\_r' + T\_sync) \end{aligned} \quad (8)$$

$$M' = M \times (1 - R_{EasyCrash}), \qquad M'' = M \times R_{EasyCrash} \quad (9)$$

With EasyCrash, the checkpoint interval (T') becomes longer ($T' > T$), and also should include a small runtime overhead (t_s). As a result, the number of checkpoints (N') becomes smaller ($N' < N$), and the checkpoint overhead ($N' \times T_{chk}$) becomes smaller. With and without EasyCrash, the useful computation remains similar because of small runtime overhead of EasyCrash. To calculate T', we use Young's formula, $T' = \sqrt{2 \times T\_chk \times MTBF_{EasyCrash}}$.

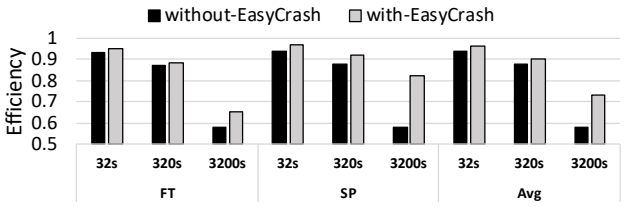
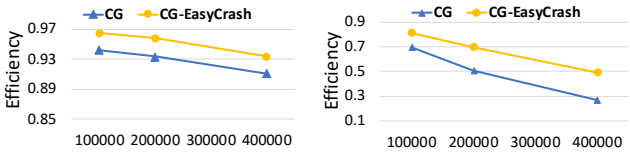


Fig. 8: System efficiency without and with EasyCrash when the system MTBF is 12 hours. The x-axis shows different checkpointing overhead. "Avg" stands for "average".



(a) $T\_chk = 32s$. **(b)** $T\_chk = 3200s$.

Fig. 9: System efficiency for CG without and with EasyCrash when the system scales from 100,000 to 200,000 and 400,000 nodes

With EasyCrash, once a crash happens, the system either goes to last checkpoint with recovery overhead modeled as $M' \times (T\_vain + T\_r + T\_sync)$, or uses EasyCrash to successfully recompute with recovery overhead modeled as $M'' \times (T\_r' + T\_sync)$. With NVM, T_r becomes T_r', which is smaller, because we load data from NVM-based main memory, not from local SSD or storage node. T_r' is estimated using data size of non-readonly data objects divided by NVM bandwidth.

Choice of parameters. The time spent on writing a checkpoint to persistent storage depends on hardware characteristics. A modern HPC node normally has 64 to 128 GB memory. For nodes using SSD and NVMe, the average I/O bandwidth is 2 GB/s; For nodes using HDD, the average I/O bandwidth is around 20 MB/s to 200 MB/s [57], [58]. As a result, we choose the checkpointing overhead (T_chk) as 32s, 320s, 3200s to represent different hardware scenarios. A similar set of values is used in previous efforts [22], [51], [55]. We emulate the system with 100,000 nodes for a long simulation time ($Total\_Time$ is 10 years). Previous work [59] shows that systems in such a scale have $MTBF$ = 12 hours. Based on this data, we scale $MTBF$ as in [22] for 200,000 and 400,000 nodes. As a result, $MTBF$ are 6 and 3 hours respectively.

Results for system efficiency. Figure 8 shows system efficiency with and without EasyCrash under different checkpoint overhead. We show the lowest and highest recomputability (from FT and SP respectively), and average recomputability of *all benchmarks* because of space limitation. EasyCrash improves system efficiency by 2%-24%.

We evaluate the system scalability with EasyCrash. We evaluate all benchmarks but only present CG because of space limitation. Results for other benchmarks can be found in [25]. Figure 9 shows the efficiency with and without EasyCrash at different system scales. With EasyCrash the system efficiency always outperform that without EasyCrash. *This trend is consistent with all benchmarks*. The system with EasyCrash achieves better efficiency as the system scale increases.

Determining τ. To ensure the system with EasyCrash

has efficiency gains, the application recomputability must be higher than a threshold τ (see Section IV-B). Given $Total\_Time$ and Equations 8 and 9, we calculate a lower bound of $R_{EasyCrash}$, which is τ.

VII. Discussions

Determining how/when to use EasyCrash. To decide whether to use EasyCrash, we need multiple information, including (1) system MTBF, (2) checkpoint overhead, (3) the application recomputability with EasyCrash to select data objects and code regions and estimate efficiency benefit, and (4) the acceptable minimum performance loss t_s. For (1), (2) and (4), it is reasonable to assume that the system operator has such information. With (1), (2) and (4), the recomputability threshold τ can be calculated. For (3), we use crash tests (Section IV-C). For an application taking long execution time, repeatedly performing crash tests is time-consuming, but if the application is commonly used and repeatedly executed in production, then the cost of crash tests is amortized. For those applications that are time-consuming but not executed very often, we propose the following solution.

Our observation reveals that by using EasyCrash to persist critical data objects at selected code regions, the application using different input problems [5] shows similar recomputability. Our evaluation with ten benchmarks, each of which uses four input problems, shows that the variance of recomputability is less than 9% (detailed in [25]). Hence, we can use a small input problem to reduce evaluation cost. In our evaluation, using the smallest input problem for crash tests to estimate recomputability for the largest input problem (memory footprint sizes of the two input problems differ by 249x), we reduce test time by more than 99%. *In our evaluation, crash tests for each benchmark can be finished in less than 14 minutes using two 48-core machines (each has two Xeon Gold 6126 processors).* The rationale to support the above solution is that EasyCrash judiciously chooses critical data objects and code regions, hence effectively guarantees application recomputability.

To reduce evaluation cost, we cannot use an arbitrarily small input problem. Our empirical observation reveals that to enable accurate estimation of application recomputability for a large input problem, the size of all non-critical data objects in the application using a small input problem should be at least 2x larger than the last level cache size. This is because data accuracy loss for non-critical data objects is not bounded by EasyCrash when a crash happens; The application relies on hardware caching effect to persist them. If most of them can be fit into the cache and not persisted often, data inconsistent rate can be high and application recomputability can be reduced, which results in an under-estimation of application recomputability for the large input problem.

What kind of application is not suitable? Two categories of applications are not suitable for EasyCrash. (1) Applications with small memory footprint. When a crash happens, most of the application data are resident in the cache and lost. To ensure high recomputability, we have to persist data objects frequently, causing high runtime overhead. (2) Applications with no tolerance for computation errors. These applications regard any application outcome (or intermediate results) different from that of the golden run as incorrect. Many of our crash-and-restart tests generate outcomes (or intermediate results) different from those of the golden run, but these tests are correct execution and pass acceptance verification.

For (1), the system can disable EasyCrash and only employ the traditional checkpoint mechanism to handle failures. Because of small memory footprint of the application, the checkpoint is small and can be stored in NVM with small overhead. For (2), when the application outcome (or intermediate results) is different from that of the golden run, the users can claim a silent data corruption (SDC) happens [22], [30]. With the acceptance verification, many applications treat this kind of SDC as benign and ignorable. Examples of these applications include many iterative solvers and machine learning training workloads, which have been leveraged in the recent approximate computing research [33], [34]. The applications that cannot tolerate SDC cannot use EasyCrash.

[5]The application using different input problems should use the same algorithm and does not have control flow difference between input problems.

VIII. Related Work

Some efforts focus on establishing crash consistency in NVM [8], [14] by software- and hardware-based techniques. Building an atomic and durable transaction by undo- and redo-logging mechanisms in NVM is the most common method to enforce crash consistency [8], [16]. Some work on NVM-aware data structures [9], [14] re-design specific data structures to explicitly trigger cache flushing for crash consistency. However, the existing work can impose big performance overhead and extensive changes to the applications, which may not be acceptable by HPC.

Recent efforts use NVM for HPC fault tolerance [10], [15], [60]. They avoid flushing caches for high performance, and use algorithm knowledge [60] or have high requirements on loop structures [10], [15] to recover computation upon application failures. EasyCrash is different from them: EasyCrash aims to explore application's intrinsic error resilience and leverage consistent *and* inconsistent data objects for recomputation; EasyCrash is general, because it does not have high requirement on code structure or application algorithms.

Approximate computing trades computation accuracy for better performance by leveraging application intrinsic error resilience. LetGo [22] is such an example. Once a failure happens, LetGo continues application execution. EasyCrash is significantly different from LetGo. EasyCrash loses dirty data in caches when a crash happens, and selectively flushes data objects in some code regions to guarantee the improvement of system efficiency. Letgo does not lose data in caches and provides no guarantee on the improvement. LetGo does not consider differences of code regions and data objects in their impacts on application recomputability. EasyCrash is highly NVM oriented, while LetGo is not.

IX. Conclusions

The emergence of NVM provides many opportunities for HPC to enable scalable scientific simulation and high system efficiency. However, integrating NVM into HPC is challenging, because of high requirements of HPC on performance, resource consumption, and code maintenance. This paper focuses on leveraging the non-volatility of NVM for HPC under failures. We demonstrate the great potential of relaxing the requirement on crash consistency for high efficiency.

X. Acknowledgement

This work was partially supported by U.S.National Science Foundation (CNS-1617967, CCF-1553645 and CCF-1718194).

REFERENCES

[1] C. Hsu and W. Feng, "A power-aware run-time system for high-performance computing," in *SC'05*, 2005.

[2] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen, "A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems," *The Journal of Supercomputing*, 2013.

[3] Y. Liu, R. Nassar, C. Leangsuksun, N. Naksinehaboon, M. Paun, and S. Scott, "A reliability-aware approach for an optimal checkpoint/restart model in hpc environments," in *Cluster'07*, 2007.

[4] E. Meneses, X. Ni, T. Jones, and D. Maxwell, "Analyzing the interplay of failures and workload on a leadership-class supercomputer," 2015.

[5] N. DeBardeleben, J. Laros, J. Daly, S. Scott, C. Engelmann, and B. Harrod, "High-end computing resilience: Analysis of issues facing the hec community and path-forward for research and development," 2019.

[6] S. Gupta, T. Patel, C. Engelmann, and D. Tiwari, "Failures in large scale systems: Long-term measurement, analysis, and implications," in *SC '17*, 2017.

[7] "Intel and micron produce breakthrough memory technology," 2015.

[8] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight Persistent Memory," in *ASPLOS'11*, 2011.

[9] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "Nv-tree: Reducing consistency cost for nvm-based single level systems," in *FAST'15*, 2015.

[10] H. Elnawawy, M. Alshboul, J. Tuck, and Y. Solihin, "Efficient checkpointing of loop-based codes for non-volatile main memory," in *PACT'17*, 2017.

[11] K. Wu, Y. Huang, and D. Li, "Unimem: Runtime Data Management on Non-Volatile Memory-based Heterogeneous Main Memory," in *SC'17*, 2017.

[12] D. Li, J. S. 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 *IPDPS'12*, 2012.

[13] K. Wu, J. Ren, and D. Li, "Runtime data management on non-volatile memory-based heterogeneous memory for task-parallel programs," in *SC'18*, 2018.

[14] 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 *ASPLOS'11*, 2011.

[15] M. Alshboul, J. Tuck, and Y. Solihin, "Lazy persistency: A high-performing and write-efficient software persistency technique," in *ISCA'18*, 2018.

[16] Intel, " Persistent Memory Development Kit," https://pmem.io/, 2014.

[17] M. Dong and H. Chen, "Soft updates made simple and fast on non-volatile memory," in *ATC'17*, 2017.

[18] J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann, "Combining partial redundancy and checkpointing for hpc," in *ICDCS'12*, 2012.

[19] I. R. Philp, "Software failures and the road to a petaflop machine," in *HPCRI'05*, 2005.

[20] 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 *PHYSOR'14*, 2014.

[21] M. Casas, B. R. de Supinski, G. Bronevetsky, and M. Schulz, "Fault Resilience of the Algebraic Multi-grid Solver," in *ICS'12*, 2012.

[22] B.Fang, Q.Guan, N.Debardeleben, K.Pattabiraman, and M.Ripeanu, "Letgo: A lightweight continuous framework for hpc applications under failures," in *HPDC'17*, 2017.

[23] L. Guo and D. Li, "MOARD: Modeling Application Resilience to Transient Faults on Data Objects," in *IPDPS'19*, 2019.

[24] P. J. Roache, *Verification and validation in computational science and engineering*. Hermosa, 1998.

[25] "Easycrash: Exploring non-volatility of non-volatile memory for high performance computing under failures (technical report)," in *Information is hidden due to double-blind reviews. Will release it after paper acceptance*, 2019.

[26] X. Ji, C. Wang, N. El-Sayed, X. Ma, Y. Kim, S. S. Vazhkudai, W. Xue, and D. Sanchez, "Understanding Object-level Memory Access Patterns Across the Spectrum," in *SC'17*, 2017.

[27] D. Li, J. S. Vetter, and W. Yu, "Classifying Soft Error Vulnerabilities in Extreme-Scale Scientific Applications Using a Binary Instrumentation Tool," in *SC'12*, 2012.

[28] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary, "HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers," 2008.

[29] D. Nicholaeff, N. Davis, D. Trujillo, and R. Robey, "Cell-based adaptive mesh refinement implemented with general purpose graphics processing units," *Tech. Rep. LA-UR-11-07127*, 2012.

[30] L. Guo, D. Li, I. Laguna, and M. Schulz, "FlipTracker: Understanding Natural Error Resilience in HPC Applications," in *SC'18*, 2018.

[31] G. Bronevetsky and B. de Supinski, "Soft Error Vulnerability of Iterative Linear Algebra Methods," in *ICS'08*, 2008.

[32] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and characterization of inherent application resilience for approximate computing," in *DAC'13*, 2013.

[33] J. Meng, A. Raghunathan, S. Chakradhar, and S. Byna, "Exploiting the forgiving nature of applications for scalable parallel execution," in *IPDPS'10*, 2010.

[34] M. Carbin, S. Misailovic, and M. C. Rinard, "Verifying Quantitative Reliability for Programs that Execute on Unreliable Hardware," in *OOPSLA'13*, 2013.

[35] W. Baek and T. M. Chilimb, "Green: a framework for supporting energy-conscious programming using controlled approximatio," in *PLDI'10*, 2010.

[36] A. Sampson, A. Baixo, B. Ransford, T. Moreau, J. Yip, L. Ceze, and M. Oskin, "ACCEPT: A Programmer-Guided Compiler Framework for Practical Approximate Computing," in *University of Washington Technical Report*, 2015.

[37] A. N. Lab, "U.S. Department of Energy and Intel to deliver first exascale supercomputer," https://www.anl.gov/article/us-department-of-energy-and-intel-to-deliver-first-exascale-supercomputer, 2019.

[38] Intel, "Intel NVM Library," http://pmem.io/nvml/libpmem/, 2014.

[39] J. Ren, K. Wu, and D. Li, "Understanding Application Recomputability Without Crash Consistency in Non-Volatile Memory," in *Proceedings of the Workshop on Memory Centric High Performance Computing*, 2018.

[40] D. H. Bailey, L. Dagum, E. Barszcz, and H. D. Simon, "NAS parallel benchmark results," *IEEE Parallel Distrib. Technol.*, vol. 1, no. 1, pp. 43–51, Feb. 1993.

[41] "SPEC OMP2012," www.spec.org/omp2012, 2012.

[42] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC'09*, 2009.

[43] LLNL, "LULESH 2.0," https://github.com/LLNL/LULESH, 2013.

[44] H. Jin, M. A. Frumkin, and J. M. Yan, "The openmp implementation of nas parallel benchmarks and its performance," 1999.

[45] I. Bermejo-Moreno, J. Bodart, J. Larsson, B. M. Barney, J. W. Nichols, and S. Jones, "Solving the compressible navier-stokes equations on up to 1.97 million cores and 4.1 trillion grid points," in *SC '13*, 2013.

[46] J. H. Zar, "Significance testing of the spearman rank correlation coefficient," *Journal of the American Statistical Association*, vol. 67, no. 339, pp. 578–580, 1972.

[47] M. Shantharam, S. Srinivasmurthy, and P. Raghavan, "Characterizing the Impact of Soft Errors on Iterative Methods in Scientific Computing," in *ICS'11*, 2011.

[48] D. Li, B. de Supinski, M. Schulz, D. S. Nikolopoulos, and K. W. Cameron, "Hybrid MPI/OpenMP Power-Aware Computing," in *IPDPS'10*, 2010.

[49] M. Silvano and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, 1990.

[50] (2019) Non-volatile memory crash tester (nvct). [Online]. Available: https://github.com/NVMCrashTester/NVCT

[51] G. Bosilca, A. Bouteiller, E. Brunet, F. Cappello, J. Dongarra, A. Guermouche, T. Herault, Y. Robert, F. Vivien, and D. Zaidouni, "Unified model for assessing checkpointing protocols at extreme-scale," *Concurr. Comput. : Pract. Exper.*, 2014.

[52] K. Mohror, A. Moody, G. Bronevetsky, and B. R. de Supinski, "Detailed Modeling and Evaluation of a Scalable Multilevel Checkpointing System," vol. 25, no. 9, pp. 2255–2263, 2014.

[53] A. Moody, G. Bronevetsky, K. Mohror, and B. de Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *SC'10*, 2010.

[54] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Commun. ACM*, 1974.

[55] N. El-Sayed and B. Schroeder, "Checkpoint/restart in practice: When 'simple is better'," in *IEEE International Conference on Cluster Computing*, 2014.

[56] J. T. Daly, "A Higher Order Estimate of the Optimum Checkpoint Interval for Restart Dumps," *Future Generation Computer Systems*, 2006.

[57] W. Bhimji, D. Bard, M.Romanus, A. Ovsyannikov, B. Friesen, M. Bryson, J. Correa, G. K. Lockwood, V. Tsulaia, S. Byna, S. Farrell, D. Gursoy, C. S. Daley, V. E. Beckner, B. van Straalen, N. J. Wright, and K. Antypas, "Accelerating science with the nersc burst buffer early user program," 2016.

[58] K. Wu, F. Ober, S. Hamlin, and D. Li, "Early evaluation of intel optane non-volatile memory with hpc i/o workloads," 2017.

[59] NCSA, "Blue Waters: Sustained Petascale Computing," 2014, http://www.ncsa.illinois.edu/BlueWaters/.

[60] S. Yang, K. Wu, Y. Qiao, D. Li, and J. Zhai, "Algorithm-directed crash consistence in non-volatile memory for hpc," in *Cluster'17*, 2017.