Yashme: Detecting Persistency Races

Hamed Gorjiara
University of California, Irvine
U.S.A.
hgorjiar@uci.edu

Guoqing Harry Xu
University of California, Los Angles
U.S.A.
harryxu@cs.ucla.edu

Brian Demsky
University of California, Irvine
U.S.A.
bdemsky@uci.edu

ABSTRACT

Persistent memory (PM) or Non-Volatile Random-Access Memory (NVRAM) hardware such as Intel's Optane memory product promises to transform how programs store and manipulate information. Ensuring that persistent memory programs are crash consistent is a major challenge. We present a novel class of crash consistency bugs for persistent memory programs, which we call *persistency races*. Persistency races can cause non-atomic stores to be made partially persistent. Persistency races arise due to the interaction of standard compiler optimizations with persistent memory semantics.

We present Yashme, the first detector for persistency races. A major challenge is that in order to detect persistency races, the execution must crash in a very narrow window between a store with a persistency race and its corresponding cache flush operation, making it challenging for naïve techniques to be effective. Yashme overcomes this challenge with a novel technique for detecting races in executions that are *prefixes of the pre-crash execution*. This technique enables Yashme to effectively find persistency races even if the injected crashes do not fall into that window. We have evaluated Yashme on a range of persistent memory benchmarks and have found **24 real persistency races** that have never been reported before.

CCS CONCEPTS

• **Hardware** → **Memory and dense storage**; • **Software and its engineering** → **Software verification and validation**; *Compilers*.

KEYWORDS

Persistent Memory, Persistency Race, Debugging, Testing

ACM Reference Format:
Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. 2022. Yashme: Detecting Persistency Races. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3503222.3507766

1 INTRODUCTION

This paper presents a new class of persistent memory bugs, referred to as *persistency races*. Persistency races stem from the fact that most programming language specifications provide compilers with the freedom to assume other threads will not observe a non-atomic store until a synchronization operation. Compilers can for example implement a non-atomic store with multiple store instructions. This is often referred to as *store tearing*. For example, given an architecture having 16-bit store instructions with immediate fields, the compiler might use two 16-bit store-immediate instructions to implement a 32-bit store. Although it is rare that compilers introduce these optimizations, it is enough of a concern that **both PMDK developers and the Linux Kernel developers take care to avoid it**. Indeed, the Linux kernel mailing list provides several examples [10] of modern compilers tearing non-atomic stores even when they are aligned, word-length stores.

Compilers can also introduce store tearing via other optimizations. Mainstream compilers commonly rewrite code that copies or initializes several contiguous fields into calls to the libc functions `memcpy`, `memmove`, or `memset`. These functions do not guarantee 64-bit atomicity and can hence result in store tearing. Store tearing creates the possibility that a poorly timed crash can cause non-atomic stores to be made partially persistent. A post-crash execution can then potentially read values that mix bytes from multiple different store operations. This could for example cause a post-crash execution to read an invalid array index, leading to further corruption.

Store tearing is not the only potential danger of persistency races. Compilers can also stash temporary values in the memory location safely assuming that data race freedom means that other threads will not see these values. Crashes can result in such temporary values being made persistent.

```
1 pmobj->val = 0x1234567812345678;
2 //crash here
3 flush(&pmobj->val);
```

(a) Pre-Crash Code

```
1 if (pmobj->val != 0) {
2   printf("0x%" PRIx64 "\n",
3     pmobj->val);
4 }
```

(b) Post-Crash Code

Figure 1: A persistency race example. We assume `pmobj->val` is initially 0 and both executions are single threaded. `gcc` optimization level `O1` and above generate ARM64 code for this example that can print `0x12345678`. PRIx64 is a macro for printf that prints a 64-bit integer as hex.

Figure 1 presents an example of persistent memory code with a persistency race. Figure 1a and Figure 1b show the code snippets executed before and after the crash, respectively. Suppose that the machine experiences a power failure immediately after line 1 in Figure 1a. Since the store to the `val` field is non-atomic, the compiler is free to implement this store with multiple store instructions. Thus, it is possible for only some of the bytes of this store to be made persistent. When the `val` field is read in line 3 of Figure 1b, the post-crash execution can read a value that is some combination of

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9205-1/22/02.
https://doi.org/10.1145/3503222.3507766

the bytes from the previous value and the newly stored value. This concern is not theoretical—the ARM64 backend of gcc generates code for this program that could print `0x12345678`.

A program has a persistency race if there exist a pre-crash execution E_{pre} and post-crash execution E_{post} such that (1) a load l in E_{post} reads from a non-atomic store s in E_{pre} and (2) the store s is not *persistency ordered* before the load of it in E_{post}. To provide an example of persistency ordering, if the pre-crash execution explicitly flushes a store s before it crashes, then the store s is persistency ordered before any loads that might read from it in the post-crash execution.

Persistency vs. Data Race. Persistency races are similar in spirit to data races because both persistency races and data races violate assumptions made by compilers and thus can break the abstraction of a language-level store writing the specified value to memory. Several tools have been designed to detect data races in code that uses standard lock-based concurrency control [12–14, 21, 40, 52, 61]. These tools generally take one of two approaches: (1) they verify that all accesses to shared data are protected by a locking discipline or (2) they compute a happens-before relation to detect concurrent conflicting accesses.

While persistency races are similar to data races, there are important fundamental differences between the two as each persistency race involves three distinct events: (1) the racing store in the pre-crash execution, (2) the crash event against which the store races, and (3) a race-observing load in the post-crash execution that observes the effects of the race. This differs from data races that consist of two memory operations that race against each other. Persistency races exist even in single-threaded programs. Intuitively, in a persistency race, a pre-crash execution thread races with the crash and a post-crash thread observes the effects of the race.

Researchers have developed techniques for detecting races in interrupt-based code [7, 55]. That body of work focuses on ensuring that interrupts are disabled when code performs a memory access that could potentially conflict with memory accesses in an interrupt handler. The focus on analyzing interrupt code that relies on disabling interrupts means that the analysis techniques are not applicable to persistent memory where a crash can occur at any point.

Most existing PM bug finding tools use techniques that fundamentally cannot detect persistency races because they just validate that stores are flushed or performed in a specific order. The one exception is that model checking tools could conceptually be adapted to find persistency races by splitting the stores into single byte stores at the cost of an exponential increase in the number of executions that must be explored. Dynamic instrumentation frameworks [11, 28, 31, 37] can observe actual store instructions in the binary. The primary challenge in detecting persistency races with these frameworks is that they cannot infer whether two stores in the binary were originally one source-level store nor can they infer which stores were atomic at the source level, thus it is not possible for these tools to directly detect a persistency race. However, if these frameworks explore the correct execution, they can potentially observe a crash caused by a persistency race, *e.g.*, a segmentation fault caused by accessing a partially persisted pointer. Moreover, these tools can give no warning for stores that could potentially be torn in the future. XFDetector [37] uses a finite state machine to track the consistency and persistency of persistent data. XFDetector finds cross-failure races, which are defined as loads that read from locations that were not persisted before a failure. Cross failure races are different from persistency races in that cross failure races model normal stores as effectively atomic and do not consider the possibility that due to compiler optimizations a store may made partially persistent. Cross failure race detection cannot detect persistency races because it does not model the effects of cache coherence or the difference between atomic and normal memory operations. XFDetector is limited to detecting cross failure races in the given execution and cannot detect cross failure races in any other potential executions.

There is a large body of work on finding atomicity violation bugs [27, 39, 47, 60] which are fundamentally different from persistency races. Atomicity violations occur when code written by developers performs many operations that were intended to be atomic, but are not atomic because of a bug. However, persistency races violate language-level store abstractions because of a race with a crash event.

Yashme. Detecting persistency races is extremely challenging as it requires reasoning about the cache behavior of a program, *e.g.*, where the crash occurs, where a cache line is flushed, and which pre-execution store the post-execution code reads from. On one hand, data race detectors focus on reasoning about mutual exclusions, *not* on cache behavior. As a result, none of the data race detection algorithms are directly applicable to detecting persistency races. On the other hand, existing persistent memory bug detectors reason about the timing of cache line flushes, but rely on effective test cases and appropriate crash events to find bugs. Detecting a persistency race with respect to a store, however, requires the crash to fall into a small window of execution after the store and before the (explicit or implicit) cache line flush. Such a strict requirement dictates aggressively injecting crash events in a great number of executions and exhaustively exploring thread schedules, which is impractical.

Model checkers (*e.g.*, Yat [31] and Jaaru [18]) and fuzzers (*e.g.*, PMFuzz [36]) automatically explore many possible cache states, but their effectiveness depends on the selected thread schedule—they would not detect a persistency race if the thread schedule causes the window to close; as such, they suffer from the same weakness as other bug detectors.

Clearly, a major challenge in devising an effective persistency race detector is where to inject the crash event, *i.e.*, the nature of a persistency race is that the pre-crash event *races* with the crash event.

Key Insight. Key to the success of Yashme is a shift of focus from generating race-manifesting crash events to *generalizing/expanding the executions under a small number of crash events to make races observable*. In other words, Yashme does *not* rely on perfect crash events to trigger races; instead, given a target crash event, Yashme runs the program, obtains its pre-crash execution, and *expands* it to derive many executions that can also be used to detect races.

Yashme's approach is similar to model checking in that both derive many executions from existing ones. However, Yashme expands an existing pre-crash execution using a set of *prefix constraints*, which guarantee that the derived executions are *consistent* with the original execution with respect to the post-crash execution. In particular, if the post-crash execution reads from a store in the original pre-crash execution e, it must read from the same store in any executions derived from e. Expansion enables Yashme to detect races in derived executions without actually executing them, while model checkers incur the overhead of actually executing their derived executions.

In other words, any execution that shares a common prefix (starting at the store) with e and does not later perform and persist stores that overwrite locations read by the post-crash execution is consistent and can be used to detect persistency races. Prefix-based derivation significantly expands the race-detection scope, enabling Yashme to find more bugs even than a technique that injects a crash event before every fence instruction. As a result, Yashme has found persistency bugs in **all but one of the programs we have experimented with**.

Results. We have implemented Yashme with a full simulation of Px86_{sim} semantics and applied it on widely-used persistent memory programs including RECIPE persistent memory indexes; the PMDK library; Memcached, a high-performance cache server; and Redis, a persistent memory data store. Yashme found a total of 24 persistency bugs in every single program.

2 BACKGROUND

We next briefly overview the Intel-x86 persistent storage system following the formalized Px86_{sim} model in Raad *et al.* [49]. The Px86sim semantics capture the behavior Intel implemented in silicon and intended for the architecture. They differ slightly from the semantics presented in Intel's manual due to mistakes in precisely specifying the intended behavior in the documentation. Figure 2 presents a graphical overview of the x86-TSO storage system. Each core/thread on x86 has a store buffer that buffers stores to the cache to hide the store latency. The store buffers implement bypassing — when a core performs a load, the core checks whether there is a store to the same address in its local store buffer. If so, it returns the value written by the most recent such store. Effectively, this allows the local core to observe the effect of a local store before that store becomes visible to other cores. The memory fence instruction `mfence` clears the store buffer before future instructions can be executed. Locked `RMW` instructions also clear the store buffer before future instructions can be executed.

Stores in the store buffer are written to the cache in order. The stores update the cache in a total order across all cores and all threads/cores observe these stores in that same order. The cache is volatile—a power loss event will cause cached data that has not been written back to persistent storage to be lost. Cache lines are written back to main memory non-deterministically when the cache needs the space for other data. The x86 architecture provides instructions to force the cache to write data back to persistent storage. The three such instructions are: (1) the flush cache line instruction `clflush` that flushes a cache line, (2) the optimized flush cache line instruction `clflushopt`, and (3) the cache line write back instruction `clwb`. Each of these instructions takes as input the address of the cache line and flushes that line.

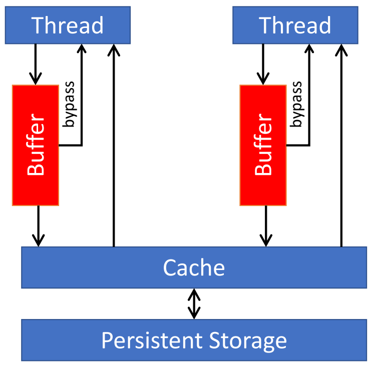

Figure 2: An x86-TSO storage system.

Table 1: Reordering constraints in the Px86_{sim}. A ✓ indicates that the order between the two instructions is preserved, a ✗ indicates that the two instructions can be reordered, and a CL indicates that the order is preserved only if they both operate on the same cache line. 'mf', 'sf', 'clfopt', and 'clf' represent 'mfence', 'sfence', 'clflushopt', and 'clflush', respectively.

		Later in Program Order						
		Re	Wr	RMW	mf	sf	clfopt	clf
Earlier in Program Order	Read	✓	✓	✓	✓	✓	✓	✓
	Write	✗	✓	✓	✓	✓	CL	✓
	RMW	✓	✓	✓	✓	✓	✓	✓
	mfence	✓	✓	✓	✓	✓	✓	✓
	sfence	✗	✓	✓	✓	✓	✓	✓
	clflushopt	✗	✗	✓	✓	✓	✗	CL
	clflush	✗	✓	✓	✓	✓	CL	✓

A key difference between these instructions is how they can be reordered across other instructions. Table 1 summarizes the instruction ordering constraints for persistent storage on x86-TSO. The `clflush` instruction is inserted into the store buffer just like store instructions, and when it exits the store buffer it causes the cache line to be flushed to persistent memory. The `clflushopt` instruction is inserted into the store buffer also like store instructions, but it can be reordered across store instructions to other cache lines, `clflush` instructions to other cache lines, and other `clflushopt` instructions. The `clflushopt` instruction cannot be reordered across `mfence` or locked `RMW` instructions. The store fence instruction `sfence` also orders `clflushopt` instructions relative to `clflush`, `clflushopt`, `clwb`, and store instructions. The `clwb` instruction only writes back the contents of the cache line and does not evict it from the cache and thus has better performance. However, from a semantic perspective, the `clwb` instruction is identical to `clflushopt` instruction [49], and thus we treat them identically in our discussions.

```
int Segment::Insert(Key_t& key, Value_t value,
  size_t loc, size_t key_hash) {
  ...
  if (CAS(&_[slot].key, &LOCK, SENTINEL))
  {
    _[slot].value = value;
    mfence();
    _[slot].key = key;
    ret = 0;
    break;
  }
  ...
}
```

Figure 3: The `Segment::Insert` method from the `CCEH` hashtable. The store to the `key` field commits an insertion into the table. This store is non-atomic and thus a poorly timed crash could cause the key to be partially written.

3 MOTIVATION

The correctness of crash consistent data structures rests on careful analysis of the ordering of operations to reason about the potential intermediate states that a crash can leave a data structure in. Applying this type of reasoning to non-atomic memory operations is problematic due to compiler optimizations. Compilers perform optimizations assuming that programs are race-free—other threads (or post-crash executions) will *not* observe updates to the states of non-atomic shared variables until a release operation, *e.g.*, an unlock, is performed. For example, a compiler may implement a non-atomic store using multiple store instructions (*i.e.*, store tearing) or even generate new store instructions (*i.e.*, store inventing) to temporarily stash intermediate results, *e.g.*, if the compiler runs out of registers to store temporary values.

3.1 Example Persistency Race

To provide a concrete example, we examined the source code of the Cacheline-Conscious Extendible Hashing (CCEH) hashtable [45], which is distributed with the RECIPE suite of persistent memory indexes [32]. Figure 3 presents the `Insert` procedure. It uses a CAS on the `key` field to lock a slot in the hashtable. When the slot is locked, it first writes the `value` field and then the `key` field. This design relies on the fact that both `value` and `key` fields reside on the same cache line to ensure that the store to the `value` field persists before the store to the `key` field. Once the `key` field is written, the key-value value insertion has been committed to the table. The caller of this procedure later flushes both stores to persistent memory.

The problem with this implementation is that since the store to the `key` field is non-atomic, the compiler is free to implement this store with multiple store instructions. While we might imagine this would only occur in cases where the `key` field is not aligned or does not match the native word size of the machine, there are examples of modern compilers implementing such aligned, word-size stores using multiple store instructions [10]. Hence, a crash could potentially cause an incorrect key to be inserted into the table. To fix this bug, the developer should implement the store of the `key` field using an atomic store operation. On an x86 processor, this fix would incur minimal overhead—the atomic store can still be compiled into a normal store instruction as long as the compiler is prevented from performing problematic optimizations (such as store tearing).

3.2 Severity of Persistency Races

Ubiquity. The conventional wisdom in concurrent programming is for developers to (1) use locks to protect critical sections and (2) only use atomic operations when strictly necessary for performance or progress guarantees. Applying such practices to persistent memory programs inevitably leads to implementations with persistency races. Developing race-free persistent data structures requires extensive use of atomic operations or other techniques like *checksums*.

In our experiments with the RECIPE [32] persistent benchmark suite, we found a total of 19 persistency races in the persistent memory indexes that we were able to execute. [1]

Compilers Performing Store Optimizations. Since persistency races hinge upon certain store optimizations performed by a compiler, we have conducted a study of recent versions of `gcc` (version 10.3) and `LLVM-clang` (version 11.0), two widely-used compilers for native code, with a goal to understand how common these optimizations are on different architectures. As reported in Table 2a, store optimizations are widely used by both `gcc` and `clang` on both ARM and x86 architectures. Whether such optimizations are applied to a particular program depends on the implementations of compilers and libraries (such as `memmove` and `memcpy`).

Empirical Validation. To understand the importance of this issue, we carried out a study on a collection of data structures [22, 32, 45]. We compiled each data structure with `clang` version 11.0 with the `-O3` optimization level. Table 2b compares the number of different memory operations (*i.e.*, `memset`, `memcpy`, and `memmove`) that appear in the source code with the number of them that appear in the assembly code. For all programs except P-ART and P-CLHT, the assembly code contains more memory operations than the source code, showing that the compiler has replaced normal stores with memory operations to optimize the code. We carefully audited both P-ART and P-CLHT programs to understand why they do not report more memory operations in their assembly code. For P-ART, it turns out the source code uses 14 `memset` operations inefficiently in the constructors. The compiler optimized them into 3 `memset` and replaced different normal write operations in the source code with 2 `memcpy`. For P-CLHT, we observed that this program uses a lock-free design and critical store operations are defined as volatile and the compiler did not optimize them with memory operations.

The transformation of normal stores into function calls to `memcpy` and `memset` is very common and disabling this optimization in the compiler would likely have significant performance penalties. Compiler optimizations are prone to persistency races not only because of store tearing but also due to store inventing [26]. Compilers can legally invent stores to memory locations that code is guaranteed to write to, *e.g.*, the compiler could generate new store instructions to temporarily stash intermediate results if the compiler runs out of registers to store temporary values. Thus, fixing persistency races requires restricting legal optimizations and adding constraints to memory operation calls. While this might seem an easy solution,

[1] *i.e.*, CCEH, Fast&Fair, and Recipe benchmarks except P-HOT. We were not able to execute the P-HOT because it did not compile with LLVM.

Table 2: Ubiquity of persistency races.

(a) Summary of popular compilers and observed store optimizations that can lead to persistency races.

Compiler	Arch	Store Optimizations
gcc	ARM64	Use a non-atomic pair of stores for a 64-bit store
gcc & LLVM-clang	ARM64	Replace a seq. of stores of zero with a `memset`
gcc & LLVM-clang	ARM64	Replace a seq. of assignments with a `memmove` or `memcpy`
LLVM-clang	x86-64	Replace a seq. of stores of zero with a `memset`
LLVM-clang	x86-64	Replace a seq. of assignments with a `memcpy`
gcc	x86-64	Replace a seq. of assignments with a `memmove`

(b) Number of memory operations (*i.e.*,`memset`, `memcpy`, `memmove`) used in source code of Fast_Fair, CCEH, and RECIPE benchmarks compared to number in the assembly code generated by `clang` version 11.0 with the `-O3` option.

Prog	#src-op	#asm-op
CCEH	6	33
Fast_Fair	1	4
P-ART	17	8
P-BwTree	6	15
P-CLHT	0	0
P-Masstree	3	14

in practice it can be extremely challenging since all of these optimizations are currently standards compliant and thus ensuring safety would *require revising the C/C++ language standard*. This process would likely take many years and there is no guarantee that the standards committee would not simply decide that developers should simply use atomics to avoid persistency races. In addition, there would be a wait for any changes to get rolled out to compilers and libraries.

Although persistency races may not manifest under a particular compiler/architecture, they can lead to bugs that are extremely difficult to detect. For example, a library or compiler update may expose a latent persistency race in recovery code, triggering cascading bugs and even system-wide failures. Such failures can lead to disasters in mission-critical systems—*e.g.*, upgrading the compiler can expose a latent bug in a storage system, causing complete loss of data. As such, there is a pressing need to find and fix such bugs *early on* before they manifest. In fact, the (Intel) developers of PMDK have confirmed [19] that "we do try to ensure that those issues are not present in PMDK and we do extensive validation on compiled binaries to that end; but something can always slip through the cracks; and we definitely don't want to depend on compiler-defined behavior if we can avoid that."

4 YASHME OVERVIEW

This section presents our basic idea and an overview of how Yashme finds persistency races. We focus our presentation of persistency races on the x86-TSO persistency model. However, persistency races are more general than x86-TSO and will be applicable on other hardware and software persistency models.

Yashme is focused on finding whether an application or library has code for which the language standard permits the compiler to generate code that exhibits a persistency race. Thus, Yashme instruments the LLVM Intermediate Representation (IR) to call into the Yashme library that simulates the x86-TSO persistency model and monitors for persistency races.

Yashme has two modes of operation: (1) model checking and (2) random execution. In the model checking mode, Yashme explores all executions to find persistency races. This mode is suitable for programs that are relatively small. For large programs for which it is time-consuming to explore all possible executions, Yashme can operate in random mode to detect persistency races.

Yashme's basic approach is to simulate the execution of a PM program, inject a crash, and then simulate the execution of the post-crash recovery program. During the post-crash execution, we compute *which stores may have persisted incorrect values due to the crash*. There are two ways that PM program executions can ensure that stores are fully persisted: (1) the execution explicitly flushes the cache line *after* the store writes to the cache line and *before* the crash to ensure that the stored value was persisted or (2) the post-crash execution reads from a later atomic store to the same cache line and relies on *cache coherence* to ensure the persistency of the store.

Section 4.1 presents our approach to handling (1) explicit flushes and (2) cache coherency to determine whether stores are fully persisted. Section 4.2 then presents how Yashme leverages the idea of *execution prefixes* to maximize the persistency races that can be detected at a given injected crash.

4.1 Basics

Flush Operations. Flush operations can be used to force a cache line to be persisted after a store is fully completed. Without an appropriate flush operation, a non-atomic store can be *partially persisted*. Thus accurately modeling the effects of cache line flush operations is critical for detecting persistency races. We first describe how Yashme models `clflush`.

Figure 4(a) presents an example of using the `clflush` instruction to flush a cache line. In this example, the pre-crash execution stores 1 to the variable `x`, and then persists this store by executing a `clflush` instruction. To ensure that a `clflush` instruction persists a store, it is critical that the store *happens before* the `clflush` instruction. Although store s is non-atomic, this execution does not expose a persistency race because s has been flushed before the crash (although other executions can expose a persistency race).

We next discuss how Yashme tracks whether stores have been persisted using the `clflush` instruction. Yashme assigns each operation an increasing clock σ that uniquely identifies the operation. Yashme tracks which stores have been flushed by building a map `flushmap`, which maps the clock of each store s to a pair of the

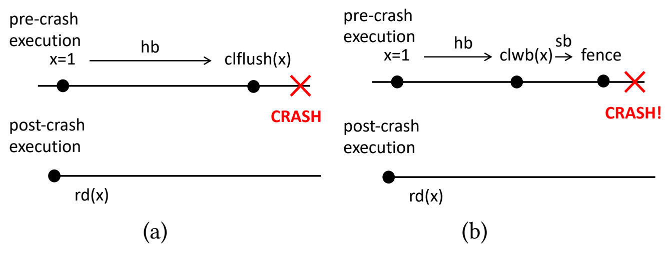

Figure 4: (a) Example of using `clflush` to flush the store to `x`. (b) Example of using `clwb` to flush the store to `x`.

form $\langle\tau, \sigma\rangle$, where τ is the identifier of the thread that performs a `clflush` that happens after s, and σ represents the clock that labels that `clflush` such that there is no other `clflush` that is ordered between the store s and this `clflush` by happens before. To build this map, when a `clflush` instruction takes effect on the cache, Yashme updates `flushmap` for the latest store to each memory location to include the thread that executed the `clflush` and the clock of the `clflush`. When the post-crash execution reads from `x`, Yashme determines that since `flushmap`($\sigma_{\texttt{x=1}}$) (*i.e.*, `flushmap` applied to the timestamp of the store `x=1`) is not empty, there is no persistency race.

The x86 architecture provides a second, more efficient cache line flush mechanism: the cache line write back instruction `clwb`. Figure 4(b) presents an example in which the pre-crash execution stores 1 to the variable `x`, and then persists this store by executing a `clwb` instruction and a fence instruction. To persist the store, it is critical that (1) the store happens before the corresponding `clwb` instruction and (2) the thread that executes `clwb` also executes a fence instruction later. In our example, although the store `x=1` is non-atomic, this execution does not have a persistency race because `x=1` has been flushed before the crash.

We next extend our approach to track whether stores have been persisted using the `clwb` instruction. Yashme maintains a per-thread set F_τ of `clwb` instructions that have *not* been followed by a fence. When a thread τ executes a fence instruction, Yashme processes each of the `clwb` instructions in F_τ for the thread. When a `clwb` instruction takes effect on the cache, Yashme updates the `flushmap` for the latest store to each memory location (if the store happens before the `clwb` instruction) to include the thread that executed the fence instruction and the sequence number of the fence. Similarly, when the post-crash execution reads from `x`, Yashme determines that `flushmap`($\sigma_{\texttt{x=1}}$) is not empty and hence there is no persistency race.

Cache Coherence. Cache coherence protocols ensure a total order in the persistence of stores to the same cache line. Figure 5(a) provides an example of an execution that uses cache coherence to avoid a persistency race. Assume that the variables `x` and `y` reside on the same cache line. We use the notation $\texttt{y}_{\texttt{rel}}\texttt{=1}$ to indicate that the store of 1 to `y` is an atomic release store. In the pre-crash execution, the store to `x` happens before the store to `y`. Since `x` and `y` are on the same cache line, cache coherence protocols guarantee `x=1` is completely written to the cache line before $\texttt{y}_{\texttt{rel}}\texttt{=1}$ even if store to `x` is torn into multiple store operations. Since the post-crash execution observes the store to `y`, the cache line is flushed sometime after persisting $\texttt{y}_{\texttt{rel}}\texttt{=1}$ and before the crash event. Consequently, the post-crash execution must also observe the fully completed store to `x` due to cache coherence. Thus, there is no persistency race in this execution.

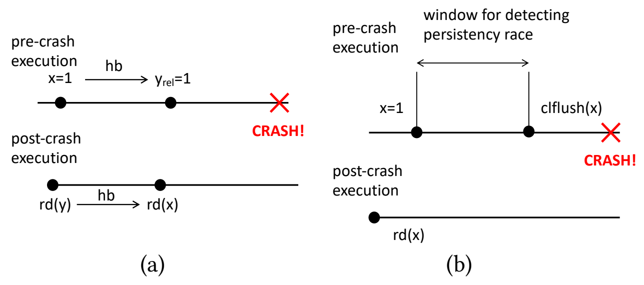

Figure 5: (a) Example of coherence preventing persistency races. Assume that the variables `x` and `y` reside on the same cache line and that the store to `y` is an atomic release store. (b) Crash misses window for detecting persistency race using core algorithm.

To model the effect of cache coherence, Yashme maintains a map `lastflush` that maps each cache line to a clock vector that represents the earliest point in the pre-crash execution where the cache line could have been written back to persistent memory—any stores that happen before this point must have been fully persisted. When the post-crash execution reads from an atomic store $\texttt{y}_{\texttt{rel}}\texttt{=1}$, Yashme updates the `lastflush` map to indicate that the cache line must have been written back some time after the atomic store to `y`. When the post-crash execution reads from the non-atomic store to `x`, Yashme uses the `lastflush` map to determine there is no persistency race because the cache line must have been persisted after the store to `x` was completed. More technical details about the basic algorithm are described in §6.

4.2 Key Idea: Expanding the Detection Window

Our core approach is to randomly inject crash events and use the aforementioned maps to determine the existence of a persistency race. In particular, we can use the `flushmap` map to detect whether a store was fully persisted via a flush operation and the `lastflush` map to determine whether a store must have been fully persisted because the post-crash execution has read from a later store to the same cache line. However, this approach can only detect persistency races involving stores in *a small window of the pre-crash execution.* In practice, persistent memory programs often flush stores in a timely manner. Hence, detecting a persistency race from a given store requires the crash point to fall into the window between the store and the explicit flushing of its corresponding cache line.

Here we present how we optimize the core approach to improve its ability to detect persistency races. Figure 5(b) shows an example crash scenario to illustrate this problem. In this example, the pre-crash execution writes to the variable `x`, flushes the write, and then crashes. The post-crash execution then reads from `x`. Since the crash occurs after the write is flushed, the approach misses detecting the persistency race in this program. To detect this persistency race, the program must crash in the small window of time between the store to `x` and the corresponding flush. This implies that detecting races using the approach would require injecting crashes in a large number of executions, which can be prohibitively expensive for large programs.

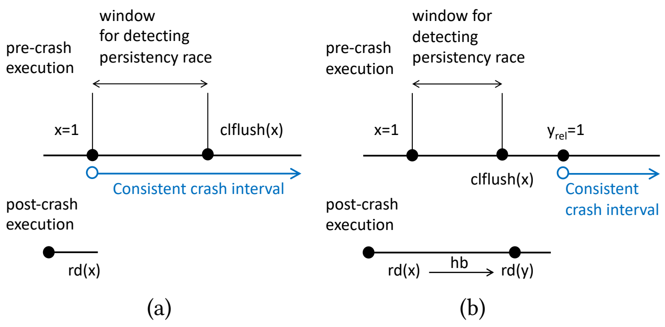

Figure 6: (a) Prefixes of pre-crash execution that are consistent with the post-crash execution (b) Prefixes of pre-crash execution that are consistent with the post-crash execution after reading from y residing on the same cache line as x.

Our key insight for effectively detecting persistency races is that we can check whether the post-crash execution E' has a persistency race with *any prefix* E^+ *of the pre-crash execution* E *that is consistent with* E'. Figure 6(a) illustrates this insight. While the pre-crash execution has flushed the store to x, the post-crash execution has not read from any store that happens after the cache line flush. Thus the post-crash execution at this point is *consistent with any prefix* of the pre-crash execution starting at the store that writes 1 to x. The blue arrow shows the range of consistent prefixes of the pre-crash execution. In Figure 6(b), as the post-crash execution reads from the atomic variable y, Yashme updates the constraint of pre-crash execution to be *consistent* with the post-crash execution and to include the `clflush(x)` instruction.

Consistent Prefixes. Intuitively, a consistent prefix of the pre-crash execution must contain any statement which happens before a pre-crash store that the post-crash execution reads from. Yashme tracks every pre-crash store that the post-crash executions reads from and computes the shortest consistent prefix by using clock-vector-based techniques [14] that are commonly used by race detectors. Yashme uses the consistent prefix to determine whether there is a prefix of the pre-crash execution that did not execute a given `clflush`, `clwb`, or fence instruction. If so, Yashme ignores the instruction when checking for races, because there is a pre-crash execution that does not execute the instruction and yields the same post-crash execution. For example, in Figure 6(a), there is a prefix of the pre-crash execution which does not execute `clflush (x)`. Thus, Yashme can ignore this instruction. However, after reading y in Figure 6(b), `clflush(x)` must be executed in all prefixes of the pre-crash execution and cannot be ignored anymore. The constraint prefix helps Yashme find persistency races even when the crash event is inserted outside of the detection window in the model-checking mode or random mode.

Multi-threaded Programs. *Yashme fully supports multi-threaded programs.* For multithreaded executions, we use a per-thread prefix of the execution. Note that in the multithreaded case the prefix-based approach can detect persistency races in executions that cannot be generated by inserting a crash event at any point in the pre-crash execution. For example, consider a pre-crash execution in which thread 1 performs a racy store to z, flushes z, then thread 2 sets an atomic flag `f` to `true` and a post-crash execution that reads from z and if `f` is `true` then reads from z. There is no point in the pre-crash execution trace that we can insert a crash to observe the race in this code. The prefix analysis can determine that the post-crash execution has not read from any store that happens after the flush of z, and therefore we can rearrange the pre-crash execution to a race revealing execution in which thread 1 performs a racy store to z, thread 2 sets an atomic flag `f` to `true`, and then it crashes.

5 ALGORITHM PRELIMINARIES

We begin by formalizing our notion of a persistency race. Intuitively, a persistency race occurs if an execution reads from a non-atomic store that could have been made partially persistent. Definition 5.1 presents our definition for whether an execution contains a persistency race.

Definition 5.1. A load l in a post-crash execution E' that reads from a store s in a pre-crash execution E is a persistency race if (1) s is not atomic, (2) there is no atomic release store s' to the same cache line as s in E such that $s \xrightarrow{hb} s'$ and E' reads from s' before it reads from s, (3) there is no cache-line flush `clflush` to the same cache line as s such that $s \xrightarrow{hb}$ `clflush`, and (4) there is no cache-line flush `clwb` to the same cache line as s and a fence `fence` such that $s \xrightarrow{hb}$ `clwb` and `clwb` $\xrightarrow{sb}$ `fence`.

The first condition ensures that the compiler could legally implement the store with several store instructions or insert other store instructions to the same memory address. The second condition ensures that there were no later atomic stores to the same cache line that were read by the post-crash execution and thus the cache line must have been written back after store s completed. The third and fourth conditions ensure that there was no cache line flush that forced the CPU to flush the entire cache line and thus the processor would have been free to flush the cache line when the store was partially completed or after a compiler inserted store.

5.1 Persistency Races in Execution Prefixes

Our key insight for effectively detecting persistency races is that we can check whether the post-crash execution E' has a persistency race with any prefix E^+ of the pre-crash execution E that is consistent with E'.

Consistent Prefixes. For each store s in the pre-crash execution E that some load in the post-crash execution E' reads from, Yashme computes the prefix E_s of the pre-crash execution that happens-before s. Since the post-crash execution E' has observed s, it is only consistent with prefixes of the pre-crash execution that include E_s. Yashme computes E^+ as the union of the sets E_s that correspond to each pre-crash store s that the post-crash execution reads from. This union is efficiently computed using clock vector techniques that are commonly used by race detectors. The prefix E^+ is the smallest prefix of E that is consistent with the post-crash execution.

Yashme uses E^+ to compute whether a store must have been made persistent in all prefixes of E that are consistent with E'. There are two situations in which the post-crash execution must observe s as fully persisted: (1) s was flushed to persistent memory in E^+ or (2) the execution E' has already observed a later store to the same cache line.

Under x86 TSO, recall that there are two ways to flush a cache line after a store s. The program can execute the `clflush` instruction or it can execute the `clwb` instruction followed by a fence. Yashme checks whether there is a `clflush` or `clwb` instruction in E^+ to the same cache line written by the store s that happens after s. If there is a `clflush` instruction, the store s has been made persistent. If there is a `clwb` instruction, Yashme must also check whether the thread that executed the `clwb` instruction later executed an instruction in E^+ with fence semantics. If so, the store s was fully persisted.

If the cache line was not explicitly flushed, Yashme checks whether the post-crash execution has already observed a later store to the same cache line. If not, Yashme reports a persistency race and the pre-crash execution prefix E^+ combined with the post-crash execution E' as a witness.

Theorem 1 (Persistency Race in Execution Prefixes)**.** Given a pre-crash execution E and a post-crash execution E', there exists a race-revealing pre-crash execution E^+ that has a persistency race with E' if there is a load l in E' that reads from a store s in E and all of the following conditions hold:

(1) s is not atomic.
(2) $\nexists s' \in E$, *such that* s' is an atomic release store, and $s \xrightarrow{hb} s' \wedge \texttt{samecacheline}(s, s') \wedge$ E' reads from s' before it reads from s
(3) $\nexists \texttt{clflush} \in E$ *such that* $s' \in E, s \xrightarrow{hb} \texttt{clflush} \wedge \texttt{samecacheline}(s, \texttt{clflush}) \wedge \texttt{clflush} \xrightarrow{hb} s' \wedge$ E' reads from s'
(4) $\nexists \texttt{clwb}, \texttt{fence} \in E$ *such that* $s' \in E, s \xrightarrow{hb} \texttt{clwb} \wedge \texttt{samecacheline}(s, \texttt{clwb}) \wedge \texttt{clwb} \xrightarrow{sb} \texttt{fence} \wedge \texttt{fence} \xrightarrow{hb} s' \wedge E'$ reads from s'

Proof. Assume that we have a pre-crash execution E and post-crash execution E' where a load in E' reads from a store s in E that satisfies the conditions 1 through 4.

We define the execution E^+ to include all statements in E that happen before the stores in E that are read by E' and those stores. We next show that E^+ has a persistency race with the post-crash execution E'.

The store s in E^+ is non-atomic by the assumed condition 1 and thus satisfies condition 1 from Definition 5.1. The store s in E^+ satisfies condition 2 by assumption and thus satisfies condition 2 from Definition 5.1. By assumption, condition 3 is true for the store s in E. Since E^+ by construction only contains events from E that happen before some store s' that E' reads from, there cannot be a `clflush` in E^+ to the same cache line as s that is ordered after s. Therefore, s in E^+ satisfies condition 3 of Definition 5.1. By the same argument, the store s in E^+ satisfies condition 4 from Definition 5.1. Therefore, the execution E^+ has a race with E'. □

6 RACE DETECTION ALGORITHM

We next present our persistency race detection algorithm. We begin by presenting the following notations that we will use throughout the paper:

- We refer to an execution as e.
- We denote a thread using $\tau \in \mathcal{T}$.
- Each thread τ has a store buffer S_τ that keeps a queue of store, `clflush`, and `sfence` operations that have not yet taken effect on the cache.
- Each thread τ has a cache line flush buffer F_τ that stores the set of `clwb` operations that have not yet flushed the cache line to persistent storage.
- A given failure scenario may involve a sequence of multiple executions ending in failures. For example, a persistency race in the recovery procedure would require two crashes: one to get into the recovery procedure and a second to reveal a bug in the recovery procedure. We record this sequence of executions that have been executed on the persistent store using a stack, referred to as *exec*.
- Function `top`(*exec*) denotes the most recent execution (the current one) on the stack *exec*.
- Function `prev`(e) returns the execution that immediately precedes e in *exec*.
- A global sequence number counter σ_{curr} is used to assign increasing sequence numbers to stores, `clflush`, and `sfence` instructions. Each store, `clflush`, and `sfence` instruction i is assigned a sequence number σ_i. These numbers record the total order in which these instructions take effect in the cache. Using a global sequence number has no performance drawbacks since Yashme already determines the interleaving of threads and has full control over the scheduling of all memory operations.
- Each execution e has a map e.`storemap` that maps each address *addr* to the thread τ and sequence number σ generated at the moment that value was stored.
- Each e has a map e.`flushmap` that maps a store's sequence number σ to a set of pairs $\langle \tau, \sigma \rangle$ of a τ and the sequence number σ of the first flush it performed after the store.
- Each e has a map e.`lastflush` that maps a cache line identifier to a clock vector that is a lower bound for when the cache line was written back.
- Each e has a clock vector $e.\text{CV}_{\text{pre}}$ that records how much of the execution e that later executions have observed.

The TSO memory model separates the executions of stores, cache flush operations, and `sfence` operations into two phases: (1) the initial phase that often inserts an operation into a buffer and (2) the second phase that removes the instruction from the buffer and updates the state of the cache or persistent storage. We present our algorithm for each of the stages.

Executing instructions. Figure 7 presents our algorithm for the first phase of instruction execution, which inserts an instruction into each thread's local store buffer S_τ. The `mfence` instruction waits until S_τ is empty and then clears the thread's flush buffer F_τ. RMW instructions also have `mfence` like semantics and are handled in the same fashion.

Evicting Operations. Figure 8 presents our algorithm for the second phase of instruction execution when the instructions exit the core's store buffer and take effect on the memory system. The Evict_SB procedure for stores assigns a store a clock when it is evicted from the store buffer. It then updates the `storemap` for the

```
function Exec_Store($addr$, $val$, $\tau$)
    Enqueue $\langle$store, $addr$, $val\rangle$ into $S_\tau$.
function Exec_CLFLUSH($addr$)
    Enqueue $\langle$clflush, $addr\rangle$ into $S_\tau$.
function Exec_CLWB($addr$)
    Enqueue $\langle$clwb, $addr\rangle$ into $S_\tau$.
function Exec_SFENCE
    Enqueue $\langle$sfence$\rangle$ into $S_\tau$.
function Exec_MFENCE
    Evict all entries in $S_\tau$.
    Flush $F_\tau$.
```

Figure 7: Algorithm for executing instructions.

```
function Evict_SB($\langle$store, $addr$, $val\rangle$, $\tau$)
    $\sigma_{curr} := \sigma_{curr} + 1$
    top($exec$).storemap := top($exec$).storemap[$addr \to \langle\tau, \sigma_{curr}\rangle$].
function Evict_SB($\langle$clflush, $addr$, $CV_{clflush}\rangle$, $\tau$)
    $\sigma_{curr} := \sigma_{curr} + 1$
    for all $addr_s$ such that CacheID($addr_s$) = CacheID($addr$) do
        $\langle\tau_s, \sigma_s\rangle$ = top($exec$).storemap($addr_s$)
        if $\sigma_s < CV_{clflush}(\tau_s) \wedge$
           $\nexists\langle\tau', \sigma_{\tau'}\rangle \in$ top($exec$).flushmap($\sigma_s$), $\sigma_{\tau'} < CV_{clflush}(\tau')$ then
            top($exec$).flushmap($\sigma_s$) := top($exec$).flushmap($\sigma_s$) $\cup$ $\{\langle\sigma_\tau, CV_{clflush}(\tau_{fence})\rangle\}$
function Evict_SB($\langle$clwb, $addr$, CV$\rangle$, $\tau$)
    Add $\langle addr, CV, \tau\rangle$ to $F_\tau$.
function Evict_SB($\langle$sfence$\rangle$, $\tau$)
    $\sigma_{curr} := \sigma_{curr} + 1$
    Flush $F_\tau$.
function Evict_FB($\langle addr, CV_{flush}\rangle$, $\tau_{flush}$, $\langle CV_{fence}, \tau_{fence}\rangle$)
    for all $addr_s$ such that CacheID($addr_s$) = CacheID($addr$) do
        $\langle\tau_s, \sigma_s\rangle$ = top($exec$).storemap($addr$)
        if $\sigma_s < CV_{flush}(\tau_s) \wedge$
           $\nexists\langle\tau, \sigma_\tau\rangle \in$ top($exec$).flushmap($\sigma_s$), $\sigma_\tau < CV_{fence}(\tau)$ then
            top($exec$).flushmap($\sigma_s$) := top($exec$).flushmap($\sigma_s$) $\cup$ $\{\langle\tau_{fence}, CV_{fence}(\tau_{fence})\rangle\}$
```

Figure 8: Algorithm for evicting store and flush buffers.

execution to record that the address *addr* was written to by the current thread τ and store. The Evict_SB procedure for `clflush` instructions assigns the `clflush` instruction a clock.

Next it updates the `storemap` for each most recent store to an address on the same cache line to include the thread and clock vector for this `clflush` instruction if (1) the store happens before the `clflush` instruction and (2) there is not already a flush instruction in `storemap` that happens before the `clflush`. The Evict_SB procedure for `clwb` instructions adds the `clwb` instruction to the thread's flushbuffer F_τ.

The Evict_SB procedure for `sfence` instruction evicts the `clwb` instructions in the thread's flushbuffer F_τ. The Evict_FB procedure handles evicting a `clwb` instruction from the flushbuffer. It takes as parameters (1) the address, clock vector, and thread identifier for the `clwb` instruction and (2) the clock vector and thread identifier for the fence instruction. This procedure updates the `storemap` for each most recent store to an address on the same cache line to include the thread and clock vector for this `clflush` instruction if (1) the store happens before the `clwb` instruction and (2) there is no flush instruction in `storemap` that happens before the fence.

Processing Loads. Figure 9 presents our algorithm for handling loads that read from stores performed by a prior execution e. The Load_Atomic function handles atomic loads. It updates the lower

```
function Load_Atomic($addr$, $\langle e, CV_s, val\rangle$)
    $e$.lastflush := $e$.lastflush[CacheID($addr$) $\to$ $e$.lastflush(CacheID($addr$)) $\cup$ $CV_s$]
    $e.CV_{pre} := e.CV_{pre} \cup CV_s$
function Load_NonAtomic($addr$, $\langle e, CV_s, \tau, val\rangle$)
    if $CV_s(\tau) > e$.lastflush(CacheID($addr$))($\tau$) $\wedge$
           $\nexists\langle\tau_f, \sigma_f\rangle \in e$.flushmap($s$), $\sigma_f < e.CV_{pre}(\tau_f)$ then
        Print persistency race error
    $e.CV_{pre} := e.CV_{pre} \cup CV_s$
```

Figure 9: Algorithm for loads.

bound for the last time that the respective cache line was flushed in execution e. It then updates the clock vector CV_{pre} that is used to compute the smallest consistent prefix of the execution e. The Load_NonAtomic function handles non-atomic loads. It checks whether the store that the load reads from is ordered after the lower bound for the last time that its cache line was flush. If so, then it checks whether the cache line was flushed after the store. If not, it prints an error. It then updates the clock vector CV_{pre} that is used to compute the smallest consistent prefix of the execution e.

Implementation. Yashme uses the Jaaru open-source model checking infrastructure [18] to simulate program executions. This infrastructure uses an LLVM compiler frontend to automatically instrument programs to intercept reads, writes, `clflush` instructions, `clwb` instructions, and memory fences. This infrastructure implements a simulation framework for persistent memory and this framework supports injecting crashes between executions. The simulation framework enables Yashme to reason about all potential effects of cache flushes and precisely control execution while at the same time avoids requiring access to actual hardware supporting these instructions. Yashme is implemented as a plugin for the model checking infrastructure, which reports persistent memory relevant execution events to Yashme. In model checking mode, Yashme systematically injects crashes before every `clflush` or `fence` operation. While Yashme controls multithreaded scheduling to regenerate the same execution, it does not exhaustively explore the space of schedules.

We implement a new mode, *random mode*, on the infrastructure that randomly generates executions in addition to the existing model checking mode. This enables Yashme to execute programs that cannot easily be model checked. At each load, the infrastructure computes a set of candidate stores from pre-crash executions that the load could read from depending on when a cache line was made persistent. Yashme leverages this design to check all of these candidate stores for potential data races using the Load_NonAtomic procedure. In random mode, Yashme randomly explores different concurrent schedules and read choices and simulates crashes before random fence operations. Users can specify the number of random executions based on the complexity and size of the tool under test.

7 EVALUATION

We ran our experiments on an Ubuntu Linux 18.04 machine with a 4 core Intel Xeon E3-1245 v3 CPU and 32GB RAM. We used gcc version 7.5.0 and clang version 11.0.0.

Our Benchmarks. We have evaluated Yashme on state-of-the-art persistent memory applications and libraries. Yashme was tested with the RECIPE benchmarks [32], FAST_FAIR [22], CCEH [45], PMDK [24], Memcached [9], and Redis [30]. These frameworks were used in prior works to evaluate their bug-finding tools [18, 37, 38, 46]. Prior testing tools revealed multiple bugs but none of them were capable of detecting persistency races. Yashme is the *first persistency race detector* and has found 24 persistency races in well-tested persistent memory applications.

7.1 Methodology

We first evaluated Yashme on a collection of data structures [22, 32, 45]. RECIPE [32] is a collection of concurrent DRAM indexes for persistent memory. We used all RECIPE benchmarks (*i.e.*, P-ART, P-BwTree, P-CLHT, and P-Masstree) except P-HOT because it did not compile with LLVM. CCEH [45] is an efficient hash table for persistent memory. FAST_FAIR [22] is a fault-tolerant B+-tree for persistent memory. We changed the compiler options for these benchmarks to disable optimizations to avoid any optimizations that might reorder memory operations and potentially cause us to miss reporting persistency races. We recompiled these programs with Yashme and used their example test application to drive our testing. These example programs manipulate each data structure through standard insertion, deletion, and lookup operations.

We also evaluated Yashme on real-world frameworks [9, 24, 30]. PMDK [24] is a collection of libraries and tools developed by Intel for application developers to simplify accessing persistent memory devices. This is the most active open-source PM framework, which has been maintained for 7 years, and is used both by academia and industry. Similar to prior works [18, 37, 38, 46], we used example data structures provided with PMDK to find bugs in the PMDK library (*i.e.*, BTree, CTree, RBTree, Hashmap atomic and Hashmap TX). Redis [30] is a popular in-memory database and memory cache ported by Intel to use both DRAM and persistent memory. It uses PMDK's transaction APIs to store data on persistent memory. We developed our own client to modify the database server using insertion and lookup operations. Memcached [9] is a high-performance distributed memory caching system ported to use persistent memory. This in-memory key-value store uses low-level *libpmem* APIs to flush cache lines to persistent memory. We developed our own client from Memcached's test cases to evaluate Yashme. This client modifies the cache server using insertion and lookup operations. We adapted the original compilation flags for these frameworks and *only* changed the flags to cause them to link against Yashme dynamic library.

We evaluated Yashme with PM indexes with the model checking mode, and we used the random execution mode for Memcached, Redis, and PMDK which are relatively more complicated. The working mode can be specified by a command-line argument.

7.2 Race Detection

We run Yashme over RECIPE, CCEH, FAST_FAIR, PMDK, Memcached, and Redis to automatically detect persistency races. We manually deduplicated all race reports since one variable can participate in multiple buggy scenarios. Then, we manually inspected the race reports. Yashme has found a total of 24 races in these programs that are *all new and have not been discovered by prior tools.* We first discuss our experience with the collection of PM data structures [22, 32, 45]. Table 3 reports 19 races we found in these data structures. For each bug, we list the program in which the bugs were found and the field that causes the persistency race.

```
Value_t CCEH::Get(Key_t& key) {
  ...
  if (dir_->_[slot].key == key) {
    ...
    return dir_->_[slot].value;
  }
  ...
}
```

Figure 10: The `CCEH::Get` method from the `CCEH` hashtable reads from the non-atomic `key` and `value` fields.

Table 3: Races found in CCEH, FAST_FAIR, and RECIPE benchmarks.

#	Benchmark	Root Cause of Bug
1	CCEH	*value* in *Pair* struct in pair.h
2	CCEH	*key* in *Pair* struct in pair.h
3	FAST_FAIR	*last_index* in *header* class in btree.h
4	FAST_FAIR	*switch_counter* in *header* class in btree.h
5	FAST_FAIR	*key* in *entry* class in btree.h
6	FAST_FAIR	*ptr* in *entry* class in btree.h
7	FAST_FAIR	*root* in *btree* class in btree.h
8	FAST_FAIR	*sibling_ptr* in *header* class in btree.h
9	P-ART	*compactCount* in *N* class in N.h
10	P-ART	*count* in *N* class in N.h
11	P-ART	*deletitionListCount* in *DeletionList* class in Epoche.h
12	P-ART	*headDeletionList* in *DeletionList* class in Epoche.h
13	P-ART	*nodesCount* in *LabelDelete* struct in Epoche.h
14	P-ART	*added* in *DeletionList* class in Epoche.h
15	P-ART	*thresholdCounter* in *DeletionList* class in Epoche.h
16	P-BwTree	*epoch* in *BwTreeBase* class in bwtree.h
17	P-Masstree	*root_* in *masstree* class in masstree.h
18	P-Masstree	*permutation* in *leafnode* class in masstree.h
19	P-Masstree	*next* in *leafnode* class in masstree.h

Due to space constraints, we only elaborate on the persistency races for bug #1 and #2 of Table 3. Figure 3 shows the source code of pre-crash execution where the program writes to the `key` and `value` fields of a `Segment` and flushes them. Figure 10 shows the post crash execution where the program reads from `key` and `value` fields. These variables are non-atomic and thus a poorly timed crash could cause the program to read from from partially persisted stores in the `CCEH:Get` method in the post-crash execution.

Note that the majority of these persistency race bugs are in the core implementation of the data structures, *e.g.*, the key and value fields of the tree. Some of the persistency races were found in memory allocators. These races can lead to different symptoms in the program including (1) accessing an illegal memory address and crashing with a segmentation fault, (2) exiting with assertion failure, and (3) showing wrong or undefined behavior.

Next, we discuss the results for the PMDK benchmarks, Redis, and Memcached [9, 24, 30]. Redis uses PMDK's libpmemobj and transaction APIs to modify persistent memory and Memcached uses PMDK's libpmem API. Table 4 reports 5 new persistency races found by Yashme in PMDK, Redis, and Memcached. For each bug, we list the variable that causes the persistency race. The majority of races revealed by Memcached and PMDK testcases involve header fields for the object pool. PMDK library uses these fields for memory management, *e.g.*, defragmentation and garbage collection. Most of these races could be revealed by Redis as well. Yashme found 4 persistency races in Memcached. Similar to PMDK, the majority of these races are related to the internal representation of the object pool, *e.g.*, flags for validating the data. Persistency races in Table 4 can corrupt a persisted store leading to data loss. All these races are new and none have been reported before.

Table 4: Races found in PMDK, Redis, and Memcached.

#	Benchmark	Root Cause of Bug
1	PMDK	pointer to *ulog_entry* in ulog.c
2	memcached	*valid* variable in *pslab_pool_t* struct in pslab.c
3	memcached	*id* variable in *pslab_t* struct in pslab.c
4	memcached	*it_flags* variable in *item_chunk* struct in memcached.h
5	memcached	*cas* variable in *item* struct in memcached.h

Table 3 and Table 4 report the variables that cause persistency race in each benchmark. As mentioned in Section 3, store tearing is not the only way the compiler can introduce problems. The compiler can invent stores to locations that are guaranteed to be written to [26]. Thus a persistency race on byte-size fields such as #14 in Table 3 and #2 - #4 in Table 4 are not safe. This bug report containing information for each problematic store is very beneficial to the developers to reason about different buggy scenarios. To fix these bugs, the developers need to replace racing non-atomic stores with atomic ones (in C++ change int to `atomic<int>` or in C change int to `atomic_int`. On x86 this incurs no overhead if one uses atomic stores with the `memory_order_release` memory ordering, because they are implemented with normal move instructions. But it ensures that compiler optimizations will not tear the store.

7.3 Optimization & Performance

Persistency races have a narrow window for detection and this can make detecting persistency races rather challenging. To understand the importance of searching for persistency races in prefixes of available executions, we injected crashes before every fence in the execution of Fast_Fair, CCEH, and the RECIPE benchmarks. We ran Yashme with this optimization (prefix) and without this optimization (baseline) to compare their bug finding capabilities. Table 5 reports persistency races detected by these two techniques. For each technique, we report numbers for running a single randomly generated execution. Yashme finds **5**× more persistency races. This demonstrates that the prefix-based approach can find many more persistency races because prefixes generalize executions—many of the derived executions would otherwise be hard to reach. Table 5 also reports the times taken to run one random execution on both Yashme and Jaaru, the underlying infrastructure. They have comparable running times because the race checks introduce minimal overheads.

Note that *persistency races were not known before and hence there does not exist any other tool with which we can compare Yashme directly.*

Table 5: # races detected w/ and w/o prefix-based expansion for a single execution on RECIPE, PMDK, Memcached, and Redis benchmarks, as well as execution times for both Yashme and Jaaru (the underlying checking infrastructure). Yashme incurs minimal overhead compared to Jaaru.

Benchmark	Prefix	Baseline	Yashme Time	Jaaru Time
CCEH	2	0	0.043s	0.041s
Fast_Fair	2	1	0.039s	0.039s
P-ART	0	0	0.046s	0.044s
P-BwTree	0	0	0.034s	0.033s
P-CLHT	0	0	0.159s	0.157s
P-Masstree	2	0	0.037s	0.038s
Btree	1	0	2.541s	2.095s
Ctree	1	0	2.544s	2.099s
RBtree	1	0	2.552s	2.100s
hashmap-atomic	1	0	2.298s	1.896s
hashmap-tx	1	0	2.294s	1.892s
Redis	0	0	5.623s	5.361s
Memcached	4	2	8.032s	8.035s

7.4 Bug Reporting and Confirmation

We contacted the authors of these programs to obtain their feedback on the bugs found by Yashme. As of the time of writing the paper, we have heard back from the authors of PMDK, Memcached-pmem, and the RECIPE benchmarks. For RECIPE, the authors confirmed that the bugs 9-11 and 19-21 are real bugs. For the persistency races 12-18, they are related to the code of a memory allocator that is known to be crash inconsistent—the RECIPE benchmark suite did not attempt to correctly implement a crash-consistent memory allocator. To be clear, these are all real persistency races, but the code for the memory allocator needs to be replaced anyways and hence they would not fix the bugs 12-18. Furthermore, we reported the bugs 2-5 in Memcached-pmem to its developers. They confirmed that all of these bugs are real. After our bug report, both developers of RECIPE and Memcached-pmem immediately fixed the reported persistency races. These fixes are publicly available on their github repositories. For PMDK, the developers confirmed bug 1.

7.5 Discussion

Analyzing Bugs. The current version of the code was compiled with gcc v7.5 and clang v11.0 for x86. We manually investigated the the assembly generated by this compiler and there were many cases of using memory operations (*i.e.*, `memset` and `memcpy`) in object initializations which can lead to store tearing and persistency bugs (*e.g.*, bug #8 in Fast_Fair). For other cases, while these particular compilers may not tear the racing stores, the fact that it can mean that compiler upgrades, architectural changes, or even unrelated changes to the code (*e.g.*, adding new fields to struct/class and breaking variable's alignments) can cause optimizations to generate problematic assembly code. Leaving a persistency race in mission-critical code (*e.g.*, storage systems, self-driving cars, airplane control systems, *etc.*) can lead to catastrophic failures and even disasters from library/compiler/hardware upgrades. *Although these bugs may or may not manifest in today's code, it can suddenly break tomorrow's executions even if user code is not changed at all.* As confirmed by PMDK developers, "making sure their code does not depend on compiler/library behaviors" is their daily routine and "they do extensive validation to that end".

Although the developer can manually inspect the assembly code on every compiler, architectural, and code change, this approach is very time-consuming and not practically possible for large source code bases. Yashme automatically flags out such bugs caused by miscompiling that can corrupt a large data store. We believe it is strictly necessary to fix such bugs in mission-critical code that cannot afford to break.

Benign Issues. Programs can access inconsistent data in the post-crash execution. However, these program accesses are not necessarily followed by sensitive operations that use this data. For examples, programs can use customized fault tolerance techniques to detect data inconsistency and ignore the inconsistent data. For example, PMDK, Redis, and Memcached use a checksum-based strategy to verify that data is consistent. This strategy computes a checksum on the data and writes the checksum. Before using the data, the program first verifies the checksum to validate data integrity. Even if these programs read from partially-persistent data, such data are not used as they fail the checksum validation. In addition to the 24 persistency races, Yashme found 10 bugs that are benign issues due to checksums (although these are still true persistency races by definition). **Yashme did not report any other types of benign issues other than the ones from checksums.** A future implementation of Yashme could use annotations to suppress race warnings from stores that are read by the checksum validation procedure.

As with any dynamic tool, Yashme can only find bugs in the executions it explores. As such, Yashme may miss bugs in unexplored executions.

Persistency Races on eADR CPUs. Persistency races are still possible on eADR systems [23] where flushing is not required. The absence of races on a non-eADR system implies the absence of races on eADR systems, but the opposite is not true. Thus, Yashme can be used as is to guarantee the absence of races on the eADR systems. However, Yashme could be adapted to only detect races that are possible on eADR systems by adding support to handle the slightly different persistency semantics.

8 RELATED WORK

Bug/crash consistency detection. There exists a large body of work on testing [29, 31, 44, 56], checking [43, 51, 58, 59], and formally verifying [5, 6, 54] file system implementations to find and eliminate crash consistency bugs. Fuzzing techniques such as Janus [56] and Hydra [29] mutate disk images and file operations to explore states of file system code. Using heuristics, B3 [44] employs a bounded testing technique to explore states in a bounded space. EXPLODE [58], FiSC [59], and SAMC [33] use model checkers to systematically explore states of a file system implementation. There is much work on torn writes in disk where only part of a multi-sector write completes. File systems use a variety of techniques to ensure consistency, such as shadow updates [2, 50], journaling [53], soft updates [41], and post-reboot checking [42, 48]. Although crash consistency bugs in file systems bear similarities with bugs in PM programs, they are fundamentally different in the access granularity as well as how writes are performed.

There is a recent line of work on checking/testing PM programs to find bugs. PMTest [38] lets developers annotate a program with checking rules to infer the persistency status of writes and ordering constraints between writes. Pmemcheck [28] checks how many stores were not made persistent and detects memory overwrites using binary rewriting. PMDebugger [11] is a debugger developed on top of Valgrind that tracks operations to find persistency bugs. PMDebugger relies on programmers to explicitly annotate ordering constraints. Agamotto [46] finds bugs in persistent memory programs by using symbolic execution. It tracks the state of persistent memory objects and their corresponding cache lines in the program, *i.e.*, whether the cache line is modified. Persistency races result from the interaction between the pre-crash and post-crash executions. Agamotto only explores the pre-crash execution and thus cannot be easily extended to find persistency races. Jaaru [18] takes a constraint-based approach to enumerating executions that can drastically reduce the number of post-failure executions. Yashme is built on Jaaru's constraint-based execution engine that enables Yashme to observe many possible stores that a load can read from (depending on when the cache line was evicted) and checks for persistency races in all of them. Although all existing tools are able to find many bugs, they all effectively treat writes to persistent memory as atomic or cannot distinguish if a source-level store operation is torn at the binary level. Thus, the current implementations of all previous tools are unable to detect persistency races.

Programming Models for PM. There is a great deal of work on building programming systems that allow developers to use PM in a reliable way without knowing the details of PM. For example, a line of work [3, 8, 15, 16, 34, 57] proposes to use (software or hardware) transactions to provide (failure and thread) atomicity. Another line of work [1, 4, 20, 25, 35] advocates use of locks or synchronization-free regions [17]. Yashme is complementary to these approaches, it can be used to check the correctness of their implementations.

9 CONCLUSION

This paper formally defines a new class of bug in PM programs, persistency races, and present Yashme, the first tool to detect persistency races. Yashme builds on a novel technique that improves persistency race detection by checking prefixes of the pre-crash execution. Our evaluation shows that Yashme has found 24 real persistency races in our applications.

ACKNOWLEDGMENTS

We would like to thank our shepherd, Baptiste Lepers, and the anonymous reviewers for their thorough and insightful comments that helped us substantially improve the paper. This work is supported by the National Science Foundation grants CNS-1703598, OAC-1740210, CNS-1763172, CNS-1907352, CNS-2006437, CCF-2006948, CNS-2007737, CCF-2102940, CNS-2128653, and CNS-2106838, as well as ONR grants N00014-16-1-2913 and N00014-18-1-2037.

A ARTIFACT APPENDIX

A.1 Abstract

This artifact contains a vagrant repository that downloads and compiles the source code for Yashme, its companion compiler pass, and benchmarks. The artifact enables users to reproduce the bugs that are found by Yashme in PMDK, Memcached, and Redis (Table 4 of the paper), and RECIPE (Table 3) as well as the performance results to compare Yashme with Jaaru (Table 5).

A.2 Artifact Check-List (Meta-Information)

- **Algorithm:** Persistency race detector
- **Program:** Yashme
- **Compilation:** GCC 7.5.0 and Clang
- **Binary:** Instrumentation LLVM pass
- **Data set:** RECIPE, Redis, Memcached, and PMDK benchmarks
- **Run-time environment:** Any system that can run Vagrant
- **Hardware:** One 6 core 3.7 GHz Intel i7 machine with 32 GB DDR4 memory
- **Run-time state:** Managed by Jaaru's x86 simulator
- **Execution:** Automated by Jaaru's tooling system
- **Metrics:** Reporting persistency race in program under test
- **Output:** Persistency race bugs. Logging performance measurement for executions.
- **Experiments:** Regenerating all bugs found by Yashme. Reproducing performance results and comparing them with Jaaru (underlying open-source model checker)
- **How much disk space required (approximately)?:** 80G
- **How much time is needed to prepare workflow (approximately)?:** 4 hours
- **How much time is needed to complete experiments (approximately)?:** About 90 minutes
- **Publicly available?:** Yes. Open-source on GitHub
- **Code licenses (if publicly available)?:** GNU GENERAL PUBLIC LICENSE Version 2
- **Data licenses (if publicly available)?:** Lenovo, BSD-3-Clause, and Apache License 2.0.
- **Workflow framework used?:** Vagrant.
- **Archived (provide DOI)?:** https://dl.acm.org/doi/10.1145/3462316

A.3 Description

Our workflow has four primary parts: (1) creating a virtual machine and installing dependencies needed to reproduce our results, (2) downloading the source code of Yashme and the benchmarks and building them, (3) providing the parameters corresponding to each bug to reproduce the bugs, and (4) running the benchmarks to compare Yashme with the Jaaru, the underlying model-checker. After the experiment, the corresponding output files are generated for each bug and each performance measurement.

A.3.1 How to Access. All source code is open-source and available on GitHub. Our packaging requires cloning the vagrant system repository from https://github.com/uci-plrg/pmrace-vagrant. As described in the *README.md* file of the repository, you will need to install a VirtualBox VM and Vagrant on your machine. Then, the vagrant setup will install the required dependencies and download the source code of the tools from our git repository. Next, it builds each tool on the virtual machine.

A.3.2 Hardware Dependencies. Our tooling system and Yashme have no special hardware dependencies and it can be running on any x86 machine with at least 32GB RAM and 4 cores.

A.3.3 Software Dependencies. To run our system, the following should be installed on the local machine:

- Linux (we tested on Ubuntu)
- Vagrant
- VirtualBox
- Vagrant-disksize plugin

A.3.4 Data Sets. To evaluate Yashme, our tooling system downloads the source code of RECIPE, Redis, Memcached, and PMDK from our git repository. We forked a branch from the original source code of these benchmarks that don't contain our bug fixes. The tooling system automatically sets up and builds these benchmarks and runs them under Yashme to identify bugs in them.

A.4 Installation

Please see the *README.md* file of the https://github.com/uci-plrg/pmrace-vagrant repository, which contains a detailed step-by-step guide to setup Yashme on a virtual machine. Then, our scripts automatically do the following:

(1) Install all the dependencies needed to install and evaluate Yashme on different benchmarks.
(2) Check out the source code for LLVM, Yashme, Yashme's LLVM pass, RECIPE, Redis, Memcached, and PMDK.
(3) Include Yashme's LLVM pass to LLVM and building it
(4) Set up and build Yashme with two different configurations (One for RECIPE that uses *libvmemmalloc*, and one for PMDK that uses *libpmem* APIs).
(5) Set up and building RECIPE (including CCEH, FAST_FAIR, P-ART, P-BwTree, P-CLHT, and P-Masstree benchmarks), Redis, Memcached, and PMDK benchmarks.
(6) Generate seven scripts in the *home (or ~/)* directory of the virtual machine to generate the results.

Once the scripts are finished setting up the virtual machine and benchmarks, the user can use Yashme on the virtual machine to further evaluate different benchmarks or regenerate our evaluation results.

A.5 Experiment Workflow

After setting up the virtual machine, the user can use 'vagrant ssh' to connect to the VM and use Yashme. The detailed instructions to run the suggested workflow is included in the *README.md* file of https://github.com/uci-plrg/pmrace-vagrant repository. There are seven scripts in the *home* directory of the virtual machine that user can run:

perf.sh : It runs PMDK, Redis, Memcached, and RECIPE benchmarks using Yashme and gathers measurements to compare Yashme against Jaaru. For each benchmark, the corresponding log file is generated in *~/results/performance*.

recipe-bugs.sh : It runs the RECIPE benchmarks using Yashme and sets the corresponding parameters to reproduce each bug. For each benchmark, the corresponding log file is generated in *~/results/recipe*.

pmdk-bugs.sh : It runs PMDK benchmarks by using Yashme and set the corresponding parameters to reproduce each bug. For each test case, the corresponding log file is generated in *~/results/pmdk*.
memcached-client.sh : It runs memcached client test script to execute scenarios to reproduce each bug in Memcached benchmark.
redis-client.sh : It runs Redis client test script to execute scenarios to reproduce each bug in Redis benchmark.
memcached-server.sh : It runs Memcached benchmark by using Yashme and set the corresponding parameters to reproduce each bug. The server script runs on one terminal, and the client script runs in another terminal. The persistency races are printed out on the server's terminal.
redis-server.sh : It runs Redis benchmark by using Yashme and set the corresponding parameters to reproduce each bug. The server script runs on one terminal, and the client script runs on another terminal. The persistency races are printed out on the server's terminal.

A.6 Evaluation and Expected Result

After successfully running the experiment using our scripts, the *results* directory is generated in the *home* directory. This directory contains the following results:

A.6.1 RECIPE. There are 7 files in *~/results/recipe* directory with the pattern of *[BENCHMARK_NAME]-races.log*. Each file contains the persistency races found by Yashme in each benchmark. Figure 11 contains information about how Yashme reports each bug correspond to Table 3.

#	Bug ID	Cause of Bug
1	CCEH-1	Write to *value* in CCEH_LSB.cpp:29
2	CCEH-2	Write to *key* in CCEH_LSB.cpp:31
3	FAST_FAIR-1	Write to *last_index* in btree.h:643 and btree.h:831
4	FAST_FAIR-2	Write to *switch_counter* in btree.h:578 and btree.h:820
5	FAST_FAIR-3	Write to *key* in btree.h:584, btree.h:624, and btree.h:605
6	FAST_FAIR-4	Write to *ptr* in btree.h:585, btree.h:604, btree.h:625, and btree.h:828
7	FAST_FAIR-5	Write to *root* in btree.h:1857
8	FAST_FAIR-6	Write to *sibling_ptr* in btree.h:824
9	P-ART-1	Write to *compactCount* in N4.cpp:26 and N16.cpp:15
10	P-ART-2	Write to *count* in N4.cpp:27 and N16.cpp:16
11	P-ART-3	Write to *deletitionListCount* in Epoche.cpp:44
12	P-ART-4	Write to *headDeletionList* in Epoche.cpp:57
13	P-ART-5	Write to *nodesCount* in Epoche.cpp:60:22
14	P-ART-6	Write to *added* in Epoche.cpp:63
15	P-ART-7	Write to *thresholdCounter* in Epoche.cpp:78
16	P-BwTree-1	Write to *epoch* in bwtree.h:572
17	P-Masstree-1	Write to *root_* in masstree.h:1216
18	P-Masstree-2	Write to *permutation* in masstree.h:1318, masstree.h:1364, and masstree.h:1399
19	P-Masstree-3	Write to *next* in masstree.h:1162

Figure 11: More information about the bugs found by Yashme in CCEH, Fast_Fair, and RECIPE benchmarks. Number after ':' represents the line number.

A.6.2 PMDK, Redis, Memcached. There are 2 files in *~/results/pmdk* directory. These two files contain the bug corresponding to bug #1 in Figure 12. Bug #2 - #4 are printed in the terminal output after running *memcached-server.sh* and *memcached-client.sh* scripts. Figure 12 corresponds to bugs in Table 4.

#	Benchmark Found	Symptom
1	PMDK	Write to *ulog_entry* in ulog.c:561
2	Memcached	Write to *valid* in pslab.c:368
3	Memcached	Write to *id* in pslab.c:92
4	Memcached	Write to *it_flags* in slabs.c:543, items.c:519, and items.c:343
5	Memcached	Write to *cas* in memcached.c:4290 and items.c:538

Figure 12: More information about the bugs that are found by Yashme in PMDK, Memcached and Redis benchmarks. Number after ':' represents the line number.

A.6.3 Performance Results. Running *./perf.sh* generates *performance.out* file in */results/performance* directory. This file contains the performance information as well as number of bugs found w/ or w/o prefix-based expansion algorithm corresponding to Table 5 of the paper. Note, since we manually deduplicate these bugs, the numbers of bugs in *performance.out* could be more than the number of bugs reported in paper.

A.7 Experiment Customization

The experiment workflow can be customized to install and run everything on the local machine instead of the virtual machine. To set up everything locally, download *data/setup.sh* script from the https://github.com/uci-plrg/pmrace-vagrant repository in the *home* directory of your local machine and run the script after installing the dependencies.

A.8 Notes

Note that the performance results generated for the benchmarks can be different from the numbers that are reported in the paper since there is non-determinism in scheduling threads; when stores, flushes, and fences leave the store buffer; and memory alignment in the *malloc* procedure. This non-determinism can possibly impact on the number of bugs reported in Table 3 and Table 4 for RECIPE, Redis, Memcached, and PMDK benchmarks.

REFERENCES

[1] Hans-J. Boehm and Dhruva R. Chakrabarti. 2016. Persistence Programming Models for Non-Volatile Memory. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management* (Santa Barbara, CA, USA) *(ISMM 2016)*. Association for Computing Machinery, New York, NY, USA, 55–67. https://doi.org/10.1145/2926697.2926704

[2] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. 2003. *The Zettabyte File System.* Technical Report. Sun Microsystems Solaris.

[3] Daniel Castro, Paolo Romano, and João Barreto. 2018. Hardware Transactional Memory Meets Memory Persistency. In *Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium.* Institute of Electrical and Electronics Engineers, Vancouver, BC, Canada, 368–377. https://doi.org/10.1109/IPDPS.2018.00046

[4] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-Volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 433–452. https://doi.org/10.1145/2660193.2660224

[5] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2017. Verifying a High-Performance Crash-Safe File System Using a Tree Specification. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 270–286. https://doi.org/10.1145/3132747.3132776

[6] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 18–37. https://doi.acm.org/10.1145/2815400.2815402

[7] Nikita Chopra, Rekha Pai, and Deepak D'Souza. 2019. Data Races and Static Analysis for Interrupt-Driven Kernels. In *Programming Languages and Systems*, Luís Caires (Ed.). Springer International Publishing, Cham, 697–723. https://link.springer.com/chapter/10.1007/978-3-030-17184-1_25

[8] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *ASPLOS*. Association for Computing Machinery, New York, NY, USA, 105–118. https://doi.acm.org/10.1145/1961295.1950380

[9] Inc. Danga Interactive. 2018. Memcached. https://github.com/lenovo/memcached-pmem.

[10] Will Deacon. 2019. Re: [PATCH 1/1] Fix: trace sched switch start/stop racy updates. https://lore.kernel.org/lkml/20190821103200.kpufwtviqhpbuv2n@willie-the-truck/.

[11] Bang Di, Jiawen Liu, Hao Chen, and Dong Li. 2021. Fast, Flexible, and Comprehensive Bug Detection for Persistent Memory Programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 503–516. https://doi.acm.org/10.1145/3445814.3446744

[12] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2007. Goldilocks: A Race and Transaction-Aware Java Runtime. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, USA, 245–255. http://doi.acm.org/10.1145/1250734.1250762

[13] Dawson Engler and Ken Ashcraft. 2003. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. Association for Computing Machinery, New York, NY, USA, 237–252. http://doi.acm.org/10.1145/945445.945468

[14] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, USA, 121–133. http://doi.acm.org/10.1145/1542476.1542490

[15] Kaan Genç, Michael D. Bond, and Guoqing Harry Xu. 2020. Crafty: Efficient, HTM-Compatible Persistent Transactions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 59–74. https://doi.org/10.1145/3385412.3385991

[16] Ellis Giles, Kshitij Doshi, and Peter Varman. 2017. Continuous Checkpointing of HTM Transactions in NVM. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management* (Barcelona, Spain) *(ISMM 2017)*. Association for Computing Machinery, New York, NY, USA, 70–81. https://doi.org/10.1145/3092255.3092270

[17] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. 2018. Persistency for Synchronization-Free Regions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 46–61. https://doi.org/10.1145/3192366.3192367

[18] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. 2021. Jaaru: Efficiently Model Checking Persistent Memory Programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, New York, NY, USA, 415–428. https://doi.acm.org/10.1145/3445814.3446735

[19] Piotr Balcer Hamed Gorjiara, Brian Demsky. 2021. Email exchanges with PMDK developers.

[20] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. 2017. NVthreads: Practical Persistence for Multi-Threaded Applications. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 468–482. https://doi.org/10.1145/3064176.3064204

[21] Jeff Huang, Patrick Meredith, and Grigore Rosu. 2014. Maximal Sound Predictive Race Detection with Control Flow Abstraction. In *Proceedings of the 35th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'14)*. Association for Computing Machinery, New York, NY, USA, 337–348. https://doi.org/10.1145/2594291.2594315

[22] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies* (Oakland, CA, USA) *(FAST '18)*. USENIX Association, USA, 187–200. https://doi.acm.org/10.5555/3189759.3189777

[23] Intel. 2020. Third Generation Intel Xeon Processor Scalable Family Technical Overview. https://software.intel.com/content/www/us/en/develop/articles/intel-xeon-processor-scalable-family-overview.html?wapkw=clwb.

[24] Intel Corporation. 2020. Persistent Memory Development Kit. https://pmem.io/pmdk/

[25] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 427–442. https://doi.org/10.1145/2872362.2872410

[26] Jade Alglave, Will Deacon, Boqun Feng, David Howells, Daniel Lustig, Luc Maranget, Paul E. McKenney, Andrea Parri, Nicholas Piggin, Alan Stern, Akira Yokosawa, and Peter Zijlstra. 2019. Who's afraid of a big bad optimizing compiler? https://lwn.net/Articles/793253/

[27] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. 2011. Automated Atomicity-Violation Fixing. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 389–400. https://doi.org/10.1145/1993498.1993544

[28] Tomasz Kapela. 2015. An introduction to pmemcheck (part 1) - basics. https://pmem.io/2015/07/17/pmemcheck-basic.html.

[29] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. 2019. Finding Semantic Bugs in File Systems with an Extensible Fuzzing Framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 147–161. https://doi.org/10.1145/3341301.3359662

[30] Redis Labs. 2020. Redis. https://github.com/pmem/redis.

[31] Philip Lantz, Subramanya Dulloor, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. 2014. Yat: A Validation Framework for Persistent Memory Software. In *Proceedings of the 2014 USENIX Annual Technical Conference*. USENIX Association, Philadelphia, PA, 433–438. https://www.usenix.org/conference/atc14/technical-sessions/presentation/lantz

[32] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. RECIPE: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 462–477. https://doi.org/10.1145/3341301.3359635

[33] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. 2014. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 399–414. https://doi.acm.org/10.5555/2685048.2685080

[34] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 329–343. https://doi.org/10.1145/3037697.3037714

[35] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L. Scott, Sam H. Noh, and Changhee Jung. 2018. iDO: Compiler-Directed Failure Atomicity for Nonvolatile Memory. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-51)*. IEEE Press, Fukuoka, Japan, 258–270. https://doi.org/10.1109/MICRO.2018.00029

[36] Sihang Liu, Suyash Mahar, Baishakhi Ray, and Samira Khan. 2021. PMFuzz: Test Case Generation for Persistent Memory Programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 487–502. https://doi.acm.org/10.1145/3445814.3446691

[37] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. 2020. Cross-Failure Bug Detection in Persistent Memory Programs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1187–1202. https://doi.org/10.1145/3373376.3378452

[38] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. 2019. PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 411–425. https://doi.org/10.1145/3297858.3304015

[39] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. 2006. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*. Association for Computing Machinery, New York, NY, USA, 37–48. https://doi.org/10.1145/1168857.1168864

[40] Brandon Lucia, Luis Ceze, Karin Strauss, Shaz Qadeer, and Hans Boehm. 2010. Conflict Exceptions: Simplifying Concurrent Language Semantics with Precise Hardware Exceptions for Data-Races. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*. Association for Computing Machinery, New York, NY, USA, 210–221. http://doi.acm.org/10.1145/1815961.1815987

[41] Marshall Kirk McKusick and Gregory R. Ganger. 1999. Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem. In *1999 USENIX Annual Technical Conference (USENIX ATC 99)*. USENIX Association, Monterey, CA. https://www.usenix.org/conference/1999-usenix-annual-technical-conference/soft-updates-technique-eliminating-most

[42] Marshall Kirk Mckusick and T. J. Kowalski. 1994. Fsck - The UNIX File System Check Program.

[43] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. 2015. Cross-Checking Semantic Correctness: The Case of Finding File System Bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 361–377. https://doi.org/10.1145/2815400.2815422

[44] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. 2018. Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI '18)*. USENIX Association, USA, 33–50. https://doi.acm.org/10.5555/3291168.3291172

[45] Moohyeon Nam, Hokeun Cha, Young-Ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-Optimized Dynamic Hashing for Persistent Memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies* (Boston, MA, USA) *(FAST '19)*. USENIX Association, USA, 31–44. https://doi.acm.org/10.5555/3323298.3323302

[46] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. 2020. AGAMOTTO: How Persistent is Your Persistent Memory Application?. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, USA, Article 59, 18 pages. https://doi.acm.org/10.5555/3488766.3488825

[47] Soyeon Park, Shan Lu, and Yuanyuan Zhou. 2009. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (Washington, DC, USA) *(ASPLOS XIV)*. Association for Computing Machinery, New York, NY, USA, 25–36. https://doi.org/10.1145/1508244.1508249

[48] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2005. IRON File Systems. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom) *(SOSP '05)*. Association for Computing Machinery, New York, NY, USA, 206–220. https://doi.org/10.1145/1095810.1095830

[49] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. 2019. Persistency Semantics of the Intel-x86 Architecture. *Proceedings of the ACM on Programming Languages* 4, POPL, Article 11 (December 2019), 31 pages. https://doi.org/10.1145/3371079

[50] Mendel Rosenblum and John K. Ousterhout. 1992. The Design and Implementation of a Log-Structured File System. *ACM Trans. Comput. Syst.* 10, 1 (feb 1992), 26–52. https://doi.org/10.1145/146941.146943

[51] Cindy Rubio-González, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. 2009. Error Propagation Analysis for File Systems. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 270–280. https://doi.org/10.1145/1542476.1542506

[52] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems* 15 (November 1997), 391–411. Issue 4. https://doi.acm.org/10.1145/265924.265927

[53] Stephen Tweedie Sct and Stephen C. Tweedie. 1998. Journaling the Linux ext2fs Filesystem.

[54] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. 2016. Push-Button Verification of File Systems via Crash Refinement. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*. USENIX Association, USA, 1–16. https://doi.acm.org/10.5555/3026877.3026879

[55] Yu Wang, Linzhang Wang, Tingting Yu, Jianhua Zhao, and Xuandong Li. 2017. Automatic Detection and Validation of Race Conditions in Interrupt-Driven Embedded Software. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 113–124. https://doi.org/10.1145/3092703.3092724

[56] W. Xu, H. Moon, S. Kashyap, P. Tseng, and T. Kim. 2019. Fuzzing File Systems via Two-Dimensional Input Space Exploration. In *2019 IEEE Symposium on Security and Privacy (S&P)*. Institute of Electrical and Electronics Engineers, San Francisco, CA, USA, 818–834. https://doi.org/10.1109/SP.2019.00035

[57] Yi Xu, Joseph Izraelevitz, and Steven Swanson. 2021. *Clobber-NVM: Log Less, Re-Execute More*. Association for Computing Machinery, New York, NY, USA, 346–359. https://doi.org/10.1145/3445814.3446730

[58] Junfeng Yang, Can Sar, and Dawson Engler. 2006. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7 (OSDI '06)*. USENIX Association, USA, 10. https://doi.acm.org/10.5555/1298455.1298469

[59] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. 2006. Using Model Checking to Find Serious File System Errors. *ACM Transactions on Computing Systems* 24, 4 (Nov. 2006), 393–423. https://doi.org/10.1145/1189256.1189259

[60] Wei Zhang, Chong Sun, and Shan Lu. 2010. ConMem: Detecting Severe Concurrency Bugs through an Effect-Oriented Approach. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Pittsburgh, Pennsylvania, USA) *(ASPLOS XV)*. Association for Computing Machinery, New York, NY, USA, 179–192. https://doi.org/10.1145/1736020.1736041

[61] Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. 2007. HARD: Hardware-Assisted Lockset-based Race Detection. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*. Institute of Electrical and Electronics Engineers, Scottsdale, AZ, USA, 121–132. https://doi.org/10.1109/HPCA.2007.346191