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

Anonymous Author(s)

## ABSTRACT

Hardware failures and faults often result in application crash in large-scale high performance computing (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 decide how to selectively persist application data objects to significantly increase possibility of successful recomputation. Our evaluation shows EasyCrash transforms 47% of crashes that cannot correctly recompute into the correct computation while incurring a negligible performance overhead (2.9% on average). Using EasyCrash and application intrinsic error resilience, 77% of crashes can successfully recompute. In combination with a traditional checkpoint on *storage*, EasyCrash enables up to 30% improvement (20% on average) in system efficiency at various system scales.

## 1 INTRODUCTION

The extreme-scale high performance computing (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 HPC system efficiency. A typical HPC system nowadays has a mean-time between failure (MTBF) of tens of hours [34, 48, 66, 71], 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 [30, 46].

Byte-addressable non-volatile memory (NVM) technologies, such as Intel Optane DC persistent memory DIMM [3], are emerging. NVM provides better density and power efficiency than DRAM while providing DRAM-comparable performance. Recent efforts have demonstrated the possibility of using NVM as main memory [37, 59, 78, 97, 103] with `load/store` instructions and for future HPC [40, 58, 64, 99, 101, 104]. 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 [25, 33, 59, 96, 97, 103] 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.

To enable crash consistency, the existing solutions use *in-memory* checkpoint/restart (C/R) [8, 26, 37, 58] or build crash-consistent data objects [23, 77, 90, 96, 103, 107]. 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-level techniques in place). For example, the write endurance of phase change memory (a promising NVM technology) is seven orders of magnitude lower than DRAM [81]. As a result, the endurance problem of NVM have been actively studied recently [6–8, 42, 45, 62, 107, 108]. 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 [49, 70, 97] or metadata [32, 82, 102], which often introduce new data structures and memory synchronization. Such disruptive modifications to the application are difficult to be adopted by legacy HPC applications, which are often large (tens or hundreds of thousands of code lines) and dominate scientific simulations in HPC data centers.

In conclusion, high requirements of HPC on memory consumption, performance, and code stableness are calling for a new solution to explore the 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

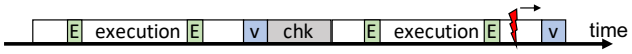



**Figure 1: An illustration of how an HPC application behaves with EasyCrash. Annotation: "E" - EasyCrash persistence operations; "v" - application acceptance verification; "chk" - checkpointing.**

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 tradition 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 [36, 80].

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 [95] 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. Furthermore, selectively persisting data objects at appropriate execution phases bounds data inconsistency and improves recomputability.

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 [85]. Those applications with acceptance verification reduces the probability of producing incorrect results that might be generated by applications.

Figure 1 illustrates how EasyCrash works with NVM as main memory for an HPC application. 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 we reduce 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.

Besides the contributions of handling application failures, our study is featured with a novel tool to characterize the sensitivity of application recomputation to crash consistency. In particular, to build EasyCrash, we must have a tool that retains data objects in main memory and caches after a crash for study, captures the impact of hardware caching on data persistence, and allows us to *repeatedly* trigger crashes. However, the traditional systems cannot meet our needs: The volatile DRAM-based system loses data in main memory and caches after a crash; The physical machines (including Intel Optane) cannot tolerate repeated crash tests (tens of thousands of tests); The existing tools to examine crash consistency [52–54, 61] ignore hardware caching effects (e.g., cache line eviction) on data persistence; The existing random fault injection method [19, 65, 98] provides no guarantee on application crash.

To address the above problem, we introduce a tool, *NVCT* (standing for *N*on-*V*olatile memory *C*rash *T*ester). NVCT is a PIN [68]-based cache simulator plus rich functionality for crash study. NVCT allows the user to trigger application crash randomly and then perform postmortem analysis on data values in caches and memory.

In summary, this paper makes the following contributions:

- A novel methodology for HPC under failures, by leveraging NVM and error resilience intrinsic to many HPC applications;
- An open-sourced tool [2] to enable crash study on NVM; Based on our knowledge, this tool is the first one for such study;
- Characterization of HPC application recomputability after crashes;
- Theoretical analysis to provide guidance for persisting data objects with guarantee on high performance and system efficiency;
- Evaluation with a spectrum of HPC applications. EasyCrash transforms 54% of crashes that cannot correctly recompute into correct computation, while incurring a negligible performance overhead (1.5% on average). Using EasyCrash 77% of crashes successfully recomputes. As a result, EasyCrash enables up to 30% improvement (20% on average) in system efficiency.

## 2 BACKGROUND

### 2.1 Cache Flushing for Data Persistence

Because of the prevalence of volatile caches, data objects in applications may not be persistent in NVM when a crash happens. To

[1] We distinguish cache line and cache block in the paper. The cache line is a location in the cache, and the cache block refers to the data that goes into a cache line.
[2] The tool is available online. https://github.com/NVMCrashTester/NVCT

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 [49], 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.

## 2.2 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 (Appendix A.1): We find that 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 [56, 65].

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 for NVM [25, 49, 97] focus on persisting data objects for the convenience of application 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 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). Leveraging application-level acceptance verification, we can determine the correctness of application outcome.

Furthermore, we define application recomputability with a high requirement on performance to make our solution practical for HPC. 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 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.** It is defined as the ratio of the accumulated useful computation time to total time spent on the HPC system. The total time includes useful computation time, checkpoint time, lost computation time because of crashes, and recovery time.

**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 [79] or during application execution [76], 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 very common in HPC applications, and increasingly common, because of the strong needs of increasing confidence in the results offered by HPC applications.

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 (see Appendix A.1). Many of those HPC applications are known to be naturally resilient to computation inaccuracy [19, 24]. They are promising to be recomputable after crashes, because they work by improving the accuracy of the solution 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 [18, 72, 73, 83, 84, 93]. We expect those applications become more common in the future, in order to enable higher performance and energy efficiency [10, 89].

**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 application recomputation. After application failure, NVM is still accessible for restart [9, 25, 27, 33, 59, 96, 97].

## 2.3 Optane DC Persistent Memory Module

The very recent release of Intel Optane DC persistent memory module (DCPMM) is arguably the most mature NVM product as *main memory* and promising for future HPC [60]. 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 in our work. 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. Figure 11 in Appendix A.2 depicts its memory organization. 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 [50, 88, 97].

# 3 NVCT: A TOOL FOR STUDYING APPLICATION RECOMPUTABILITY

To enable our study on application recomputability in NVM, we introduce a PIN-based crash emulator, NVCT. NVCT includes a simulated multi-level, coherent cache hierarchy and main memory,

```
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; R1
8          ...
9          for() {...}// a second-level inner loop
10     }
11     for () { // a first-level inner loop; R2
12         ...
13     }
14     for () { // a first-level inner loop; R3
15         ...
16     }
17     for () { // a first-level inner loop; R4
18         ...
19         cache_block_flush(u, NR*sizeof(double));
20         cache_block_flush(r, NR*sizeof(double));
21     }
22     cache_block_flush(&it, sizeof(int));
23   }
24   //result verification
25   ...
26 }
```

**(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 comp loop
10         ...
11         //flush cache blocks
12     }
13     //result verification
14     ...
15 }
```

**(b) Restart MG.**

**Figure 2: Study recomputability of MG with NVCT.**

a random crash generator, a set of APIs to support the configuration of crash tests and application restart, and a tool to examine data inconsistency for post-crash analysis. 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 and main memory.

**Main memory simulation.** Different from the microarchitectural simulation of main memory, the main memory simulation in NVCT records data values and their corresponding memory addresses. Whenever the cache simulation writes back any cache line, the corresponding data values in the simulated main memory are updated. Using this method, we can easily determine data inconsistency between caches and main memory. During a crash test, the data values of user-specified data objects in the simulated main memory can be dumped into a file for post-crash analysis.

**Random crash generation.** NVCT emulates the occurrence of a crash by stopping application execution after a randomly selected instruction. Furthermore, NVCT can report call path information when a crash happens by leveraging CCTLib [20]. The call path information introduces program context information for analyzing crash results, which is helpful to distinguish those crash that happen in the same program statement, but with different call stacks.

**Calculation of data inconsistent rate.** NVCT reports data inconsistent rate after a crash happens. When a crash happens, NVCT examines dirty cache lines in the simulated cache hierarchy and compares with the corresponding cache block in main memory to determine the number of dirty data bytes in the cache line. 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.

**Application restart**. When restarting, NVCT loads data values from the file dumped by the main memory simulation to initialize user-specified data objects. Some data objects are initialized by the application itself. After that, NVCT resumes the main loop, starting from the beginning of the iteration where the crash happens.

**Putting all together.** To use NVCT, the user needs to insert APIs to specify (1) data objects that need to be persisted during application execution, (2) the initialization phase of the application for a restart, and (3) code regions where crashes can happen. The user also needs to configure cache simulation and crash tests (e.g., how many crash tests and what probability distribution the crash tests follow). During application execution, NVCT leverages the infrastructure of PIN to instrument the application and analyze instructions for cache and memory simulations. NVCT triggers a crash as configured, and then performs post-crash analysis to report data inconsistent rate and restart the application.

**An example**. Figure 2 gives an example of how we study application recomputability. This is a multi-grid (MG) numerical kernel from the NAS parallel benchmark suite [11] (NPB). Like many HPC applications, MG has a main computation loop, within which we persist two global data objects and a loop iterator [3] at somewhere (Lines 19-20, 22 in this example). After a crash happens, we restart MG using Figure 2b. To restart, the application re-initializes computation (Line 5 in Figure 2b), 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 outcome.

# 4 CHARACTERIZATION OF APPLICATION RECOMPUTABILITY

We characterize application recomputability to motivate our design.

## 4.1 Experiment Setup

**Benchmarks for evaluation.** We use all benchmarks from NPB. To enrich our benchmark collection, we add botsspar from SPEC OMP 2012 [2], kmeans from Rodinia [22] and LULESH [67]. In total, we have 11 benchmarks, covering dense linear algebra, sparse linear algebra, spectral methods, structured-grid, graph traversal,

[3] 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.

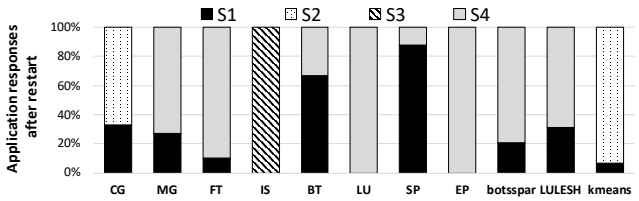



**Figure 3: Application responses after crash and restart. Figure annotation: S1 - successful recomputation without using extra iterations, S2 - successful recomputation with extra iterations, S3 - Interruption, and S4 - verification fails.**

and data mining. These benchmarks are chosen, because of their representativeness and explicit code structures to verify application outcomes. The input problems for these benchmarks are summarized as "input 3" in Table 11 in Appendix A.7. Table 1 summarizes these benchmarks and shows their characteristics.

**System configuration.** We simulate a three-level cache hierarchy, shown in Table 4 in Appendix A.3. We use both single and multiple threads to run benchmarks. We present the results of single thread, but the conclusions we draw from multiple threads are the same as from single thread.

**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 the number of 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 discrete uniform distribution. This method of interrupting applications is common in the research on system fault tolerance [19, 43, 44, 65, 98].

## 4.2 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 than the original execution; (3) Interruption: the application cannot run to completion (e.g., due to segfault); (4) Verification fails: the application cannot pass the acceptance verification, even after taking two times as many iterations as in the original execution.

Figure 3 and Table 1 (the last two columns) show the results based on the above classification. We notice that some applications show strong recomputability (e.g., 88% and 67% for SP and BT respectively). Some applications (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 of their algorithms to isolate propagation of data inconsistency. In particular, SP and BT, aiming to solve 3-dimensional compressible Navier-Stokes equations [57], decouple computation along x, y and z dimensions. Each dimension employs an iterative numerical solver. Iterative solvers tolerate data loss after crashes [12], 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 three dimensions. Although LU performs similar numerical simulation as SP and BT, data inaccuracy because of crash are 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 segmentation faults because of dangling or wild pointers. (4) EP has low recomputability, because it has a small memory footprint. Crash and restart leave most of data objects in stale states, violating the application requirement on data correctness.

**Conclusion 1.** Applications have quite different recomputability (e.g., the difference between SP and IS is 88%), 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 (i.e., the application passes acceptance verification without using extra iterations), 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 4a 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 the data object *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 rather large arrays, and a set of stencils are applied to them over many iterations. *r* are dominated by read accesses, 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).

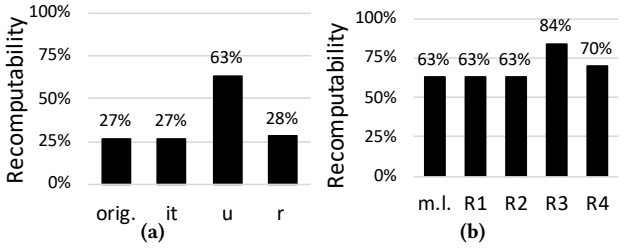



**Figure 4: The recomputability of MG after (a) persisting three different data objects in MG; and (b) persisting *u* in different code regions. "orig." stands for the recomputability without persisting anything. "m.l." stands for the case of persisting *u* at the end of each iteration of the main loop without considering code regions.**

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 2a. They represent four execution phases. They all update *u*. We persist *u* at the end of an execution phase (code region). Figure 4b 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.

Table 1: Benchmark information. "R/W" = "Read/Write ratio", "DO" = "data object", "iter" = "iterations".

| Benchmarks | Description | # of code regions | R/W | 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 | 21:1 | 947MB | 5.7MB | 2.3MB | 9.1 | 75 |
| **MG** | Structured grids | 4 | 7:1 | 3.4GB | 2.3GB | 1.2GB | 0 | 20 |
| **FT** | Spectral method | 4 | 1:1 | 5.1GB | 4.0GB | 4.0GB | 0 | 20 |
| **IS** | Graph traversal (sorting) | 8 | 2:1 | 1.0GB | 264MB | 4KB | N/A(segfault) | 10 |
| **BT** | Dense linear algebra | 15 | 2:1 | 1.43GB | 525.6MB | 361.2MB | 0 | 200 |
| **LU** | Dense linear algebra | 4 | 5:2 | 1.4GB | 599MB | 164MB | N/A (the verification fails) | 250 |
| **SP** | Dense linear algebra | 16 | 2:1 | 1.47GB | 561MB | 394MB | 0 | 400 |
| **EP** | Monte Carlo | 2 | 2:1 | 1MB | 1MB | 80B | N/A (the verification fails) | 65535 |
| **botsspar** | Sparse linear algebra | 4 | 2:1 | 3.74GB | 3.36GB | 3.36GB | 0 | 200 |
| **LELUSH** | Hydrodynamics modeling | 4 | 5:1 | 1.41GB | 251MB | 20MB | 0 | 3517 |
| **kmean** | Data mining | 1 | 9:2 | 222MB | 20B | 20B | 18.2 | 36 |

**Analysis.** MG implements a hierarchical multi-grid method that approximates the solution to a discrete Poisson equation. R3 corresponds to the solving phase on a coarse grid to accelerate the speed of computation convergence [19]. 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 random crashes, leading to higher recomputability. Other code regions work on a fine grid and the impact of data inconsistency there on application outcome correctness is limited. Hence persisting $u$ at the other code regions is not helpful.

**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.

## 5 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 (Sec 5.1) and where to persist them to maximize application recomputability (Sec 5.2). We show how to use EasyCrash at the end of this section (Sec 5.3).

### 5.1 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 of critical data objects. 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. Assuming that there are $P$ candidates (P can be hundreds or thousands in HPC applications), there are $2^P$ possible selections. We use statistical correlation analysis to efficiently select data objects.

Our method is based on the following observation. When a crash happens, data objects remaining 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 [106] to statistically quantify the correlation between the inconsistent rate of data objects and application recomputability. The result of the Spearman's rank correlation is the coefficient ($R_s$), which quantifies how well the relationship between two input vectors (data inconsistent rate and application recomputability) 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.

To use the Spearman's rank correlation analysis, we build two vectors for each candidate data object, using the results collected from a crash test campaign: One vector is composed of data inconsistent rates; The other vector is composed of application recomputation results (i.e., whether the application recomputes successfully or not). Each component of the two vectors is collected in one crash test of the crash test campaign. 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 the correlation coefficient 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 a common threshold [106], and less than it statistically shows a very strong correlation in our study.

**Verification of the selection of data objects.** To verify that our selection is effective, 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 5

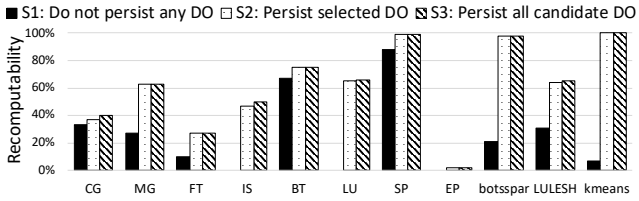



**Figure 5: Application recomputability under three strategies to persist data objects. Figure annotation: "DO" stands for data objects.**

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 of data objects.

## 5.2 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. Table 5 in Appendix A.4 summarizes the annotation for our formulation.

**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 that are typically used to update data objects iteratively. This code structure is very commonly used in HPC applications. A number of existing efforts are based on this code structure [8, 17, 28, 44, 63, 86, 87, 91]. Figure 2a shows an example of such a program abstraction for 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 such code regions easily represent computation phases of the application. 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 [44] to study application resilience to errors.

**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 the 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 $\tau$). Section 7 discusses how to decide $\tau$.

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 can be successfully recomputed. Based on the definition of recomputability (Section 2.2), 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 [4] of the code region $i$ to total execution time of the application.

Assume the code region $k$ ($1 \le k \le W$) is selected as a critical code region. After persisting critical data objects at the code region $k$, 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 the code region $k$, which 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 ($1 \le i \le W$ and $i \ne k$) 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 can meet 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 $k$ can meet 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 operation). Then, based on Equation 2 and $c'_k$, we calculate $Y'$ (recall that $Y'$ is the application recomputability after persisting the selected 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 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

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

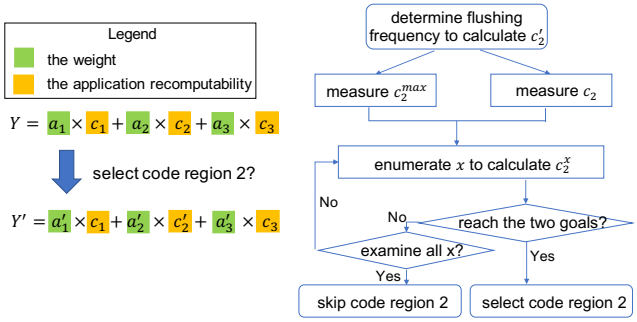



**Figure 6: An example of using our algorithm to decide whether a code region (code region 2) should be selected for a program with three code regions.**

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 loop in the code region $k$ [5] to maximize recomputability of the code region, 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 code region $k$). In particular, we persist data objects at every iteration of the 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 occur 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 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 performance loss ($l_k$) and the application recomputability ($c'_k$) for any code region $k$ ($1 \le k \le W$). To illustrate the above modeling process better, Figure 6 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.

Based on the above, 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 is larger than $\tau$ (the system efficiency goal). We also want to maximize application recomputability. This is a variant of the 0-1 knapsack problem [94] 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 we estimate $l_k$, we assume every cache block of data objects is in the cache, which overestimates performance overhead. However, overestimation is harmless, because it ensures that runtime overhead in production 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 the code region (not each code region). Our method, although avoids massive crash tests, ignores the possible propagation of computation inaccuracy from one code region to another. Such a method makes the measured recomputability smaller than the real recomputability. This means using our method, EasyCrash should result in larger recomputability and larger performance benefit in reality, which is good.

## 5.3 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 four 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., $c_k, 1 \le k \le W$) 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 ($c_k^{max}, 1 \le k \le W$ ). 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.

We discuss time cost for crash test campaigns in Section 8.

**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

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

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. Appendix A.4 shows the APIs provided by EasyCrash and an example on how to use them. For (1) and (2), the compiler can annotate the application with the APIs, freeing the programmer from changing the application.

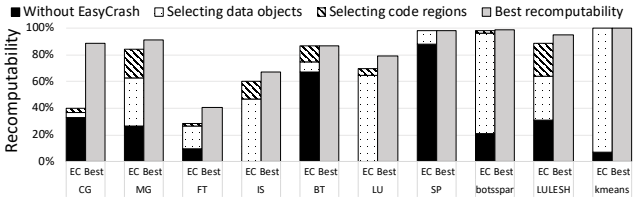



**Figure 7: Application recomputability with different techniques.**

## 6 EVALUATION

In this section, we study whether EasyCrash can effectively improve application recomputability and what is the runtime overhead of EasyCrash. In the next section, we evaluate system efficiency of EasyCrash in large scale systems in the context of C/R mechanisms. We use the benchmarks shown in Table 1. To calculate application recomputability, we use the method in Section 4.1 for crash tests.

We use two platforms for our study. One platform is an emulated NVM with traditional DDR4 DRAM. Table 8 in Appendix A.6 shows hardware details for this platform. This platform is commonly used in the existing work [26, 27, 42, 61, 104] and represents the future NVM as main memory, because its performance (latency and bandwidth) is the same as that of DRAM. The other platform is a system with Intel Optane DCPMM. Table 9 in Appendix A.6 shows hardware details for this platform. In the rest of the paper we report results for Intel Optane DCPMM. The results for the DRAM-based NVM emulation are in Appendix A.6.

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 $\tau$. We show the results of $t_s$ = 3% in this section. We do not present the results for 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). *Selection of data objects and code regions is summarized in Appendix A.5.*

**Recomputability improvement.** Figure 7 shows application recomputability after using EasyCrash. To reveal the effectiveness of our two techniques (i.e., selecting data objects and selecting code regions), we first measure recomputability without using the two techniques, 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 the effectiveness of EasyCrash, we also compare the *best recomputability* with the results of applying EasyCrash. We obtain the best recomputability by persisting critical data objects at each code region (if the code region has a loop structure, then 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 (shown in the last column of Table 2), which is not a practical solution for HPC. In addition, we do not show results for persisting all data objects, because in Section 5, we have shown that persisting critical data objects can achieve very similar recomputability as persisting all data objects. We have the following observations from Figure 7.

(1) EasyCrash achieves very high recomputability. Except for CG, the recomputability of applying EasyCrash is pretty close to the best one, with a difference of only 5% on average. For CG, there is a big difference (49%), because many successful recomputation tests in CG require extra iterations, which is not acceptable in EasyCrash due to the concerns on runtime overhead. Note that even with the big difference, EasyCrash still brings 4% improvement in system efficiency for CG (shown in Section 7).

(2) EasyCrash significantly improves application recomputability. This fact is especially pronounced in the benchmarks MG, botsspar and kmeans. We see 56%, 77%, and 93% improvement for the three benchmarks respectively. The average recomputability of all benchmarks after using EasyCrash is 75%, while it is 28% before using EasyCrash. Also, EasyCrash is able to transform 47% of crashes that cannot correctly recompute into the 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* [55] for best performance of cache flushing. Table 2 summarizes the execution time of persisting critical data objects for once (i.e., *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 the main computation loop (shown in the fifth column of the table), which is a case without the selection of data objects and code regions. This case cause 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, which is much larger than EasyCrash.

**Write Reduction.** We compare EasyCrash and the in-memory C/R mechanism in terms of the number of extra writes. For EasyCrash, the extra writes come from persisting critical data objects at critical code regions. As discussed in Section 2.1, 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 [8]. 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.

**Table 2: 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 |

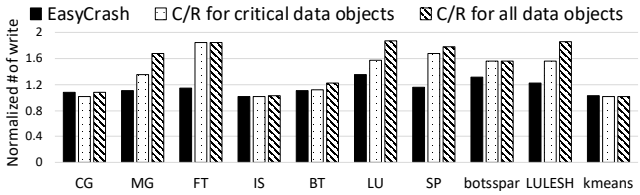



**Figure 8: Normalized number of NVM write.**

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). Also, we assume that checkpoint happens only once. This is a rather conservative assumption favoring the checkpointing mechanism. The checkpoint could happen more often (depending on system failure rate and application execution time), which causes more extra number of writes. We consider the system failure rate and application execution time to evaluate the effects of checkpoint in Section 7.

Figure 8 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% for the two methods of checkpointing respectively. Also, for those benchmarks with large data objects (e.g., FT, SP, and LU with data objects 10x larger than last level cache size), EasyCrash is especially beneficial since the number of extra writes in a persistence operation is bounded by last level cache size. A larger data object indicates that EasyCrash flushes more clean cache lines or non-resident cache blocks 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 the number of writes, but writing those small data objects does not usually cause a serious endurance problem.

## 7 END-TO-END EVALUATION

We evaluate EasyCrash in the context of large-scale parallel 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 performance analysis in Section 6.

**Basic assumptions.** We assume that the checkpointing process does not have any corruption. This is a common assumption [13, 16]. We model coordinated checkpointing, which is the most common practice of C/R in HPC and commonly used in the recent work [13, 74, 75] (we model and discuss uncoordinated checkpionting in Appendix A.10 for the completeness of the paper). 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 6), 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 included in $T\_chk$). Such a multi-level checkpoint mechanism is based on [74]. The checkpoint data should not be written into NVM-based main memory, because it significantly reduces memory space useful for applications.
(3) ***T_r***: The time for recovering from the previous checkpoint. Similar to the existing work [16], we assume $T\_r = T\_chk$.
(4) ***T_sync***: The time for synchronization across nodes. We use the assumption in [13]: 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 [105], $T = \sqrt{2 \times T\_chk \times MTBF}$. This formula has been shown to achieve almost identical performance as in realistic scenarios [35].
(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 [29], on average, half of a checkpoint interval for computation is wasted (i.e., $T\_vain = 50\% \times T$).

We summarize the *application related parameters* as follows.

(1) $R_{EasyCrash}$: The application recomputatbility achieved by using EasyCrash.
(2) $t_s$: The runtime overhead introduced by EasyCrash because of persisting critical data objects (e.g., 3% in our evaluation).

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 spent on the HPC system 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 HPC system efficiency by avoiding large recovery cost from the last checkpoint and increasing the checkpoint interval. EasyCrash also brings ignorable runtime overhead. Equation 8 models the total execution time with EasyCrash ($Total\_Time'$), where $N'$ and $T'$ are the number of checkpoints and their interval when using EasyCrash, and $M'$ is the number of crashes that have to go to the last checkpoint for recovery, and $M''$ is the number of crashes that use EasyCrash to recompute successfully.

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

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

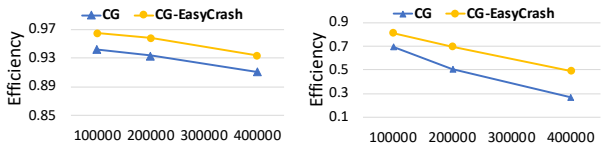
With EasyCrash, the checkpoint interval ($T'$) becomes longer ($T' > T$), and also should include a small runtime overhead ($t_s$). As a result, the number of checkpoints ($N'$) becomes smaller ($N' < N$), and the checkpoint overhead ($N' \times T_{chk}$) becomes smaller. With and without EasyCrash,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

With EasyCrash, once a crash happens, the system either goes to the last checkpoint with recovery overhead modeled as ($M' \times (T\_vain + T\_r + T\_sync)$), or uses EasyCrash to restart and successfully recompute with recovery overhead modeled as ($M'' \times (T\_r' + T\_sync)$). With NVM and EasyCrash, the recovery cost $T\_r$ becomes $T_r'$, which becomes smaller, because we load data objects from NVM-based main memory, not from local SSD or storage node. $T_r'$ is estimated using the total 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 [14, 100]. 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 [13, 16, 35]. We emulate the system with 100,000 nodes for a long simulation time (10 years, i.e., $Total\_Time$ is 10 years). Previous work [69] shows that systems in such a scale usually experience around 2 failures per day ($MTBF$ = 12 hours). Based on this data, we scale $MTBF$ as in [13] for 200,000 and 400,000 nodes. As a result, $MTBF$ for them are 6 and 3 hours respectively.

**Results for system efficiency.** Figure 9 shows the system efficiency with and without EasyCrash under different checkpoint overhead. We show the benchmarks with the lowest and highest recomputability (FT and SP respectively), and the average recomputability of *all benchmarks*. The results for other benchmarks can be found in Appendix A.9. EasyCrash improves system efficiency


**Figure 9: 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)** $T\_chk = 3200s$.

**Figure 10: System efficiency for CG without and with EasyCrash when the system scales from 100,000 to 200,000 and 400,000 nodes**

by 2%-24%. On average, the system efficiency with EasyCrash is improved by 2%, 3% and 15% when the checkpoint overhead is 32s, 320s, and 3200s respectively.

Furthermore, 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 Appendix A.9. Figure 10 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 $\tau$.** To ensure the system with EasyCrash has higher efficiency than with C/R, the application recomputability must be higher than a threshold $\tau$ (see Section 5.2). Given $Total\_Time$ and Equations 8 and 9, we calculate a lower bound of $R_{EasyCrash}$, which is $\tau$ (see Equation 12 in Appendix A.8 for details).

## 8 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 $\tau$ can be calculated.

For (3), we use crash tests (Section 5.3). 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 based on an observation shown in Appendix A.7.

Our observation reveals that by using EasyCrash to persist critical data objects at selected code regions, the application using different input problems [6] shows the similar recomputability. Our evaluation with ten benchmarks, each of which uses four input problems, shows that the variance of recomputability is less than 9%. 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, we reduce test time by more than 99%. *Crash tests for each benchmark can be finished in less than 14 minutes using two 48-core machines shown in Table 8.* 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 data objects and small memory footprint. When a crash happens to the application, most of the application data are resident in the cache and lost. To ensure high recomputability, we have to persist data objects frequently, which causes high runtime overhead. (2) Applications with no tolerance for computation errors. These applications regard any application outcome different from that of the golden run as incorrect. Many of our crash-and-restart tests generate outcomes different from those of the golden run, but these tests pass the acceptance verification.

For (1), the system can disable EasyCrash and only employ the traditional checkpoint mechanism to handle failures. Because of the 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 is different from that of the golden run, the users can claim a silent data corruption (SDC) happens [44, 98]. 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 [18, 72, 73, 83, 84, 93]. The applications that cannot tolerate SDC cannot use EasyCrash.

**Why not flush the whole cache hierarchy using `WBINVD`?** `WBINVD` is an x86 instruction that writes back all dirty cache lines in the processor's *private* caches to main memory and invalidates the private caches [51]. Using `WBINVD`, there is a potential to skip the selection of data objects, simplifying EasyCrash. However, we do not use it because of the following reasons. (1) `WBINVD` is a privileged instruction, preventing the program to perform cache flushing at the user level; (2) After executing `WBINVD`, the processor does not wait for the *shared* caches to complete their write-back before proceeding with instruction execution. There is no guarantee on when flushing shared caches is completed, even with fence instructions. (3) `WBINVD` is costly, especially on a manycore platform, because of cache coherence overhead.

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

## 9 RELATED WORK

Some efforts focus on establishing crash consistency in NVM [25, 33, 59, 96, 97] 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 [21, 33, 49, 97]. Some work on NVM-aware data structures [96, 103] 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. Different from the above work that relies on strong guarantees on crash consistency and heavily involves programmers to enforce crash consistency, EasyCrash enables automatic exploration of application recomputability without extensive changes to applications.

A few recent efforts focus on using NVM for HPC fault tolerance [8, 37, 104]. They avoid flushing caches for high performance, and rely on algorithm knowledge [104] or high requirements on loop structures [8, 37] to recover computation upon application failures. EasyCrash is significantly 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 [13] is an example of approximate computing. 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.

## 10 CONCLUSIONS

The emergence of NVM provides many opportunities for HPC to enable scalable scientific simulation and high system efficiency. However, how to integrate 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 introduce a novel methodology that relaxes the requirement on crash consistency (but with constraints) for smaller memory consumption, higher system efficiency, smaller number of writes and few application modification. Based on the characteristics of error resilience intrinsic to HPC applications, we demonstrate the great potential of this methodology with a spectrum of HPC applications.

# REFERENCES


[1] 2006. SPEC CPU2006. www.spec.org/cpu2006. (2006).
[2] 2012. SPEC OMP2012. www.spec.org/omp2012. (2012).
[3] 2015. Intel and Micron produce breakthrough memory technology. (2015).
[4] 2019. SuperLU. https://portal.nersc.gov/project/sparse/superlu/. (2019).
[5] Mark James Abraham, Teemu J Murtola, Roland Schulz, Szilárd Páll, J. C. Smith, Berk Hess, and Erik Lindahl. 2015. GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers.
[6] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. 2018. Write-rationing Garbage Collection for Hybrid Memories. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*.
[7] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. 2019. Crystal Gazer: Profile-Driven Write-Rationing Garbage Collection for Hybrid Memories. In *Abstracts of the 2019 SIGMETRICS/Performance Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '19)*.
[8] Mohammad Alshboul, James Tuck, and Yan Solihin. 2018. Lazy Persistency: A High-performing and Write-efficient Software Persistency Technique. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*.
[9] Mihnea Andrei, Christian Lemke, Günter Radestock, Robert Schulze, Carsten Thiel, Rolando Blanco, Akanksha Meghlan, Muhammad Sharique, Sebastian Seifert, Surendra Vishnoi, Daniel Booss, Thomas Peh, Ivan Schreter, Werner Thesing, Mehul Wagle, and Thomas Willhalm. 2017. SAP HANA Adoption of Non-volatile Memory. *Proc. VLDB Endow.* 10, 12 (Aug. 2017).
[10] Woongki Baek and Trishul M. Chilimb. 2010. Green: a framework for supporting energy-conscious programming using controlled approximatio. In *Programming Language Design and Implementation (PLDI)*.
[11] D. H. Bailey, L. Dagum, E. Barszcz, and H. D. Simon. 1993. NAS parallel benchmark results. *IEEE Parallel Distrib. Technol.* 1, 1 (Feb. 1993), 43–51.
[12] Iván Bermejo-Moreno, Julien Bodart, Johan Larsson, Blaise M. Barney, Joseph W. Nichols, and Steve Jones. 2013. Solving the Compressible Navier-stokes Equations on Up to 1.97 Million Cores and 4.1 Trillion Grid Points. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*.
[13] B.Fang, Q.Guan, N.Debardeleben, K.Pattabiraman, and M.Ripeanu. 2017. LetGo: A Lightweight Continuous Framework for HPC Applications Under Failures. In *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing*.
[14] Wahid Bhimji, Debbie Bard, Melissa Romanus, Andrey Ovsyannikov, Brian Friesen, Matt Bryson, Joaquin Correa, Glenn K. Lockwood, Vakho Tsulaia, Suren Byna, Steve Farrell, Doga Gursoy, Christopher S. Daley, Vincent E. Beckner, Brian van Straalen, Nicholas J. Wright, and Katie Antypas. 2016. Accelerating Science with the NERSC Burst Buffer Early User Program.
[15] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*.
[16] George Bosilca, Aurélien Bouteiller, Elisabeth Brunet, Franck Cappello, Jack Dongarra, Amina Guermouche, Thomas Herault, Yves Robert, Frédéric Vivien, and Dounia Zaidouni. 2014. Unified Model for Assessing Checkpointing Protocols at Extreme-scale. *Concurr. Comput. : Pract. Exper.* (2014).
[17] G. Bronevetsky and B. de Supinski. 2008. Soft Error Vulnerability of Iterative Linear Algebra Methods. In *International Conference on Supercomputing (ICS)*.
[18] Michael Carbin, Sasa Misailovic, and Martin C. Rinard. 2013. Verifying Quantitative Reliability for Programs that Execute on Unreliable Hardware. In *International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*.
[19] Marc Casas, Bronis R. de Supinski, Greg Bronevetsky, and Martin Schulz. 2012. Fault Resilience of the Algebraic Multi-grid Solver. In *ACM International Conference on Supercomputing*.
[20] Milind Chabbi, Xu Liu, and John Mellor-Crummey. 2014. Call Paths for Pin Tools. In *Proceedings of International Symposium on Code Generation and Optimization*.
[21] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *Proceedings of International Conference on Object Oriented Programming Systems Languages & Applications*.
[22] 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*.
[23] Shimin Chen and Qin Jin. 2015. Persistent B+-trees in Non-volatile Main Memory. *Proc. VLDB Endow.* 8, 7 (Feb. 2015), 786–797.
[24] Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. 2013. Analysis and Characterization of Inherent Application Resilience for Approximate Computing. In *Proceedings of Annual Design Automation Conference*.
[25] J. Coburn, A.M. Caulfield, A. Akel, L.M. Grupp, R.K. Gupta, R. Jhala, and S. Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*.
[26] Nachshon Cohen, David T. Aksun, Hillel Avni, and James R. Larus. 2019. Fine-Grain Checkpointing with In-Cache-Line Logging. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*.
[27] Nachshon Cohen, David T. Aksun, and James R. Larus. 2018. Object-oriented Recovery for Non-volatile Memory. *Proc. ACM Program. Lang.* 2, OOPSLA (Oct. 2018).
[28] Matthew Curtis-Maury, Ankur Shah, Filip Blagojevic, Dimitrios S. Nikolopoulos, Bronis R. de Supinski, and Martin Schulz. 2008. Prediction models for multi-dimensional power-performance optimization on many cores. In *International Conference on Parallel Architectures and Compilation Techniques*.
[29] J. T. Daly. 2006. A Higher Order Estimate of the Optimum Checkpoint Interval for Restart Dumps. *Future Generation Computer Systems* (2006).
[30] N. DeBardeleben, J. Laros, J. Daly, S. Scott, C. Engelmann, and B. Harrod. 2019. High-End Computing Resilience: Analysis of Issues Facing the HEC Community and Path-Forward for Research and Development. (01 2019).
[31] Research Computing Documentation. 2017. MpiBLAST — Research Computing Documentation,. (2017). https://wiki.rc.usf.edu/index.php?title=MpiBLAST&oldid=1634
[32] Mingkai Dong and Haibo Chen. 2017. Soft Updates Made Simple and Fast on Non-volatile Memory. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*.
[33] 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 European Conference on Computer Systems*.
[34] Ifeanyi P. Egwutuoha, David Levy, Bran Selic, and Shiping Chen. 2013. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing* (2013).
[35] N. El-Sayed and B. Schroeder. 2014. Checkpoint/restart in practice: When 'simple is better'. In *IEEE International Conference on Cluster Computing*.
[36] J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann. 2012. Combining Partial Redundancy and Checkpointing for HPC. In *2012 IEEE 32nd International Conference on Distributed Computing Systems*.
[37] H. Elnawawy, M. Alshboul, J. Tuck, and Y. Solihin. 2017. Efficient Checkpointing of Loop-Based Codes for Non-volatile Main Memory. In *International Conference on Parallel Architectures and Compilation Techniques*.
[38] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. [n. d.]. A Survey of Rollback-recovery Protocols in Message-passing Systems. *ACM Comput. Surv.* ([n. d.]).
[39] Charles R. Ferenbaugh. 2015. PENNANT: An Unstructured Mesh Mini-app for Advanced Architecture Research. *Concurr. Comput. : Pract. Exper.* 27, 17 (Dec. 2015).
[40] P. Fernando, A. Gavrilovska, S. Kannan, and G. Eisenhauer. 2018. NVStream: Accelerating HPC Workflows with NVRAM-based Transport for Streaming Objects. In *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing*.
[41] Leonardo Fialho and Dolores Rexachs. 2011. Defining the Checkpoint Interval for Uncoordinated Checkpointing Protocols.
[42] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Aasheesh Kolli, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2019. Software Wear Management for Persistent Memories. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*.
[43] Qiang Guan, Nathan Debardeleben, Sean Blanchard, and Song Fu. 2014. F-sefi: A Fine-grained Soft Error Fault Injection Tool for Profiling Application Vulnerability. In *IEEE Parallel and Distributed Processing Symposium*.
[44] L. Guo, D. Li, I. Laguna, and M. Schulz. 2018. FlipTracker: Understanding Natural Error Resilience in HPC Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*.
[45] Y. Guo, Y. Hua, and P. Zuo. 2018. A Latency-optimized and Energy-efficient Write Scheme in NVM-based Main Memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2018).
[46] Saurabh Gupta, Tirthak Patel, Christian Engelmann, and Devesh Tiwari. 2017. Failures in Large Scale Systems: Long-term Measurement, Analysis, and Implications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*.
[47] Michael A. Heroux, Douglas W. Doerfler, Paul S. Crozier, James M. Willenbring, H. Carter Edwards, Alan Williams, Mahesh Rajan, Eric R. Keiter, Heidi K. Thornquist, and Robert W. Numrich. 2009. Improving Performance via Mini-applications. In *SANDIA REPORT*.
[48] C. Hsu and W. Feng. 2005. A Power-Aware Run-Time System for High-Performance Computing. In *ACM/IEEE Conference on Supercomputing*.
[49] Intel. 2014. Persistent Memory Development Kit. https://pmem.io/. (2014).
[50] Intel. 2014. Intel NVM Library. http://pmem.io/nvml/libpmem/. (2014).
[51] Intel. 2016. Intel 64 and IA-32 Architectures Software Developer's Manual. (2016). https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developers-manual.pdf

[52] Intel. 2018. Intel Persistence Inspector. https://software.intel.com/en-us/articles/detect-persistent-memory-programming-errors-with-intel-inspector-persistence-inspector. (2018).

[53] Intel. 2018. Intel Pmemcheck. https://software.intel.com/en-us/articles/discover-persistent-memory-programming-errors-with-pmemcheck. (2018).

[54] Intel. 2018. Intel pmreorder. https://pmem.io/pmdk/manpages/linux/master/pmreorder/pmreorder.1.html. (2018).

[55] Intel Corporation. 2009. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Number 248966-018.

[56] Xu Ji, Chao Wang, Nosayba El-Sayed, Xiaosong Ma, Youngjae Kim, Sudharshan S. Vazhkudai, Wei Xue, and Daniel Sanchez. 2017. Understanding Object-level Memory Access Patterns Across the Spectrum. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.

[57] Haoqiang Jin, Michael A. Frumkin, and Jerry Mingtao Yan. 1999. The OpenMP Implementation of NAS Parallel Benchmarks and its Performance.

[58] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic. 2013. Optimizing Checkpoints Using NVM as Virtual Memory. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*.

[59] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch. 2016. High-Performance Transactions for Persistent Memories. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*.

[60] Argonne National Lab. 2019. 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).

[61] Philip Lantz, Subramanya Dulloor, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. 2014. Yat: A Validation Framework for Persistent Memory Software. In *USENIX Annual Technical Conference*.

[62] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger. 2010. Phase-Change Technology and the Future of Main Memory. *IEEE Micro* (2010).

[63] Dong Li, Bronis de Supinski, Martin Schulz, Dimitrios S. Nikolopoulos, and Kirk W. Cameron. 2010. Hybrid MPI/OpenMP Power-Aware Computing. In *International Parallel and Distributed Processing Symposium*.

[64] D. Li, J. S. Vetter, G. Marin, C. McCurdy, C. Cira, Z. Liu, and W. Yu. 2012. Identifying Opportunities for Byte-Addressable Non-Volatile Memory in Extreme-Scale Scientific Applications. In *IPDPS*.

[65] D. Li, J. S. Vetter, and W. Yu. 2012. Classifying Soft Error Vulnerabilities in Extreme-Scale Scientific Applications Using a Binary Instrumentation Tool. In *Conference for High Performance Computing, Networking, Storage and Analysis*.

[66] Y. Liu, R. Nassar, C. Leangsuksun, N. Naksinehaboon, M. Paun, and S. Scott. 2007. A reliability-aware approach for an optimal checkpoint/restart model in HPC environments. In *2007 IEEE International Conference on Cluster Computing*.

[67] LLNL. 2013. LULESH 2.0. https://github.com/LLNL/LULESH. (2013).

[68] 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*.

[69] C. D. Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer. 2014. Lessons Learned from the Analysis of System Failures at Petascale: The Case of Blue Waters. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*.

[70] 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)*.

[71] Esteban Meneses, Xiang Ni, Terry Jones, and Don Maxwell. 2015. Analyzing the Interplay of Failures and Workload on a Leadership-Class Supercomputer.

[72] J. Meng, A. Raghunathan, S. Chakradhar, and S. Byna. 2010. Exploiting the forgiving nature of applications for scalable parallel execution. In *IEEE International Symposium on Parallel and Distributed Processing*.

[73] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. Rinard. 2014. Chisel: Reliability- and Accuracy-Aware Optimization of Approximate Computational Kernels. In *International Conference on Object Oriented Programming Systems Languages and Applications*.

[74] Kathryn Mohror, Adam Moody, Greg Bronevetsky, and Bronis R. de Supinski. 2014. Detailed Modeling and Evaluation of a Scalable Multilevel Checkpointing System. *IEEE Trans. Parallel Distrib. Syst.* 25, 9 (2014), 2255–2263.

[75] A. Moody, G. Bronevetsky, K. Mohror, and B. de Supinski. 2010. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *International Conference for High Performance Computing, Networking, Storage and Analysis*.

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

[77] 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*.

[78] Steven Pelly, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistency. In *International Symposium on Computer Architecture*.

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

[80] Ian R. Philp. 2005. Software Failures and the Road to a Petaflop Machine. In *1st Workshop on High Performance Computing Reliability Issues (HPCRI) 2005*.

[81] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. 2009. Enhancing lifetime and security of PCM-based Main Memory with Start-Gap Wear Leveling. In *IEEE/ACM International Symposium on Microarchitecture*.

[82] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. 2015. ThyNVM: Enabling Software-Transparent Crash Consistency in Persistent Memory Systems. In *Proceedings of the International Symposium on Microarchitecture (MICRO'15)*.

[83] Martin Rinard. 2006. Probabilistic accuracy bounds for fault-tolerant computations that discard task. In *International Conference on Supercomputing (ICS)*.

[84] Martin Rinard, Henry Hoffmann, Sasa Misailovic, and Stelios Sidiroglou. 2010. Patterns and Statistical Analysis for Understanding Reduced Resource Computing. In *Onward! 2010*.

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

[86] Barry Rountree, David K. Lowenthal, Bronis R. de Supinski, Martin Schulz, Vincent W. Freeh, and Tyler K. Bletsch. 2009. Adagio: making DVS practical for complex HPC applications. In *International conference on Supercomputing (ICS)*.

[87] Barry Rountree, David K. Lowenthal, Martin Schulz, and Bronis R. de Supinski. 2011. Practical performance prediction under Dynamic Voltage Frequency Scaling. In *International Green Computing Conference and Workshops*.

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

[89] Adrian Sampson, André Baixo, Benjamin Ransford, Thierry Moreau, Joshua Yip, Luis Ceze, and Mark Oskin. 2015. ACCEPT: A Programmer-Guided Compiler Framework for Practical Approximate Computing. In *University of Washington Technical Report*.

[90] 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)*.

[91] M. Shantharam, S. Srinivasmurthy, and P. Raghavan. 2011. Characterizing the Impact of Soft Errors on Iterative Methods in Scientific Computing. In *International Conference on Supercomputing (ICS)*.

[92] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. 2012. Brief Announcement: The Problem Based Benchmark Suite. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '12)*.

[93] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 2011. Managing Performance vs. Accuracy Trade-offs with Loop Perforation. In *ACM SIGSOFT Symposium and European Conference on Foundations of Software Engineering (FSE)*.

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

[95] John R. Tramm, Andrew R. Siegel, Tanzima Islam, and Martin Schulz. 2014. XSBench – The Development and Verification of A Performance Abstraction for Monte Carlo Reactor Analysis. In *International Conference on Physics of Reactors*.

[96] S. Venkataraman, N. Tolia, P. Ranganathan, and R. 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*.

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

[98] K. Wu, W. Dong, Q. Guan, N. DeBardeleben, and D. Li. 2018. Modeling Application Resilience in Large-scale Parallel Execution. In *Proceedings of the 47th International Conference on Parallel Processing*.

[99] K. Wu, Y. Huang, and D. Li. 2017. Unimem: Runtime Data Management on Non-Volatile Memory-based Heterogeneous Main Memory. In *International Conference for High Performance Computing, Networking, Storage and Analysis*.

[100] K. Wu, F. Ober, S. Hamlin, and D. Li. 2017. Early Evaluation of Intel Optane Non-Volatile Memory with HPC I/O Workloads. (2017). arXiv:cs.DC/1708.02199

[101] K. Wu, J. Ren, and D. Li. 2018. Runtime Data Management on Non-volatile Memory-based Heterogeneous Memory for Task-parallel Programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*.

[102] 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)*.

[103] 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*.

[104] S. Yang, K. Wu, Y. Qiao, D. Li, and J. Zhai. 2017. Algorithm-Directed Crash Consistence in Non-volatile Memory for HPC. In *IEEE Cluster Computing*.
[105] John W. Young. 1974. A First Order Approximation to the Optimum Checkpoint Interval. *Commun. ACM* (1974).
[106] Jerrold H. Zar. 1972. Significance Testing of the Spearman Rank Correlation Coefficient. *J. Amer. Statist. Assoc.* 67, 339 (1972), 578–580.
[107] Pengfei Zuo, Yu Hua, and Jie Wu. 2019. Level Hashing: A High-performance and Flexible-resizing Persistent Hashing Index Structure. *ACM Trans. Storage* (2019).
[108] P. Zuo, Y. Hua, M. Zhao, W. Zhou, and Y. Guo. 2019. Write Deduplication and Hash Mode Encryption for Secure Nonvolatile Main Memory. *IEEE Micro* 39, 1 (2019), 44–51.

# Appendix A DISCUSSIONS

We add the following discussions to supplement the paper. The discussions present a survey on HPC applications (Appendix A.1), the memory organization in Optane DCPMM (Appendix A.2), NVCT configuration for application recomputability study (Appendix A.3), model annotations and APIs in EasyCrash design and an example to show how to use EasyCrash (Appendix A.4), some details on the selection of data objects and code regions after using EasyCrash (Appendix A.5), performance evaluation results with emulated NVM based on DRAM (Appendix A.6). The discussions also add justifications on our proposed solutions to avoid the overhead of crash tests based on the correlation between small and large input problems (Appendix A.7), formulation to calculate $\tau$ (Appendix A.8), more end-to-end evaluation results for coordinated checkpointing (Appendix A.9), and extension of performance modeling in Section 7 to consider uncommon uncoordinated checkpointing (Appendix A.10).

## A.1 A Survey on HPC Applications

We study 60 HPC applications from the following benchmark suites, SPEC OMP 2012 [2], Rodinia [22], SPEC CPU 2006 [1], PBBS [92], NPB [11], PARSEC [15], and six representative HPC applications [4, 5, 31, 39, 47, 67]. Table 3 reveals that all of these applications in various scientific and engineering fields have a main computation loop which dominates the execution time, and 51 of them have a verification phase.

## A.2 Memory Organization in Optane DCPMM

Figure 11 shows memory organization in Optane DCPMM. In the app-direct mode, DRAM and NVM are logically and physically placed side by side. DRAM and NVM shares a physical address space but with different addresses. With this mode, cache flushing writes back cache blocks to NVM. When a crash happens and the application restarts, data objects in DRAM are lost and loaded from last checkpoint, and data objects in NVM are loaded from a memory mapped file [50, 88, 97].

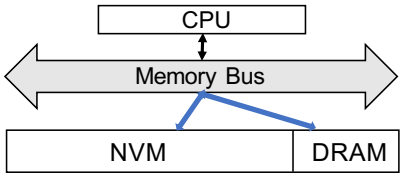



**Figure 11: Memory organization in Optane DCPMM.**

## A.3 Cache Configuration in NVCT

We use NVCT to examine data objects in main memory and caches after a random crash happens. Table 4 shows the configuration of the cache hierarchy for our application recomputability study.

## A.4 Model Annotation and EasyCrash APIs

Table 5 summarizes model parameters used in Section 5. Table 6 shows the APIs provided by EasyCrash.

**An example.** Figure 12 shows an example of how the user uses EasyCrash APIs listed in Table 6. This example is a multi-grid (MG) numerical kernel from NPB benchmark suite (We see MG in Section 3 too).

When using EasyCrash, the user is expected to use Intel PMDK [49] to create a memory-mapped heap file in NVM and load data objects (after a crash) from the file in NVM. PMDK uses environment variables to control location and size of memory mapped file to load data objects. The user uses the EasyCrash's customized memory allocation API to identify all candidate data objects, shown in Lines 4 and 5 in Figure 12. The user also need to insert the API *easycrash()* to specify code regions and main loop. In this example, the API is inserted at the end of first level inner loops and the main loop, which is exemplified in Lines 13,17,21,25, and 27. Note that if there is no first-level inner loop, the user is free to insert the API at the end of the main loop or other code regions.EasyCrash automatically decides which data objects should be persisted (Section 5.1) and where to persist them (Section 5.2) to maximize application recomputability. Based on the decisions, EasyCrash persists critical data objects by cache flushing exemplified in Lines 19, 20 and 22 in Figure 2a.

```
1  #include <libvmmalloc.h>
2  #include <libeasycrash.h>
3  ...
4  double u = easycrash_mem_alloc(NR*sizeof(double));
5  double r = easycrash_mem_alloc(NR*sizeof(double));
6  void main(int argc, char **argv) {
7    int it;
8    initialize();
9    for (it = 1; it <= nit; it++) {//main comp loop
10     for () { // a first-level inner loop; R1
11         ...
12         for() {...}// a second-level inner loop
13         easycrash();
14     }
15     for () { // a first-level inner loop; R2
16         ...
17          easycrash();
18     }
19     for () { // a first-level inner loop; R3
20         ...
21         easycrash();
22     }
23     for () { // a first-level inner loop; R4
24         ...
25         easycrash();
26     }
27     easycrash();
28   }
29   //result verification
30   ...
31 }
```

**Figure 12: MG with a few changes to use EasyCrash. The changes are highlighted with blue.**

## A.5 Results of Selection of Data Objects and Code Regions After Applying EasyCrash

**Selection of data objects.** In Section 5.1, we explain how EasyCrash selects data objects as critical data objects. Leveraging the statistical correlation analysis, EasyCrash selects data objects from

Table 3: HPC Applications in our study. "Iter." = "have an iterative code structure?". "Veri." = "have a verification phase?".

| | ID # | Benchmarks | Iter. | Veri. | | ID # | Benchmarks | Iter. | Veri. | | ID# | Benchmarks | Iter. | Veri. |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **SPEC (OMP)** | 1 | md | Y | Y | **SPEC CPU** | 21 | astar | Y | Y | **NPB** | 41 | CG | Y | Y |
| | 2 | bwaves | Y | Y | | 22 | libquantum | Y | N | | 42 | MG | Y | Y |
| | 3 | nab | Y | Y | | 23 | sjeng | Y | N | | 43 | FT | Y | Y |
| | 4 | bt331 | Y | Y | | 24 | hmmer | Y | N | | 44 | IS | Y | Y |
| | 5 | botsalgn | Y | Y | | 25 | gobmk | Y | N | | 45 | EP | Y | Y |
| | 6 | botsspar | Y | Y | | 26 | gamess | Y | Y | | 46 | BT | Y | Y |
| | 7 | ilbdc | Y | Y | | 27 | milc | Y | Y | | 47 | SP | Y | Y |
| | 8 | fma3d | Y | Y | | 28 | zeusmp | Y | N | | 48 | LU | Y | Y |
| | 9 | swim | Y | Y | | 29 | gromacs | Y | Y | **PARSEC** | 49 | blackscholes | Y | Y |
| | 10 | mgrid331 | Y | Y | | 30 | cactusAMD | Y | Y | | 50 | ferret | Y | Y |
| | 11 | applu331 | Y | Y | | 31 | leslie3d | Y | N | | 51 | dedup | Y | Y |
| | 12 | smithwa | Y | Y | | 32 | namd | Y | Y | | 52 | canneal | Y | Y |
| | 13 | kdtree | Y | Y | | 33 | soplex | Y | N | | 53 | fluidanimate | Y | Y |
| | 14 | imagick | Y | Y | | 34 | calculix | Y | N | | 54 | swaptions | Y | Y |
| **Rodinia** | 15 | Kmeans | Y | Y | | 35 | lbm | Y | N | **Individual HPC apps** | 55 | LULESH [67] | Y | N |
| | 16 | Hotspot3D | Y | Y | | 36 | wrf | Y | Y | | 56 | gromacs [5] | Y | Y |
| | 17 | KNN | Y | Y | | 37 | tonto | Y | Y | | 57 | mpiBLAST [31] | Y | Y |
| | 18 | backprop | Y | Y | **PBBS** | 38 | SORT | Y | Y | | 58 | PENNANT [39] | Y | Y |
| | 19 | Myocyte | Y | Y | | 39 | RDUPS | Y | Y | | 59 | SuperLU [4] | Y | Y |
| | 20 | Streamcluster | Y | Y | | 40 | SF | Y | Y | | 60 | miniFE [47] | Y | Y |

Table 4: Cache configuration in NVCT for our crash tests in Section 4.

| | |
|---|---|
| L1 Cache: | 32KB and 8-way set associativity |
| L2 Cache: | 1MB and 12-way set associativity |
| L3 Cache: | 19.25MB, 11-way set associativity |
| Cache Line Size: | 64 byte |
| Write Policy: | write-back, write-allocation |
| Eviction Policy: | LRU policies |

some candidates and persists the selected data objects to improve application recomputability. Table 7 shows the results of selection of data objects for benchmarks listed in Table 1.

**Selection of Code Regions.** In Section 5.2, we explain how EasyCrash selects code regions as critical code regions. EasyCrash persists critical data objects at critical code regions. Table 1 shows total number of code regions in each benchmark. More details on which code regions are selected can be found from our open source code[7].

## A.6 Performance Evaluation with DRAM-based NVM Emulation

We evaluate EasyCrash on two platforms. In Section 6, we use Intel Optane DCPMM; In this section, we use DRAM to emulate NVM. Tables 8 and 9 summarize the two platforms we use for evaluation (including crash emulation and performance study).

**Effectiveness of EasyCrash.** Figure 13 shows the application recomputability before and after we apply EasyCrash on the DRAM-based NVM emulation platform. Similar as Section 6, we report how different techniques(i.e., selecting data objects and selecting code regions) helps improve application recomputability. To show the effectiveness of EasyCrash, we also show the *best recomputability*

[7]https://github.com/NVMCrashTester/NVCT/tree/master/Benchmarks

Table 5: Model annotation

| Parameter | Description |
|---|---|
| $P$ | # of candidates of critical data objects |
| $R_s$ | Spearman's rank correlation coefficient |
| $a_k$ | The ratio of the accumulated execution time of the code region $k$ to the total execution time of the application |
| $c_k$ | Recomputability of the code region $k$ |
| $Y$ | Application recomputability |
| $l_k$ | The performance loss due to persistence operations in the code region $k$ |
| $t_s$ | Runtime overhead because of persistence operations in the application |
| $\tau$ | The system efficiency goal for long-running applications with EasyCrash |
| $N$ **and** $M$ | # checkpoint and # crashes in the whole system time |
| $W$ | # code regions in the application |
| $M'$ | # crashes that go to the last checkpoint for recovery after using EasyCrash |
| $M''$ | # crashes that use EasyCrash to recompute successfully |
| **Any para. with "prime"** | The corresponding parameters after applying EasyCrash |
| $c_k^{max}$ | best recomputability of the code region $k$ after persisting data objects |
| $x$ | The frequency to persist data objects in a loop-based code region |
| $c_k^x$ | Recomputability when using $x$ as the frequency for cache flushing in the code region $k$ |

*results*, and compare them with those after applying EasyCrash. The measurement method for the best recomputability results is described in Section 6. The overhead of achieving the best recomputability is very high (shown in the last column of Table 10), which

**Table 6: EasyCrash APIs**

| Signature | Description |
|---|---|
| void* easycrash_mem_alloc(size_t size); | A customized memory allocation API to identify candidate data objects for EasyCrash. |
| void easycrash(); | An API to identify the end of first-level inner loop (or code region if there is no inner loop). |

**Table 7: The candidates of critical data objects and selected critical data objects for benchmarks listed in Table 1.**

| Benchmarks | Candidates of critical data objects | Selected critical data objects |
|---|---|---|
| **CG** | x, r, p, q, z | p, q |
| **MG** | u, r | u |
| **FT** | ty1, ty2, u0, u1 | ty1, ty2, u0, u1 |
| **IS** | key_array, key_buff1, key_buff2, bucket_ptrs | bucket_ptrs |
| **BT** | us, vs, ws, qs, rho_i, square, u, rhs, ue, fjac, njac, lhs | us, vs, ws, qs, rho_i, square, u |
| **LU** | u, rsd, frct, gs, rho_i, rsdnm, a, b, c, d, au, bu, cu, du | u |
| **SP** | u, us, vs, ws, qs, rho_i, speed,square, rhs, cv, rhon, rhos, rhoq, lhs, lhsp, lhsm | u, us, vs, ws, qs, rho_i, speed, square |
| **botsspar** | tmp | tmp |
| **LULESH** | m_x,m_y,m_z,m_xd,m_yd,m_zd,m_xdd,m_ydd,m_zdd,m_fx, m_fy, m_fz, m_symmX, m_symmY, m_symmZ | m_symmX, m_symmY, m_symmZ |
| **kmeans** | cluster_centres | cluster_centres |

**Table 8: DRAM-based NVM emulation platform."Mem." stands for "main memory".**

| | |
|---|---|
| CPU: | Two Xeon Gold 6126 processors (Skylake) @ 2.6 GHz |
| Mem.: | DDR4 129GB BW: 106 GB/s Latency: 87 ns |

**Table 9: Optane DCPMM platform."Mem." stands for "main memory".**

| | |
|---|---|
| CPU: | Two Xeon(R) Platinum 8260L CPU @ 2.4 GHz |
| Mem.: | Optane DCPMM 1.5TB |
| | Bandwidth: read: 39 GB/s write: 13GB/s |
| | Latency: sequential: 174 ns random: 304 ns |

means the method to get the best recomputability is not a practical solution.

Figure 13 shows that EasyCrash achieve very high recomputability. The difference between the best recomputability and the recomputability after applying EasyCrash is only 9% at most. Except for CG, the recomputability after applying EasyCrash is pretty close to the best one, with a difference of only 4% on average. Furthermore, EasyCrash significantly improves application recomputability. The average recomputability of all benchmarks after applying EasyCrash is 82%, while it is 28% before applying EasyCrash. The above observations are consistent with what we have on the Optane DCPMM platform shown in Section 6. Compared with the application recomputability on Optane DCPMM, the application recomputability on the emulated NVM is slightly better (by 5% on average). This is because the emulated NVM has higher memory bandwidth and lower latency than Optane DCPMM, which allows EasyCrash to persist data objects more frequently. In summary, EasyCrash effectively adapts to the hardware changes to improve application recomputability, and shows great potential for future NVM.

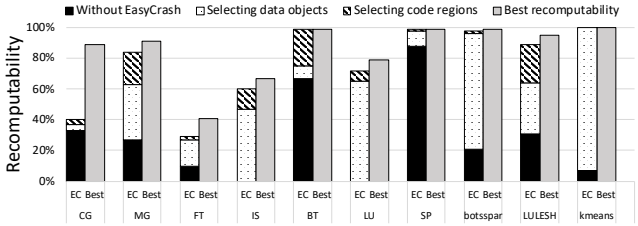



**Figure 13: Application recomputability with different methods. Figure annotation: "EC", "best", and "VFY" stand for EasyCrash, best recomputability, and verified recomputability, respectively.**

**Performance overhead.** We measure runtime overhead incurred by EasyCrash. Table 10 reports execution time of persisting critical data objects for once (i.e., *one persistence operation*) and the number of persistence operations performed by EasyCrash; Table 10 also shows application execution time normalized by execution time without any persistence operation.

Compared with the overhead of persisting *all* candidate data objects at the end of each iteration of the main computation loop (19% on average), EasyCrash is much lightweight. The runtime overhead with EasyCrash is no larger than 2.5% (bounded by $t_s$ = 3%). We also evaluate the overhead of achieving the best recomputability by persisting critical data objects at the end of each iteration of the loop in each code region (i.e., the highest cache flushing frequency). The runtime overhead is 35% on average, which is much larger than EasyCrash.

## A.7 Recomputability Correlation between Small and Large Input Problems

In Section 8, we propose a solution to avoid expensive crash tests and enhance the usability of EasyCrash. The solution is based on the observation that given an application, its recomputability for

**Table 10: 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** | 0.035 s | 40 | 1.012 | 1.26 | 1.24 |
| **FT** | 0.032 s | 80 | 1.016 | 1.22 | 1.12 |
| **IS** | 0.030 s | 10 | 1.011 | 1.15 | 1.43 |
| **BT** | 0.034 s | 200 | 1.025 | 1.10 | 1.34 |
| **SP** | 0.034 s | 200 | 1.022 | 1.23 | 1.55 |
| **LU** | 0.033 s | 250 | 1.025 | 1.23 | 1.58 |
| **botsspar** | 0.030 s | 200 | 1.015 | 1.28 | 1.62 |
| **LULESH** | 0.030 s | 293 | 1.016 | 1.25 | 1.43 |
| **kmeans** | <0.001 s | 36 | 1.000 | 1.00 | 1.00 |
| **Average** | ≈ 0.026 s | 138 | 1.015 | 1.19 | 1.35 |

small and large input problems is similar and correlated. In this section, we present our study that makes the observation.

For each benchmark in our evaluation (Table 1), we try three more input problems (in total, we have four input problems for each benchmark in our evaluation). Those input problems are summarized in Table 11. We persist the same critical data objects at the same code regions for all input problems. For all input problems, we use the same frequency to persist data objects (The frequency is the same as the one used in Appendix A.6.).

Figure 14 shows application recomputability for all input problems. The figure reveals that application recomputability remains stable across input problems. The variance of application recomputability is less than 9%. More importantly, using a small input problem can significantly reduce crash test time for a large input problem. Table 12 shows crash test time, including the time to perform two crash test campaigns (see Section 5.3) for various input problems. We use two machines described in Table 8, each of which uses 48 cores to run crash tests. The crash test times increase as the input problem becomes bigger. Using the input 1, the crash test time is less than 14 minutes for each benchmark, while using the input 3, the crash test time is up to 20 hours. Using the input 1, we significantly reduce crash test time for the input 3 (the input problem used in Section 6) by up to 152 times (68 times on average).

## A.8 Formulation of Calculating $\tau$

$\tau$ is a threshold to guarantee the improvement of system efficiency after using EasyCrash. The improvement of system efficiency ($\Delta_{eff}$) is modeled in Equation 10 based on the definition of system efficiency. In particular, the system efficiency is the ratio of useful computation ($N \times T$) to total execution time ($Total\_Time$). To ensure the improvement of system efficiency with EasyCrash, we must have $\Delta_{eff} > 0$.

$$\Delta_{eff} = \frac{N' \times T'}{Total\_Time} - \frac{N \times T}{Total\_Time} \tag{10}$$

We replace $T'$ and $T$ in Equation 10 with Equations 6 and 8 respectively, and get a new formulation shown in Equation 11.

$$\begin{aligned}\Delta_{eff} = (&(N - N') \times T\_chk + \\ &M \times R_{EasyCrash} \times (T\_r - T'\_r) + \\ &M \times (T\_vain - (1 - R_{EasyCrash}) \times T'\_vain)) \times \frac{1}{Total\_Time}\end{aligned} \tag{11}$$

Since $\Delta_{eff} > 0$, we have Equation 12 based on Equation 11. Equation 12 gives a lower bound of $R_{EasyCrash}$, which is $\tau$.

$$R_{EasyCrash} > \frac{(N' - N) \times T\_chk + M \times (T'\_vain - T\_vain)}{M \times (T\_r - T'\_r + T'\_vain)} \tag{12}$$

## A.9 More End-to-End Evaluation Results for Coordinated Checkpointing

In Section 7, we show system efficiency and scalability for a few benchmarks because of space limitation. In this section, we add results for all benchmarks. Figure 15 shows the system efficiency without and with EasyCrash. EasyCrash improves system efficiency for all benchmarks. On average, EasyCrash improves system efficiency by 2%, 6% and 20% when the checkpointing overhead is 32s, 320s and 3200s respectively. We further evaluate system scalability with and without EasyCrash.

Figure 16 shows the results for scalability evaluation. For all benchmarks, with EasyCrash the system efficiency is always better than without EasyCrash. When the system scale is 400,000 nodes, we achieve the largest improvement in system efficiency: The improvement is 4% and 27% on average, when the checkpointing overhead is 32s and 3200s respectively. Also, the the improvement of system efficiency becomes larger when the system scale becomes larger, demonstrating the effectiveness of EasyCrash.

## A.10 Modeling of Uncoordinated Checkpointing

With uncoordinated checkpointing [38], each process has autonomy to take checkpoints, and there is no synchronization between processes when checkpointing happens. Processes record the dependencies among their checkpoints caused by message exchange during failure-free computation, for the benefit of failure recovery. During a recovery, processes need to iterate to find a consistent state among all checkpoints taken by processes.

Our modeling of uncoordinated checkpointing is an extension of the modeling of coordinated checkpointing in Section 7. The modeling of uncoordinated checkpointing considers the differences between the two checkpointing mechanisms, which are summarized as follows. (1) With uncoordinated checkpointing, the time for writing a checkpoint can he hidden. This means that when one process takes a checkpoint, another process is doing computation, effectively overlapping computation time with checkpointing time. (2) With uncoordinated checkpointing, the recovery time is longer, because of the search of a globally consistent state among all checkpoints.

Based on the above discussion, we make changes to the following model parameters of coordinated checkpointing to model uncoordinated checkpointing. Other parameters remain the same.

**Table 11: Benchmark information for crash experiments. "fp" = "footprint"; "DO" = "data object".**

| Benchmarks | input 1 | | | input 2 | | | input 3 | | | input 4 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | input | mem fp size | critical DO size | input | mem fp size | critical DO size | input | mem fp size | critical DO size | input | mem fp size | critical DO size |
| **CG** | CLASS A | 55MB | 290KB | CLASS B | 398MB | 1.7MB | CLASS C | 947MB | 2.3MB | CLASS D | 9GB | 7.2GB |
| **MG** | CLASS A | 431MB | 310MB | CLASS B | 431MB | 310MB | CLASS C | 3.4GB | 1.2GB | CLASS D | 27GB | 10.8GB |
| **FT** | CLASS A | 321MB | 257MB | CLASS B | 1.25GB | 1GB | CLASS C | 5.1GB | 4.0GB | CLASS D | 80GB | 64GB |
| **IS** | CLASS A | 762MB | 4KB | CLASS B | 1.0GB | 4KB | CLASS C | 1GB | 4KB | CLASS D | 2GB | 8KB |
| **BT** | CLASS A | 815MB | 23MB | CLASS B | 950MB | 91MB | CLASS C | 1.43GB | 361MB | CLASS D | 11GB | 5.6GB |
| **LU** | CLASS A | 820MB | 10MB | CLASS B | 936MB | 41MB | CLASS C | 1.4GB | 164MB | CLASS D | 9GB | 2.5GB |
| **SP** | CLASS A | 827MB | 24.8MB | CLASS B | 965MB | 99MB | CLASS C | 1.47GB | 394MB | CLASS D | 11GB | 6.1GB |
| **botsspar** | m=50, n=25 (test) | 364MB | 1.5MB | m=50, n=100 (train) | 1GB | 191MB | m=120, n=501 (ref) | 3.74GB | 2.45GB GB | m=240, n=100 (-) | 7.8GB | 4.29GB |
| **LELUSH** | s=50 | 870MB | 860KB | s=80 | 1.1GB | 1.5MB | s=100 | 1.41GB | 20MB | s=120 | 1.88GB | 37MB |
| **kmeans** | 819200.txt | 3.1MB | 20B | kdd_cup.txt | 222MB | 20B | 100000_34.txt | 805MB | 20B | 300000_34.txt | 860MB | 20B |

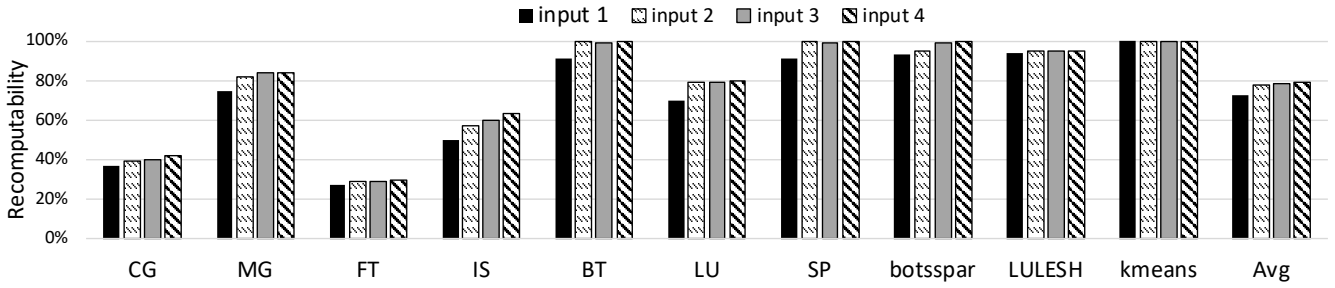



**Figure 14: Application recomputability with different problems as application input.**

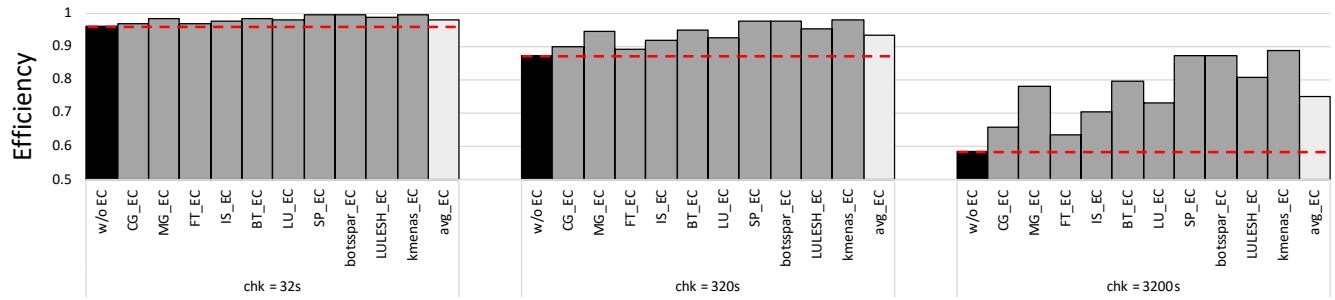



**Figure 15: System efficiency without and with EasyCrash when the system MTBF is 12 hours. The x-axis shows checkpoint overhead.**

- ***T_chk***: The time for writing a system checkpoint. It can be completely hidden (i.e., T_chk = 0), no hidden at all, or somewhere in between. In the worst case, the checkpointing overhead is completely exposed to the critical path of computation. We model the checkpoint overhead as $T\_chk = \alpha \times T\_chk\_co$, where $T\_chk\_co$ is the time for writing a coordinated checkpointing, and $\alpha$ quantifies how much checkpointing overhead is hidden ($0 \leq \alpha \leq 1$).
- ***T_r***: The time for a recovery. This is the search time to identify a globally consistent state, which is dominated by I/O time for reading previous checkpoints until a globally consistent state is found. On average, half of all checkpoints needs to be read to identify a globally consistent state. Hence, $T\_r\_unco = n/2 \times T\_r\_co$, where $T\_r\_unco$ and $T\_r\_co$ are the recovery time for uncoordinated and coordinated checkpointing respectively, and $n$ is the number of checkpoint intervals when the crash happens ($0 \leq n \leq N$ and $N$ is total number of checkpoint intervals without any crash). Similar to the existing work [16, 41], we assume $T\_r\_co = T\_chk\_co$.
- ***T_sync***: The time for synchronization across nodes. $T\_sync = 0$.
- ***T_vain***: The wasted computation time. When the application rolls back to a globally consistent checkpoint, the computation already performed between the checkpoint and crash point is lost. On average, half of useful computation is wasted per recovery (i.e., $T\_vain = n \times T \times 1/2$, where $T$ is the checkpoint interval, and $n$ is the number of checkpoints taken since the application starts.).

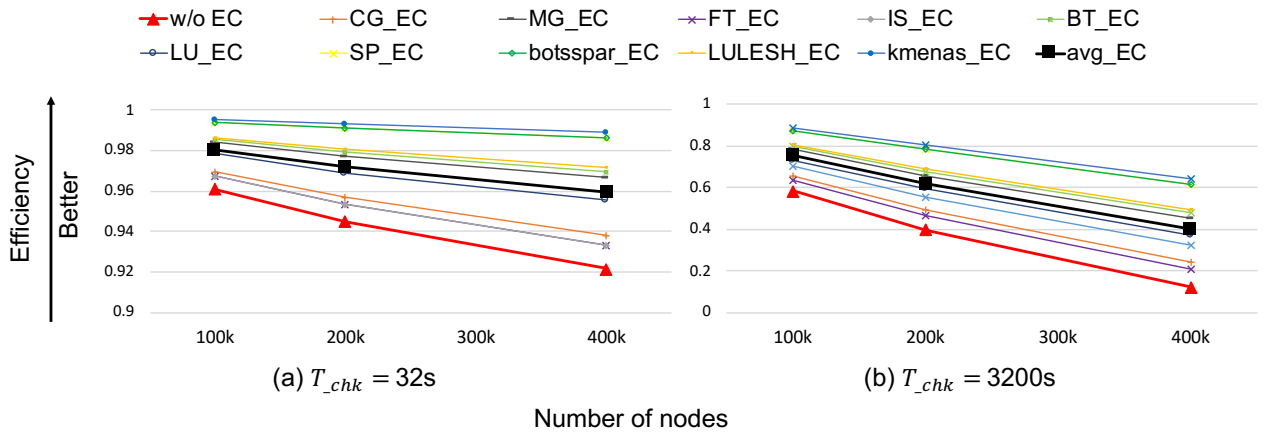



**Figure 16: System efficiency without and with EasyCrash when the system scales from 100k to 200k and 400k nodes.**

**Table 12: Crash test time with NVCT**

| | input 1 | input 2 | input 3 | input 4 |
|---|---|---|---|---|
| **CG** | 4m50s | 29m18s | 275m22s | 710m22s |
| **MG** | 6m30s | 28m14s | 854m46s | 843m21s |
| **FT** | 2m48s | 25m10s | 362m46s | 2362m22s |
| **IS** | 1m42s | 8m36s | 26m58s | 38m34s |
| **BT** | 11m48s | 244m36s | 804m22s | 3204m46s |
| **LU** | 13m42s | 322m48s | 1196m14s | 4602m54s |
| **SP** | 7m10s | 154m46s | 481m26s | 1962m46s |
| **botsspar** | 6m50s | 90m26s | 358m46s | 1222m42s |
| **LULESH** | 5m18s | 21m8s. | 50m18s | 62m34s |
| **kmeans** | 2m46s | 4m26s | 20m34s | 86m42s |

**Choice of parameters.** The choice of $T\_chk$ depends on communication patterns in the application (e.g., how intensively communication happens, and dependencies between message exchanges). In our evaluation, we use 0, $T\_chk\_co$, $1/2T\_chk\_co$ corresponding to the cases of best performance, worst performance of checkpointing, and in between. The choice of $T\_r$ and $T\_vain$ depends on the value of $n$. Given crash randomness, $n$ can be any value between 1 and $N$. But the case of $n = N$ or any large value in the range of $[1, N]$ is rare, because a large-scale system without any crash for a long time is rare. Given the checkpoint interval and typical crash frequency, we empirically decide that $n$ is in the range of $[1, 5]$. Furthermore, for each case of $n$ (we have five cases of $n$), we calculate $T\_r$ and $T\_vain$, and then use the average values of $T\_r$ and $T\_vain$ of the five cases as the final values of $T\_r$ and $T\_vain$. This method is used to ensure that our model is general and representative.

Similar as Section 7, we emulate the system with 100,000 nodes for a long simulation time of 10 years. As shown in previous work [69], such a system usually experiences two failures per day ($MTBF$ = 12 hours).

**Results for system efficiency.** Figure 17 shows the system efficiency with and without EasyCrash with various uncoordinated checkpointing overhead ($T\_chk\_unco = \alpha \times T\_chk\_co$, $\alpha$ = 0, 1/2 ,1 and $T\_chk\_co$ = 32s, 320s, 3200s). We evaluate all benchmarks from Table 1, and we show average recomputability of all benchmarks, as well as two benchmarks with the lowest and highest recomputability (i.e., FT and SP respectively). In general, EasyCrash improves system efficiency by 1% to 60% for uncoordinated checkpointing. Moreover, as $\alpha$ becomes larger, the improvement of system efficiency with EasyCrash is larger. This result is interesting, because a larger $\alpha$ indicates that the overhead of uncoordinated checkpoint is not hidden very well. Using EasyCrash, we can effectively reduce the impact of this overhead on system efficiency.

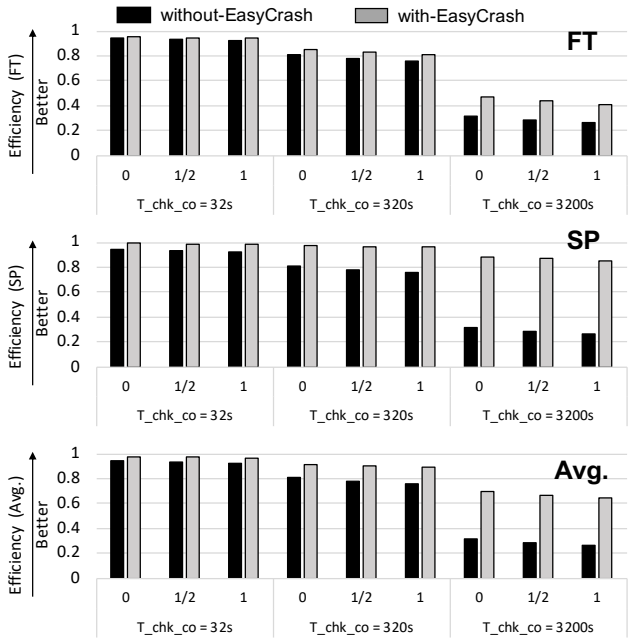



**Figure 17: System efficiency without and with EasyCrash when the system MTBF is 12 hours. The $x$ axis shows different overhead of uncoordinated checkpointing.**

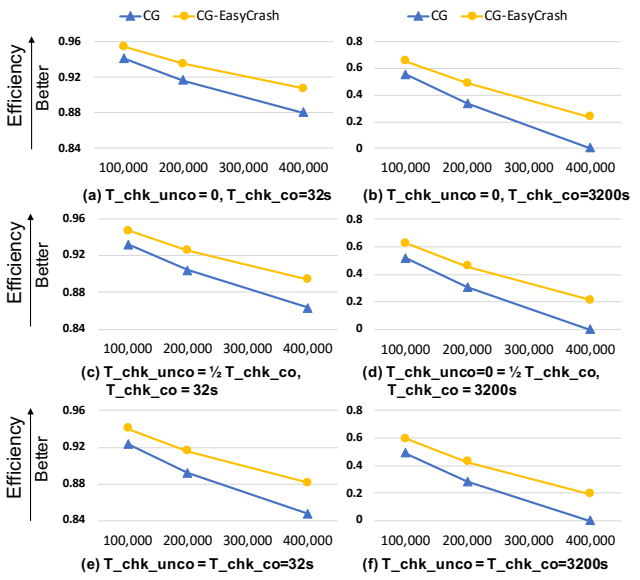



**Figure 18: System efficiency for CG without and with EasyCrash when the system scales from 100,000, 200,000 to 400,000 nodes. Different subfigures show system efficiency with different overheads of uncoordinated checkpointing.**

Compared with using EasyCrash for coordinated checkpointing in Section 7, using EasyCrash for uncoordinated checkpointing causes larger improvement in system efficiency. This is because uncoordinated checkpointing suffers from a larger cost for failure recovery than coordinated checkpointing. Using EasyCrash, the system is able to restart immediately from the crash without rolling back to the last globally consistent state, which significantly reduces recovery cost.

Furthermore, we evaluate the system scalability with EasyCrash and coordinated checkpointing. Figure 18 shows the results for CG, but the performance trend is consistent for all benchmarks: As the system scale is larger, the system with EasyCrash achieves higher efficiency.