AGAMOTTO: How Persistent is your Persistent Memory Application?

Ian Neal, Ben Reeves, Ben Stoler, and Andrew Quinn, *University of Michigan;* Youngjin Kwon, *KAIST;* Simon Peter, *University of Texas at Austin;* Baris Kasikci, *University of Michigan*

https://www.usenix.org/conference/osdi20/presentation/neal

This paper is included in the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation

November 4–6, 2020

978-1-939133-19-9

Open access to the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX

AGAMOTTO: How Persistent is your Persistent Memory Application?

Ian Neal
University of Michigan

Ben Reeves
University of Michigan

Ben Stoler
University of Michigan

Andrew Quinn
University of Michigan

Youngjin Kwon
KAIST

Simon Peter
University of Texas at Austin

Baris Kasikci
University of Michigan

Abstract

Persistent Memory (PM) can be used by applications to directly and quickly persist any data structure, without the overhead of a file system. However, writing PM applications that are simultaneously correct and efficient is challenging. As a result, PM applications contain correctness and performance bugs. Prior work on testing PM systems has low bug coverage as it relies primarily on extensive test cases and developer annotations.

In this paper we aim to build a system for more thoroughly testing PM applications. We inform our design using a detailed study of 63 bugs from popular PM projects. We identify two application-independent patterns of PM misuse which account for the majority of bugs in our study and can be detected automatically. The remaining application-specific bugs can be detected using compact custom oracles provided by developers.

We then present AGAMOTTO, a generic and extensible system for discovering misuse of persistent memory in PM applications. Unlike existing tools that rely on extensive test cases or annotations, AGAMOTTO symbolically executes PM systems to discover bugs. AGAMOTTO introduces a new symbolic memory model that is able to represent whether or not PM state has been made persistent. AGAMOTTO uses a state space exploration algorithm, which drives symbolic execution towards program locations that are susceptible to persistency bugs. AGAMOTTO has so far identified 84 new bugs in 5 different PM applications and frameworks while incurring no false positives.

1 Introduction

Persistent Memory (PM) is a promising new technology that offers an appealing performance-cost tradeoff for application developers. PM technologies, such as Intel Optane DC [36], can offer persistent memory accesses with latencies that are only 2–3× higher than the latencies of DRAM [70]. Moreover, such PM technologies are cheaper than DRAM per GB of capacity [3]. As byte-addressable memory, PM can also be accessed via processor load and store instructions. Application developers have already started building systems that use PM directly, without relying on heavyweight system calls to ensure durability, including ports of popular systems such as memcached [24] and Redis [21].

While using PM directly via persistent data structures can offer performance, it is challenging to write PM-based applications that are simultaneously correct and efficient [12, 18, 33, 52, 54, 60, 71, 76]. Persistent memory writes in the CPU cache must be explicitly flushed to PM using specific instructions or APIs. In certain cases, PM flush operations need to be ordered using memory fences to enforce crash consistency. Incorrect usage of these mechanisms can result in *persistency bugs* which break crash-consistency guarantees or degrade application performance. Persistency bugs are challenging to diagnose because their symptoms are easily masked. For example, crash-consistency bugs may be masked because PM writes are implicitly flushed when dirty (or updated) cache lines are evicted from the CPU—furthermore, flushes which are required for proper crash consistency under one execution path may be redundant and unnecessary under a different program execution path, leading to performance degradations.

Several systems have been built to aid with testing PM applications; however, these existing approaches are either specific to a target application or require significant manual developer effort. Intel designed Yat [44] and pmemcheck [65] specifically to test the crash consistency and durability of PMFS (Persistent Memory File System) [27] and PMDK (Persistent Memory Development Kit) [20], respectively. To find bugs, Yat exhaustively tests all possible update orderings, and pmemcheck tracks annotated updates. Both of these tools are specific to a single system (PMFS and PMDK, respectively) and are hard to generalize. Other tools like Persistency Inspector [62], PMTest [50], and XFDetector [49] are applicable to general PM systems, but require developer annotations and extensive test suites to thoroughly test PM applications.

In order to determine the extent to which persistency bug finding can be automated (i.e., not require program annota-

tions) to test general systems, we perform a study of 63 bugs in PM applications and frameworks. We identify two application-independent patterns of PM misuse (*missing flush/fence* and *extra flush/fence*) which cover the majority (89%, or 56 out of 63) of bugs in our study and can be detected automatically. The remaining bugs are application-specific; for example, many of the remaining bugs involve misusing transactions when updating data-structures. Existing PM testing approaches do not identify application-independent patterns of misuse, and therefore require annotations to detect any PM bug. In addition to classifying bugs based on their pattern of PM misuse, we also classify bugs based on whether they affect performance or correctness.

Based on the insights gained through our study, we present AGAMOTTO, a framework for detecting bugs in PM applications that does not rely on extensive test cases. Instead, AGAMOTTO uses symbolic execution [8] to thoroughly explore the state space of a program. In addition to expanding path coverage, symbolic execution also allows AGAMOTTO to detect persistency bugs in an application without access to underlying physical PM resources. AGAMOTTO introduces a memory model to track updates made to PM by the explored program paths, and supports *bug oracles* which use the PM state to identify bugs in the program. AGAMOTTO automatically detects persistency bugs using two *universal persistency bug oracles* based on the common patterns of PM misuse identified by our study. The first is an *unflushed/unfenced* oracle that identifies modifications to PM cache lines that are not flushed or fenced (both a correctness and performance issue) and the second an *extra-flushed/fenced* oracle that identifies duplicate flushes of the same cache line or unnecessary fences (a performance issue [18, 52, 60, 71, 76]).

To identify application-specific persistency bugs, AGAMOTTO allows developers to provide *custom persistency bug oracles*. To demonstrate the versatility of custom oracles, we implemented two such oracles in AGAMOTTO to detect bugs related to misuse of the PMDK transactional API [20, 49, 50].

Analyzing large PM applications using traditional symbolic execution [8] leads to scalability issues since the state space of possible executions grows exponentially with the size of the analyzed program. AGAMOTTO uses a novel search algorithm that prunes the execution states it analyzes, allowing AGAMOTTO to discover more bugs. Prior to symbolic execution, AGAMOTTO uses a whole-program static analysis to determine instructions that modify PM (stores, flushes, etc.) and assigns a unit priority to them. AGAMOTTO then assigns an aggregate priority to each instruction by back-propagating the unit priorities from each PM-modifying instruction—this makes the aggregate priority a measure of the number of PM-modifying instructions reachable from a particular instruction. AGAMOTTO uses priorities to steer symbolic execution into program states that frequently modify PM.

We used AGAMOTTO to find 84 new persistency bugs in real-world systems including PMDK (a mature PM library) [20], memcached-pm [24], Redis-pmem [21], NVM-Direct [7], and RECIPE [45]. In particular, we found 13 new correctness and 70 new performance bugs using the universal persistency bug oracles, and 1 new correctness bug using a custom persistency bug oracle. We report all bugs to their authors, and so far 40 of them have been confirmed and none denied.

In this paper we make the following contributions:

- We perform a detailed study of persistency bugs in PMDK as well as bugs found by prior work, and present a new taxonomy of persistency bugs.
- We build AGAMOTTO[1], a persistency bug detection tool that can test real-world PM programs using a novel state exploration algorithm. AGAMOTTO automatically detects bugs using two universal persistency bug oracles, without relying on user annotations or an extensive test suite. AGAMOTTO is extensible with custom bug oracles that can detect application-specific bugs.
- We use AGAMOTTO to find 84 new bugs in 5 applications and persistent memory libraries, compared to the 6 persistency bugs found in persistent applications by the state of the art (PMTest [50], which finds 3 bugs, and XFDetector [49], which finds 3 bugs). AGAMOTTO does not incur any false positives in our evaluation.

In the rest of this paper, we first provide background on PM programming and describe the challenges of PM bug finding (§2). We then present the results of our PM bug study and provide common patterns of PM misuse that identify PM bugs (§3). Then, we discuss the persistency bug detection algorithms and search techniques underlying AGAMOTTO (§4). Next, we describe the high-level design of AGAMOTTO and evaluate the system with respect to both the number of bugs found and the impact of these bugs (§6). Finally, we describe related PM bug detection work (§7).

2 Background and Challenges

We now provide a background on persistent memory (PM) programming and difficulties associated with writing correct and efficient PM programs.

2.1 Persistent Memory Programming

```
1 int *x = pm_alloc(), *y = pm_alloc();
2 *x = 1;
3 clwb(x)
4 sfence()
5 *y = 1;
6 clwb(y)
7 sfence()
```

Listing 1: A PM programming example.

PM implementations support a programming interface that diverges from that of conventional storage devices. Rather than

[1] Released at https://github.com/efeslab/agamotto

using comparatively slow system calls to access persistent memory, applications can accelerate PM accesses by directly mapping pages of PM into their address space and performing byte-addressable load/store operations. Like volatile memory accesses, PM IO may be cached and buffered in volatile memory (i.e., the CPU cache) in order to increase performance.

The added performance comes at the cost of increased complexity for the application developer. Volatile memory can retain updates to PM for an indefinite period of time (e.g., until a cache line gets evicted). Ensuring that stores to PM are durable requires two steps. First, a developer must issue a *flush* for the cache-line that contains the updated data. Then, the developer orders flushes using existing fence operations (e.g., `SFENCE`). Note that an unordered flush may not be written to persistent memory before a crash, so fences are required for durability. Consider Listing 1, which allocates two integers in persistent memory and issues ordered writes to the integers. In order to guarantee that the write to `x` (line 2) is ordered before the write to `y` (line 5), a flush and fence must occur between the updates (lines 3 and 4). To ensure that the write to `y` (line 5) is durable, a flush and fence must occur after the write (lines 6 and 7).

The x86 instruction set architecture (ISA) provides two flush instructions: `CLFLUSHOPT` and `CLWB`. `CLWB` differs from `CLFLUSHOPT` in that `CLWB` hints the CPU to keep the cache line in the cache whereas `CLFLUSHOPT` does not. x86 provides two fence instructions: `MFENCE`, which orders all loads, stores, and flushes; and `SFENCE`, which orders all stores and all flushes. Additionally, x86 provides `CLFLUSH`, which acts as both a flush and fence for a specific cache line (i.e., only orders the flush that the `CLFLUSH` itself issues, other `CLWB` and `CLFLUSHOPT` instructions must be ordered by a separate fence). Finally, x86 allows non-temporal stores, which bypass the cache and thus do not require a flush but do require a fence for durability. Note that the classification of PM instructions into flush and fence operations is not x86-specific. For example, ARM provides flush (e.g., `DC CVAP`) and fence (e.g., `DSB`) operations [5, 67] with similar semantics to x86 flushes and fences.

2.2 Challenges of Detecting PM Bugs

PM interfaces for durability and performance are easy to misuse [49, 50] and the resulting persistency bugs can be challenging to detect. Persistency bugs exhibit many characteristics that make them difficult to detect. First, finding a persistency bug requires identifying whether PM cache-lines are dirty, but the x86 ISA does not provide a mechanism to determine the state of a cache-line. Thus, detecting a persistency bug requires modeling PM state and instrumenting the program for tracking state updates, which is challenging to accomplish using traditional debugging tools. Second, in the case of correctness bugs, the root cause and symptoms of a persistency bug are often loosely tied together: while the symptoms of a correctness persistency bug is only revealed after a crash, the PM misuse (i.e., the root cause) may be hundreds of thousands of instructions before the crash even occurred. Finally, persistency bugs are easily masked by other system behavior. For example, flushes which are redundant in one execution path of the program may be necessary under a slightly different execution path, while correctness persistency bugs may be masked by the CPU when evicting a dirty cache-line from its cache.

Unfortunately, developers cannot solely rely on PM frameworks (e.g., PMDK [20]) to prevent these bugs. As we show in §3, many applications use PM libraries incorrectly and even these established libraries themselves may misuse PM.

Project	Missing Flush/Fence	Extra Flush/Fence	Other	Total
PMDK	49	6	2	57
PMTest	1	1	1	3
XFDetector	-	-	3	3
Total	**50**	**6**	**7**	**63**

Table 1: The results of our bug survey.

3 PM Bug Study and Classification

In this section, we present a study of persistency bugs. We construct a corpus of 63 persistency bugs from a mature PM library, PMDK [20], and persistency bugs from PM projects (PMFS [27] and Redis-pmem [21]) that were found by state-of-the-art PM bug detection tools (PMTest [50] and XFDetector [49]). We chose PMDK, because it is a mature project with a thorough issue tracker [23] representing a large collection of existing bugs. We use this corpus to identify common patterns of PM bugs.

Table 1 shows a summary of our results[2]. Overall, we find that two application-independent PM patterns explain the vast majority (56/63 bugs) of the reported persistency bugs. We find that PM bugs can result in either correctness problems, which may lead to data corruption, or performance problems. In particular, the *missing flush/fence* pattern, in which an update to persistent memory is missing subsequent flush and/or fence operations, accounts for 50/63 bugs and can lead to either correctness or performance issues. The *extra flush/fence* pattern, in which a cache-line is redundantly flushed or a fence instruction is issued that is not needed for PM durability, accounts for 6/63 bugs and leads to performance degradation. The remaining 7 are caused by application-specific violations, most of which involve a misuse of the PMDK transaction API. Note, our study may be biased towards bugs that are detectable by existing PM bug detection tools, because PMDK

[2]We provide a link to our bug study results in the AGAMOTTO GitHub repository: https://github.com/efeslab/agamotto/blob/artifact-eval-osdi20/artifact/README.md

developers extensively use pmcheck [65] to detect bugs. In the rest of this section, we present examples of these bugs together with more detailed descriptions.

3.1 Missing Flush/Fence Pattern

```
1 //oid is a pointer to PM
2 if (if_free != 0)
3   *oid = NULL;
4   // BUG: missing flush and fence
```

Listing 2: A missing flush/fence correctness bug adapted from PMDK Issue #1103, Pull Request (PR) #3907.

The most common bug pattern in the bugs in our study is the missing flush/fence pattern, in part because PMDK developers extensively use pmemcheck [65] which identifies this pattern of PM misuse. In this bug pattern, an update to PM is not made durable because it is missing a subsequent flush and/or fence operation. An example of the pattern is shown in Listing 2. Here, a pointer to persistent memory, `oid`, is not flushed when `if_free != 0`. If the program crashed and restarted, the pointer might point to its old value, which could lead to rogue writes or malformed data reads. This bug is fixed by adding proper flush and fence operations after the modification.

In contrast, the missing flush/fence pattern is detectable without any application-specific information. In our study, instances of the missing flush/fence pattern are correctness issues, where the program is unable to recover from a crash similar to the one in Listing 2. In our evaluation (see §6), we also found instances of the missing flush/fence pattern which are performance bugs. In these instances, an application uses persistent memory to store volatile data, which hinders performance due to the higher latency of PM accesses relative to DRAM accesses. Existing studies suggest that placing volatile data in PM can decrease application performance by as much as 5% [26]. There are PM data structures that intentionally include this pattern [53] as a programming simplification. However, in the applications included in our study and evaluation, all instances of the missing flush/fence pattern are persistency bugs.

3.2 Extra Flush/Fence Pattern

The other common pattern of persistent memory misuse which we identify in our study is the extra flush/fence pattern. In this pattern, a cache-line is redundantly flushed, or a fence instruction which is not needed for PM durability is executed. An example of this is shown in Listing 3. In this example, an array located in persistent memory is resized in-place using the call to `resize_array`, new elements are initialized to 0, and new elements are flushed to persistent memory. However, when the size of the array is reduced (i.e., `new_size < size`), an underflow in line 12 causes unnecessary flushes and leads to a performance degradation [18, 60, 71, 76] (e.g., an additional flush and fence can add an average of 250ns of latency [51, 73], where the base latency of uncached PM accesses can be as low as 96ns [37]).

```
1  //array is an array of integers in PM
2  //with length = size
3
4  //resizes array in-place
5  resize_array(array, new_size);
6
7  // if size >= new_size, no copying occurs
8  for (size_t i = size; i < new_size; i++)
9      array[i] = 0;
10
11 // BUG: when new_size < size, underflow!
12 for (size_t i = 0; i < new_size - size; ++i)
13     clwb(array[i + size])
14 sfence();
```

Listing 3: An extra flush/fence performance bug adapted from PMDK issue #1117, PR #3860 .

Similar to the missing flush/fence pattern, the extra flush/fence pattern is detectable without any application-specific information. The extra flush/fence pattern results in performance degradation. As flush and fence instructions are used in non-PM contexts (e.g., fences provide semantics for memory consistency), there may be instances of this pattern that are not persitency bugs. However, in the applications in our study and evaluation, all instances of the extra flush/fence pattern are persistency bugs.

3.3 Other Bugs

```
1 // store pool's header
2 /* BUG: header made valid before
3   pool data made valid */
4 header = ...
5 clwb(header);
6 sfence();
7 pool = ...
8 clwb(pool);
9 sfence();
```

Listing 4: An example correctness bug adapted from PMDK Issue #14.

The remaining 7 bugs in the study are application-specific; i.e., in these cases, data is correctly flushed to PM and there are no redundant flush operations, but the application misuses PM, leading to performance or correctness issues. For example, Listing 4 depicts a bug adapted from the memory pool allocator in PMDK which results in a correctness issue. In order to recover from a crash, the values in `header` and `pool` must be consistent; however a crash at Line 7 will result in an updated value of `header` without an updated value of `pool`.

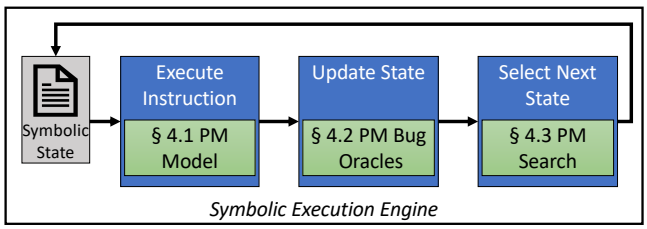

Figure 1: Components of AGAMOTTO. Green-shaded boxes are AGAMOTTO-specific.

3.4 Summary and Insights

We summarize several key results we obtained and the insights we gathered from this bug study which inform AGAMOTTO's design decisions.

- The missing flush/fence and extra flush patterns are prevalent (56/63 of the bugs we found) and application-independent. Hence, an automated approach (i.e., requiring little to no developer effort or source modification) could and should be used to detect them across a variety of platforms.
- In our study, all instances of the missing flush/fence and extra flush/fence patterns are persistency bugs; we hypothesize that this trend will hold for general PM applications. In §6, we find that all instances of these patterns are persistency bugs across a variety of PM libraries and applications.
- The remaining bugs, while less prevalent in our survey, are still potential sources of inconsistency and/or performance loss. An ideal tool should allow developers to specify application-specific patterns without requiring extensive test cases and significant developer annotations.

4 Design

In this section, we describe the design of AGAMOTTO. AGAMOTTO aims to achieve four high-level design principles:

Automation. Bug-finding can take a substantial amount of developer effort [56,68]; AGAMOTTO aims to automate as much as possible to reduce this burden. For example, AGAMOTTO is non-intrusive (i.e., requires no source-code modifications) and leverages basic test cases (e.g., existing unit tests or example code) to explore execution paths in an application.

Generality. AGAMOTTO can test any PM application.

High Accuracy. AGAMOTTO aims to report no false positives (i.e., reporting a bug where there is none) while also reducing false negatives (i.e., failure to find a bug).

Extensibility. AGAMOTTO can be easily extended to find application-specific bugs.

The major components of AGAMOTTO are shown in Fig. 1 (green-shaded boxes represent the key components unique to AGAMOTTO). AGAMOTTO relies on an existing symbolic execution engine (KLEE [8] in our prototype) to explore the state space of a PM program. During this exploration, AGAMOTTO uses a custom PM model to express and track updates to persistent memory regions (i.e., writes, flushes and fences). Since AGAMOTTO tracks PM symbolically, it does not need access to PM resources in order to detect persistency bugs in a PM application. As AGAMOTTO explores the state space of the program, it checks for PM bugs using universal bug oracles, as well as any custom bug oracles that users may provide. Universal oracles check for the missing flush/fence pattern and the extra flush/fence patterns of PM misuse identified in our study. Custom oracles can check for application-specific bugs, which may be correctness bugs (e.g., ordering bugs) and/or performance bugs (e.g., redundant transaction operations) akin to prior work [49,50].

At the heart of AGAMOTTO lies its PM-aware state space exploration algorithm, which is effective in steering symbolic execution towards program locations that exercise PM. In symbolic execution, inputs are symbolic (unconstrained) values in a program's initial state. When the program reaches a branch depending on symbolic input, the current state is forked and the constraints on input are updated depending on the branch condition. As states increase by forking, symbolic execution needs to employ a state-space exploration strategy. Existing state space exploration strategies, such as maximizing code coverage, are not optimized for finding PM bugs, and thus waste resources exploring uninteresting paths.

Instead, before symbolically executing the program, AGAMOTTO uses a custom static analysis to determine instructions that can modify persistent memory. AGAMOTTO then uses a back-propagation algorithm to assign a weight to each instruction equal to the number of PM-modifying instructions that are reachable from that instruction. AGAMOTTO prioritizes exploring the program state whose currently-executed instruction has the highest such weight. We find that the number of PM-modifying paths is much smaller than the total number of execution paths in practice, allowing AGAMOTTO to thoroughly explore the set of executions that lead to persistency bugs (see §6).

When AGAMOTTO's oracles detect a bug during state space exploration, AGAMOTTO relies on its underlying symbolic execution engine to invoke a constraint solver and determine the inputs that led to the bug, thereby creating a test case that a developer can use for debugging.

In the rest of this section we provide details regarding the key components of AGAMOTTO.

4.1 PM Model and PM State Tracking

AGAMOTTO facilitates persistency bug detection by tracking the state of persistent memory objects in the program. For each PM allocation, AGAMOTTO tracks constraints on the *persistency state* of the allocated cache lines. The persistency

state of a cache-line indicates whether the cache line is *dirty* (i.e., modified), *pending* (i.e., updates to the cache-line are flushed but not ordered) or *clean* (i.e., updates to the cache-line are both flushed and ordered). As AGAMOTTO symbolically executes, it updates constraints on the persistency state of PM cache-lines to reflect the behavior of the program. AGAMOTTO uses these constraints to identify execution paths which contain persistency bugs, (i.e., when redundant flushes are issued, or updates are not properly ordered).

Identifying PM allocations In order to be application-agnostic and automated, AGAMOTTO tracks persistent memory allocations from the system level, rather than tracking high-level calls to persistent memory allocators (e.g., `pmem_alloc`) [50]. Tracking PM allocations at a system level trades off performance in favor of automation, since this approach over-approximates PM allocations. AGAMOTTO marks all opened files that match a user-specified persistent memory device regular expression (e.g., `pmem/*`) as PM files and treats memory-mappings of PM files as persistent memory objects.

Tracking Persistent Memory State. When AGAMOTTO symbolically executes an instruction that operates on a PM object, it generates constraints on the persistency state of the cache-lines that comprise the memory objects. A store instruction (e.g., x86 `MOV`) adds a constraint that the destination of the store is in the dirty state. Flush instructions (e.g., `CLWB` and `CLFLUSHOPT`) generate a constraint that denotes that the destination is in the pending state. Non-temporal stores (e.g., x86 `MOVNT` are similar to regular stores, except their destination is immediately put into the pending state (i.e., non-temporal stores are treated as a store+flush), as non-temporal stores bypass the CPU cache but are weakly ordered (like flush instructions) and still require some form of memory fence. Global fences (e.g., `SFENCE`, `MFENCE`) add constraints to indicate that all PM cache lines are clean, whereas cache-line fences (e.g., `CLFLUSH`) add a constraint denoting that their destination is clean.

4.2 Persistency Bug Oracles

AGAMOTTO uses the persistent memory state in order to support two types of persistency bug oracles. First, AGAMOTTO provides two built-in *Universal Peristency Bug Oracles*, which check for bugs based on the patterns we identify in §3. Second, AGAMOTTO allows developers to specify custom, application-specific persistency bug oracles, which we have used to provide two oracles for the PMDK Transaction interface [20].

```
// Unflushed Bug Oracle
def check_unflushed(state):
    for pm_obj in state:
        forall cachelines in pm_obj:
            if not cacheline.is_clean:
                raise error(correctness)

// Extra flush/fence Bug Oracle
def check_extra_flush(state, cacheline):
    if cacheline in state is clean:
        raise error(performance)
def check_extra_fence(state):
    if state has no pending updates:
        raise error(performance)

// Call Oracles on instructions:
def executeInstruction(state, inst):
    if (state.terminated or state.unmapped):
        check_unflushed(state)
    if inst is flush:
        check_extra_flush(state,
                          inst.cacheline)
        // do flush
    if inst is fence:
        check_extra_fence(state)
        state.commit_pending()
```

Listing 5: Pseudo-code for Universal Persistency Bug Oracles and how they are used as AGAMOTTO explores the state space.

4.2.1 Universal Persistency Bug Oracles

AGAMOTTO provides two universal persistency bug oracles, one that detects an instance of the missing flush/fence bug pattern (indicating a correctness or performance bug), and one that detects an instance of the extraneous flush/fence bug pattern (indicating a performance bug). We sketch the algorithms in Listing 5. AGAMOTTO reports a missing flush/fence bug for each cache-line in a persistent memory object that is not clean (i.e., the constraints on the persistent state indicate that the cache-line may be dirty or pending) at the time when the persistent memory is no longer addressable (due to either `munmap` or program exit). AGAMOTTO identifies an extraneous flush/fence operation bug on any flush (e.g., `CLFLUSH`) to a cache-line which must already be pending or clean based on the constraints on the persistent state. AGAMOTTO also identifies an extraneous flush/fence bug on any fence (e.g., `SFENCE` or `MFENCE`) which has no pending flushes to mark clean. For both of these oracles, AGAMOTTO reports program location information (e.g., stack frame and source code location) for the most recent update to each cache line which violates the conditions checked by the oracle. In our evaluation (see §6), we show that these oracles do not incur any false positives across a variety of PM frameworks and applications.

4.2.2 Custom Bug Oracles

In addition to the generic bug oracles, AGAMOTTO facilitates the use of custom bug oracles. Custom bug oracles are defined separately from the application, which allows them to be versatile tools for detecting application-specific bugs. For example, a developer might use a custom oracle to validate the correct usage of PM frameworks (e.g., identifying duplicate log entries in the PMDK `libpmemlog`) or assert that certain

```
class PmemObjTxAddChecker
    : public CustomChecker {
  bool in_tx;
  // [address, address+size)
  typedef pair<ref<Expr>, ref<Expr>> TxRange;
  list<TxRange> added_ranges;

  void checkTxBegin(Function *f,
                    ExecutionState &state) {
    if (!in_tx && f->getName() == "
    pmemobj_tx_begin")
      in_tx = true;
  }

  void checkTxAdd(Function *f,
                  ExecutionState &state) {
    if (f->getName() !=
        "pmemobj_tx_add_common") return;
    // 1. Get the address from the stack.
    ref<Expr> address = f.getArgument(0);
    ref<Expr> size = f.getArgument(1)
    // 2. Get end bound
    auto r_end = address + size;
    auto new_range = TxRange(address, r_end);
    // 3. Check for overlaps.
    //    If overlap, there's a bug!
    if (overlaps(state, new_range))
      reportError(state, RedundantTxAdd);
    // 4. Add the new range.
    added_ranges.push_back(new_range);
  }

  void checkTxEnd(Function *f,
                  ExecutionState &state) {
    if (f->getName() == "pmemobj_tx_end")
      in_tx = false;
  }

public:
  PmemObjTxAddChecker(...) {...}
  // This is the entry point
  virtual void operator()(
    ExecutionState &state) override {
    checkTxBegin(getFunction(state), state);
    checkTxAdd(getFunction(state), state);
    checkTxEnd(getFunction(state), state);

    if (!in_tx) added_ranges.clear();
  }
};
```

Listing 6: An psuedo-code example of a custom oracle, designed to check for redundant PMDK transaction "adds" (i.e., redundant log updates).

structures are operated on in the correct way (e.g., checking that PM referenced as `struct foo` is only ever modified in a PMDK transaction). Custom bug oracles define a function that takes as input an explored program state (i.e., the current state of symbolic memory and variables in the program) and an instruction; after each instruction is executed within this state, AGAMOTTO calls all configured custom bug oracles. We provide two case studies on designing and implementing custom oracles, which we use to find 4 application-specific bugs that were reported by prior work and 1 new application-specific bug. Both of the custom oracles which we present are precise, i.e., they do not introduce false positives. We describe them at a high-level below, then discuss their implementation in §5.

Redundant Undo Log Oracle. This oracle checks to ensure that data does not get logged in PMDK's undo log mechanism multiple times. We show a pseudo-code example of an oracle in Listing 6. PMDK's transactional API implements an undo log which is used to back up data before it is modified—if a transaction is interrupted by a program error or a crash, the data can be recovered from the log. A misuse of this API, however, can lead to redundant entries being created in the undo log, which degrades performance. To track these errors, this oracle keeps track of transaction boundaries (`TX_BEGIN`, `TX_END`) and the memory ranges backed up in the undo log. If overlapping memory ranges are added during a single transaction, the oracle signals a performance bug. We use this oracle to reproduce the application-specific performance bug found by PMTest in PMDK's example B-tree data structure.

Atomic Operation Oracle. This oracle ensures that a developer-specified structure is crash-recoverable through correct use of a PMDK transaction. In particular, the oracle verifies that the structure is only updated within a PMDK transaction and is properly added to the PMDK undo log. We used this oracle to find 3 existing bugs; 2 in the PMDK Atomic Hashmap and 1 in Redis-pmem.

4.3 PM-Aware Search Algorithm

AGAMOTTO uses symbolic execution to explore the state space of the program. In order to analyze large persistent memory applications, AGAMOTTO prioritizes exploring program states that are most likely to modify persistent memory using a PM-aware search algorithm. We now first explain the static analysis that AGAMOTTO uses to compute exploration priorities. We then explain the operation of AGAMOTTO's state space exploration and why AGAMOTTO's approach is more effective at finding persistency bugs than traditional coverage-guided exploration heuristics.

4.3.1 Whole-Program Static Priority Computation

The goal of AGAMOTTO's static analysis is to determine the number of reachable PM-modifying instructions from each instruction in the program. That way, AGAMOTTO can guide symbolic execution towards program locations that are expected to access PM heavily, and uncover more bugs. This technique can be effective as the number of overall instructions expected to modify PM is much smaller than the number of instructions which modify volatile memory [59].

To achieve this, AGAMOTTO first identifies all PM-modifying instructions in the program by leveraging a sound, whole-program (i.e., interprocedural) pointer analysis [4, 14, 31, 32]. The analysis maps each pointer in the program to a set of memory locations; soundness guarantees that any two pointers which may alias will have a non-empty intersection of these sets of memory locations.

```
1  char *pbuf = mmap(<PM file>);
2  ...          // (# of PM-modifying insts)
3  do_read = ...                          // (2)
4  if (do_read)                           // (0)
5      a = pbuf[x]                        // (0)
6      foo()                              // (0)
7  else                                   // (2)
8      a = ...                            // (2)
9      pbuf[x] = a                        // (2)
10     clwb(pbuf[x])                      // (1)
11     // BUG: Missing sfence!
12 exit(0)                                // (0)
```

Listing 7: An example of AGAMOTTO's static analysis. All PM-modifying instructions are highlighted. Each instruction is annotated with a comment which denotes the result of the priority calculation.

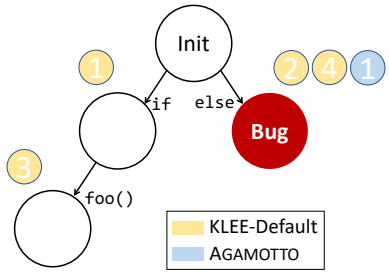

Figure 2: State space exploration with two strategies: (1) KLEE-Default (based on code coverage), (2) AGAMOTTO's priority-driven exploration. This example corresponds with the bug described in Listing 7.

AGAMOTTO then determines whether a given memory location may have been allocated as persistent memory. To do this, AGAMOTTO conservatively assumes that all `mmap` calls which accept a non-negative or variable file descriptor may return a pointer to persistent memory. While this approach over-approximates the persistent memory allocated by the program, as we show in §6, it accelerates persistency bug finding compared to default exploration strategies. Note that this conservative approach only affects the PM-aware search strategy, it does not introduce false positives in AGAMOTTO's PM state tracking.

Then, AGAMOTTO classifies each instruction in the program as a persistent memory-modifying instruction if the instruction is a global fence (e.g., `SFENCE`), or, a store (e.g., x86 `MOV`), flush (e.g., `CLWB`), or cache-line fence (e.g., `CLFLUSH`) that may point to a persistent memory location.

AGAMOTTO only computes points-to information for pointers which may alias PM. For shared libraries, AGAMOTTO first statically links the binary, then computes the alias information. If the shared library is used to modify PM (i.e., has some shared memory modification function which is used to modify PM), then that part of the shared library code will be analyzed.

Finally, AGAMOTTO uses a back-propagation algorithm to calculate the number of reachable PM modifying instructions for each program location. AGAMOTTO iterates through the interprocedural control flow graph from the exit points in the program (e.g., calls to `exit` or `return` from `main`) to the first instruction in the program. For each instruction, AGAMOTTO assigns the *priority* of the instruction to be the sum of the *weight* of the current instruction (1 if the current instruction is a PM-modifying instruction, 0 otherwise) and the maximum number of reachable PM-modifying instructions from the current instruction.

We show a small example of this priority computation in Listing 7, where each instruction is annotated with the result of the priority calculation. Each PM-modifying instruction (`pbuf[x]=a` and `clwb(pbuf[x])`) adds 1 to the priority and the priorities are backpropagated to the entry point (Line 3).

4.3.2 State Exploration Strategy

AGAMOTTO relies on an existing symbolic execution engine (KLEE [8]) to explore the possible states of the program. Symbolic execution starts with an initial program state which contains a current statement (similar to a program counter), a symbolic memory (where memory values are unknown), and symbolic inputs (e.g., an unknown `integer` value). As the program statements are symbolically executed, the symbolic execution engine simulates the effects of the program statements on symbolic inputs and memory, and updates explored program state accordingly. Moreover, the symbolic execution engine forks the explored state into two every time a branch that depends on symbolic values is encountered.

After executing a program statement in an explored state, the symbolic execution engine selects a new state to advance next. When selecting a state to explore, AGAMOTTO chooses the state whose current statement has the highest statically-computed aggregate priority (i.e., number of reachable PM modifying statements from the current instruction).

Fig. 2 shows an example of state space exploration for the the example code snippet in Listing 7, where Init represents the initial state of the program and the buggy state where the program omitted an `sfence` instruction is in the `else` path. For brevity, `foo` is depicted as a single statement that is explored at once.

The KLEE-Default strategy, which is a breadth-first exploration strategy augmented by randomized, coverage-guided prioritization, may explore states that are not useful to detecting the bug. When applied to the code in Listing 7, the KLEE-Default exploration strategy will explore the state in the `if` branch for a single statement (`a=pbuf[x]`) and switch to the state in the `else` branch for another statement (`a=...`). This cycle will repeat once more in the `if` branch (`foo()`) and in the else branch (`pbuf[x]=a, clwb(pbuf[x])`); exploration will reach the bug in a total of 4 state transitions.

AGAMOTTO, on the other hand, directly explores the `else` branch because its static analysis assigns the `else` branch a high aggregate priority. Consequently, AGAMOTTO can

discover the bug with a single state transition.

Although the number of explored states in our example is small, in practice, the number of states in a program is exponential in the number of branches that depend on symbolic input. Consequently, AGAMOTTO's exploration strategy allows it to discover many more bugs compared to KLEE's default strategy, as we demonstrate in §6.

5 Implementation

AGAMOTTO comprises a persistent memory model (~400 LOC of C++), a static analysis component (~2600 LOC of C++), and a state space exploration component (~100 LOC of C++) built atop Klee [8]). AGAMOTTO also provides 2 custom bug oracles for validating the use of the PMDK transaction API (~180 LOC of C++ for both oracles and ~200 LOC of C++ for shared custom oracle API functions).

Running real-world complex PM applications also required expanding KLEE by ~4000 LOC of C++. These additional changes were primarily to the *environment model*, which symbolically simulates syscalls and operating system facilities, such as a file system. AGAMOTTO targets the Intel x86 ISA since it is the most broadly-used platform for PM programming. Hence, AGAMOTTO adds support to KLEE for interpreting PM-specific x86 instructions (e.g., `CLWB`). Supporting a different ISA or persistency model [34, 42, 63] simply requires identifying the flush and fence operations in the ISA. In addition, AGAMOTTO adds to KLEE support for common inline assembly functions such as atomic instructions, as well as porting an extensive environment model for multithreading (i.e., POSIX threads) from Cloud9 [16], which was built on an older version of KLEE. AGAMOTTO adds support for symbolic files to model and track the state of mapped persistent memory and anonymous symbolic `mmap`. Finally, AGAMOTTO adds symbolic socket traffic to the environment model, which allows an application to receive symbolic input over a socket. Symbolic socket traffic allows AGAMOTTO to model client applications that send commands to a server process.

Developing an automated bug finding tool for persistent memory presents key challenges. To identify persistent memory allocations in a PM framework agnostic way without relying on developer annotations, AGAMOTTO tracks allocations at the system level (e.g., calls to map a persistent memory file). This represents a significant divergence from KLEE, which tracks allocations at the libc interface (e.g., `malloc` and `free`), and introduces performance challenges. Applications often allocate MBs or GBs of persistent memory, but KLEE is optimized for tracking memory objects that are KBs in size; treating each persistent memory mapping as a single memory object leads to poor performance when KLEE solves constraints. Instead, AGAMOTTO carefully partitions persistent memory into separate, yet logically adjacent, objects (empirically, we find 16KB chunks to balance the tradeoff between solver time and management overhead). AGAMOTTO also tracks the set of live persistent memory objects to reduce time resolving symbolic addresses for global fence operations.

AGAMOTTO supports custom persistency bug checkers with a simple yet powerful interface. Specifically, a developer implements a method that takes as input the state being explored symbolically and asserts pre- and post- conditions on the state of persistent memory based on an understanding of how their application should behave. AGAMOTTO provides a library of basic utilities (e.g., error reporting, calls to the symbolic solver) that comprise ~200 LOC and allows bug oracles to use type information provided by LLVM. AGAMOTTO provides 2 custom oracles to detect application-specific persistency bugs in PMDK and Redis (§4.2.2). We implement the *Redundant Undo Log Oracle* in 96 LOC and less than a day of developer effort. The *Atomic Operation Oracle* extends the Redundant Undo Log Oracle—it comprises an additional 86 LOC on top of the inherited functionality and also took less than a day to implement.

6 Evaluation

In this section, we evaluate the effectiveness and usefulness of AGAMOTTO. We start by giving an overview of the new bugs AGAMOTTO has found (84)[3] and the insights we gather from them (§6.1). We also discuss the positive responses that we have received after reporting bugs to PM application developers (§6.2). We then evaluate the performance of AGAMOTTO and how our novel search tactic compares to the default symbolic execution search strategy in KLEE (§6.3).

Evaluation Targets. We evaluate AGAMOTTO by testing representative state-of-the-art PM-application and libraries consistent with the libraries and applications tested by prior work [49, 50]. We evaluate AGAMOTTO on two PM libraries. First, we test the PMDK [20] library from Intel, the most active and well-maintained open-source PM project, which has been maintained for over 6 years. Consistent with existing tools [50], we use example data structures provided with PMDK (e.g., B-tree, RB-tree and hashmap implementations) and an application provided by Intel [22] as drivers for our testing. In addition to PMDK, we test NVM-Direct, a PM library from Oracle that is under active development. To drive our testing of NVM-Direct, we use their example test application they provide for demonstrating the API.

We additionally evaluate AGAMOTTO by testing three real-world PM applications. We test Redis-pmem, a port of Redis, a popular in-memory database and memory caching service, to PMDK that is maintained by Intel. We likewise select memcached-pm, a port of memcached, a popular high-performance memory caching server, to PMDK that is main-

[3] We provide a link to our evaluations results in the AGAMOTTO GitHub repository: https://github.com/efeslab/agamotto/blob/artifact-eval-osdi20/artifact/README.md

System	Source (GitHub)	Version
PMDK	pmem/pmdk	v1.8
RECIPE	utsaslab/RECIPE/tree/pmdk	53923cf
memcached-pm	lenovo/memcached-pmem	8f121f6
NVM Direct	oracle/nvm-direct	51f347c
Redis-pmem	pmem/pmem-redis	cc54b55
	pmem/redis	v3.2

Table 2: Software configuration; we tested two versions of Redis-pmem

tained by Lenovo. Finally we test RECIPE's P-CLHT index, a state-of-the-art persistent index representing a research prototype. Note, we only test the P-CLHT index from RECIPE because the other four indices all use a volatile allocator which prevents crash-consistency. Since KLEE symbolically emulates system calls without running real kernel code, we are unable to test PMFS [27], an evaluation target that has been considered by prior work [50].

We test each application by providing a symbolic environment model (e.g., providing symbolic arguments and files with symbolic contents) rather than instrumenting the source code to create symbolic variables. We test RECIPE's P-CLHT index using their example application, which manipulates the basic structure of the index through standard insertion, deletion, and lookup operations. We use symbolic socket traffic (See §5) to test the Redis-pmem and memcached-pm server daemons using partially symbolic packets (i.e., packets with some concrete values, like the Redis command string, with symbolic values for the keys and values).

When testing applications that use PMDK (PMDK, Redis-pmem, and RECIPE), we enable both universal bug oracles and our two custom bug oracles designed for PMDK (see §4.2.2). When testing NVM-Direct, we only use the universal bug oracles.

When using AGAMOTTO to test an application, AGAMOTTO also tracks all persistent memory use from the libraries used by the application. In the case that AGAMOTTO finds a bug in PMDK while testing an application which uses PMDK (e.g., memcached-pm, Redis-pmem, or RECIPE), we report the bug as a bug in PMDK.

Evaluation Setup. We ran our experiments across two servers, one with a Intel(R) Xeon(R) Silver 4114 CPU @ 2.20GHz and one with a Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz. Each individual experiment (a single run of AGAMOTTO) was limited to a max of 10 GB of DRAM and 1 hour of runtime. We show our software configuration in Table 2. Note that none of our experiments use persistent memory hardware since AGAMOTTO symbolically models all interactions with persistent memory.

	MC		MP		EP		AS		Total	
System	N	K	N	K	N	K	N	K	N	K
memcached-pm	1	-	19	-	1	-	-	-	21	-
NVM-Direct	7	-	7	-	9	-	-	-	23	-
PMDK	1	1	14	-	6	-	1	3	22	4
RECIPE	1	-	7	-	6	-	-	-	14	-
Redis-pmem	3	-	1	-	-	-	-	1	4	1
Total	**13**	**1**	**48**	**-**	**22**	**-**	**1**	**4**	**84**	**5**

Table 3: The Bugs found using AGAMOTTO. For each bug class (**MC**: Missing flush/fence Correctness, **MP**: Missing flush/fence Performance, **EP**: Extra flush/fence Performance, and **AS**: Application-Specific), we report the number of new bugs AGAMOTTO found, **N**, and the number of bugs detected that were previously known, **K**.

6.1 Overview

We show a summary of our bug-finding results in Table 3[4]. Overall, AGAMOTTO found 84 new bugs across our 5 main test targets: 62 missing flush/fence bugs (13 correctness bugs and 48 performance bugs), 22 extra flush/fence performance bugs and 1 new application-specific correctness bug. We also detect all 5 persistency bugs found by prior work in user-space applications and confirm that we find no false positives with our universal or custom oracles. Here, we describe the bugs that we find in greater detail.

Missing flush/fence bugs. Using our built-in unflushed bug oracle, we found 62 new bugs; we manually identified that 13 are correctness bugs and 48 are performance bugs. Of the 13 correctness bugs, 10 are caused by missing flushes and 3 are caused by missing fences—all of the missing fence bugs are found in Redis-pmem. AGAMOTTO found the missing flush/fence bug in PMDK that was reported by PMTest. Of the correctness bugs, AGAMOTTO finds 1 in memcached-pm, 1 in PMDK, 1 in RECIPE's P-CLHT index, 7 in NVM-Direct, and 3 in Redis-pmem. Of the performance bugs, AGAMOTTO finds 19 in memcached-pm, 14 in PMDK, 7 in RECIPE's P-CLHT index, 7 in NVM-Direct, and 1 in Redis-pmem.

Extra flush/fence bugs. We found 22 new bugs using the extra flush/fence bug oracle. Of these bugs, AGAMOTTO found 9 in NVM-Direct, 6 in PMDK library functions and 6 in RECIPE's P-CLHT index.

Application-specific bugs. AGAMOTTO identified 1 new application-specific correctness bug in the PMDK atomic hashmap example using the extra flush/fence universal bug oracle. Using the atomic operation oracle, AGAMOTTO found all

[4]We provide the full detailed table in an online table available here: https://github.com/efeslab/agamotto/tree/artifact-eval-osdi20/artifact#resources.

3 application-specific correctness bugs which were reported by XFDetector[5] Using the redundant undo log oracle, AGAMOTTO detected the application-specific performance bug in the PMDK example B-tree structure that was discovered by PMTest. AGAMOTTO is unable to find the application-specific performance bug that PMTest found in PMFS because AGAMOTTO is unable to execute kernel code.

6.2 AGAMOTTO Reporting

We presented our initial results to Intel's PMDK team, Oracle's NVM-Direct team, and to the authors of RECIPE and received overall positive feedback. At the time of writing, we have not yet heard back from Lenovo developers regarding bugs in memcached-pm. PMDK developers confirmed our findings about performance issues. Oracle's developers confirmed they were aware of some of the issues we reported and noted that "Resources for software development are always in short supply, so the open source version of NVM_Direct has suffered. I wish it was not so, but it is. Your email may be the push that gets us to do something about it. Thank you." RECIPE's authors confirmed and started patching all the bugs we reported to them and asked us to open-source AGAMOTTO for continued testing. Despite existing tools for testing PM (one of which was even built for RECIPE [45]), one of RECIPE's authors stated that "These are some really good finds, since it was difficult to debug our own code without having a proper tool."

We conclude that AGAMOTTO has been successful in finding bugs that developers care about.

6.3 Performance Analysis

Benefit of AGAMOTTO's State Exploration Strategy. We evaluate AGAMOTTO's state exploration strategy compared to the default search strategy in KLEE. We compare these two strategies for all of our 5 test targets: memcached-pm (Fig. 3a), NVM-Direct (Fig. 3b), RECIPE's P-CLHT index (Fig. 3d), on PMDK's libpmemobj examples (Fig. 3c), and on Redis-pmem (Fig. 3e). We run each exploration strategy for one hour, since one hour is short enough to integrate into a development cycle but long enough to cover a substantial number of execution paths. In all cases, AGAMOTTO's search strategy finds all reported bugs in less than 40 minutes. For Redis-pmem, the bugs we detect were exposed quickly, allowing both strategies to find all 4 in under 3 minutes. For all of our tests, AGAMOTTO is able to find at least one bug in under 5 minutes, which suggests that AGAMOTTO might even be usable during interactive debugging sessions.

We conclude that AGAMOTTO's static-analysis guided search strategy is more effective in finding bugs than the default state exploration strategy in KLEE.

[5] XFDetector reports 4 new bugs, but one of these bugs is unrelated to persistent memory but detectable with their fault injection framework.

System	Source Size (KLOC)	Dependencies (KLOC)	Static Analysis Run time (min)
memcached-pm	18	36	2.20
NVM-Direct	1	14	0.02
PMDK	2	35	0.60
RECIPE	13	35	0.55
Redis-pmem	54	149	19.6

Table 4: The offline overhead of AGAMOTTO's static analysis. Thousand lines of code (KLOC) is provided for program sources (the driver applications for NVM-Direct and PMDK) and for shared libraries.

Static Analysis Run time. We show the run time of AGAMOTTO's static analysis in Table 4. For most applications we test, the overhead of static analysis is low (less than 4 minutes) relative to the length of time spent finding bugs. Redis-pmem has a larger static analysis run time, particularly due to the number of external libraries it links with—however, the results of the static analysis can be cached across many runs for external libraries.

6.4 Case Study: PM Performance Bugs

Prior works on PM argues for the importance of the performance bugs that are identified by AGAMOTTO. For example, Pelley et al. show that extra flush and fence operations are detrimental to application performance [63], and a study of memcached-pm found that storing volatile data in PM reduces application performance by roughly 5% [26].

To further validate the importance of the performance bugs identified by AGAMOTTO, we perform a performance case study on the P-CLHT data structure from RECIPE. We manually fix the performance bugs and then measure the performance of the data structure on concurrent insert operations, i.e., load operations (each thread inserts new keys into the hash table). We chose insert operations, since they stress the update path on which these bugs were found. We report the performance in Fig. 4. The overall throughput increases dramatically, ranging between 24% to 47%. The main contributor to this throughput increase is moving commonly used locks from PM to DRAM.

7 Related Work

Persistent Memory Frameworks. Crash consistency mechanisms for persistent memory have been considered for years [6, 11, 15, 18, 64]. The difficulty of designing crash-consistent programs for persistent memory has inspired many persistent memory specific crash-consistent frameworks which ease the burden on PM application developers. These frameworks either provide a library interface that can be used in standard programming languages (PMDK [20], NV-Heaps [17], LSNVMM [35]), provide lan-

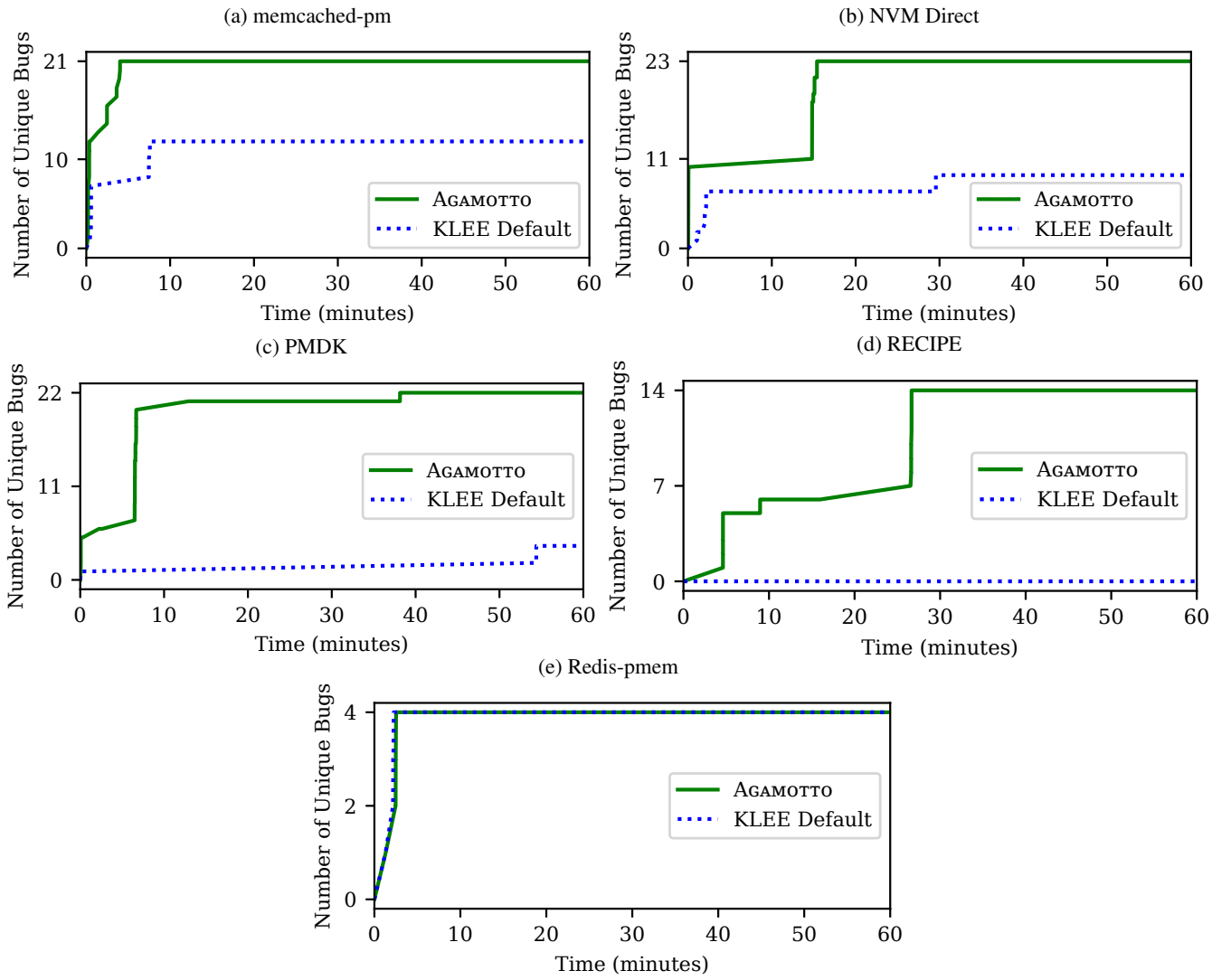

Figure 3: Comparison of the KLEE default search strategy to AGAMOTTO.

guage extensions to augment C/C++ with persistent data types (e.g., Mnemosyne [73], NVL-C [25]), or both (e.g., NVM-Direct [7]). Some systems also use transactional hardware mechanisms to provide more efficient updates to persistent memory (NV-HTM [10], Crafty [30]). However, while these mechanisms may make programming easier, they may still contain persistency bugs. Furthermore, this plethora of PM libraries and extensions motivate the need for generalizable, automated debugging tools.

PM-optimized file systems offer some degree of crash consistency as well [19,27,43,72,74,75], as many PM-optimized file systems offer full-data consistency, rather than just maintaining metadata consistency [9]. However, these mechanisms require the application to use the POSIX interface, as data journaling cannot be efficiently performed for direct-access files. Additionally, applications can suffer from significant performance degredations by acccessing PM through the file system rather than through direct memory mappings [37].

Tools for Detecting Persistency Bugs. The state-of-the-art tools for detecting persistency bugs are PMTest [50] and XFDetector [49]. PMTest is a tracing system which transforms updates to persistent memory into a trace of operations, which is asynchronously validated against programmer-defined rules for persistent memory updates. PMTest is flexible and fast, but requires developer effort to generate persistent memory rules and incurs a high rate of false negatives, as it must be driven by concrete test cases. The authors of PMTest [50] manually instrument applications to find two similar patterns to AGAMOTTO application-independent patterns: the extra flush/fence bug pattern and a delayed flush/fence pattern, in which a delay in the durability of an PM update prevents crash consistency. Delayed flush/fences are inherently application-specific (and thus require developer effort), and there were no delayed flush/fence bugs in our study. XFDetector is a fault injection framework designed to detect cross-failure bugs, which manifest when recovery code accesses

	Agamotto	PMTest	XFDetector	pmemcheck	Persistency Inspector
Core Mechanism	Symbolic Execution	Trace Validation	Fault Injection	Binary Instrumentation	Binary Instrumentation
Accuracy	High	Low	Medium	Low	Low
Automation	High	Low	Medium	Low	Low
Generality	Medium	High	Medium	Very Low	Low
Extensibility	High	High	Low	Low	Low

Table 5: A qualitative comparison between AGAMOTTO and related work, as measured by our design goals (§4).

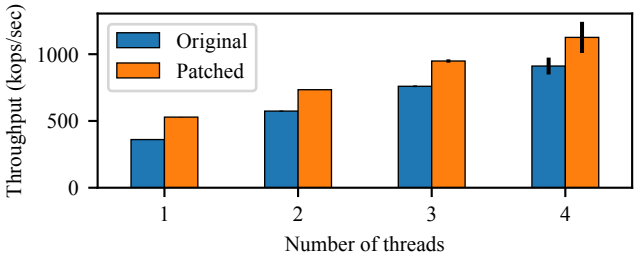

Figure 4: The write throughput (in kilo-operations per second) of the P-CLHT data structure before and after patching performance bugs. "Original" denotes the unmodified P-CLHT structure and "Patched" denotes P-CLHT after we patch the performance bugs.

data which was not guaranteed to be safely persisted before a failure. While XFDetector is effective at detecting semantic bugs with low developer effort, XFDetector still relies on developer-provided concrete test cases. RECIPE [45] uses a PIN-based tool for testing their converted PM indices, which also incurs a high false positive rate due to requiring extensive test cases. pmemcheck [65] and Persistence Inspector [62], which are binary instrumentation tools built by Intel, require a large amount of developer effort to use as they are heavily annotation based. We summarize the high-level feature differences between AGAMOTTO and other persistency bug detection frameworks in Table 5.

Tools for Testing Crash Consistency. Crash consistency testing has been the study of many works on both legacy file systems and PM-optimized file systems [13, 28, 29, 41, 44, 55, 58]. Many of these tools either test for semantic bugs specific to file systems or are only targeted for block-based storage devices. Yat [44] specifically targets crash consistency testing for Intel's persistent memory file system (PMFS [27]). However, Yat tests crash consistency by computing all possible instruction orderings to find crash consistency bugs—a task which can take over 5 years to fully test [44].

Bug Taxonomies. Many papers taxonomize software bugs in other contexts. In the storage context, JUXTA [57] draws a distinction between shallow (roughly equivalent to application-independent) and semantic (application-specific) bugs while CrashMonkey [55] studies the effects and number of operations required to induce crash consistency bugs in file systems. More generally, Li et al. [47] and Liu et al. [48] classify software bugs into universal bug classes (e.g., memory-related, concurrency and incorrect failure handling) and semantic (application-specific) bugs. The key distinction between our study and these prior studies is our focus on persistent memory systems.

The Thread Between Concurrency and Consistency. Several works have identified a similarity in data races [1, 39, 61] in concurrent programs and semantic crash consistency bugs [45, 49]. Traditional data races result in inconsistent data being read across threads of execution, which many systems have been designed to detect and fix [2, 38, 40, 46, 66, 69]. Principles from data race detection have been adapted to build PM crash consistency mechanisms (i.e., in RECIPE [45]) and PM semantic crash consistency detection tools (i.e., XFDetector [49]). When applied to AGAMOTTO, these principles inform the design of custom bug oracles.

8 Conclusion

Persistent Memory (PM) can be used by applications to directly and quickly persist data without the overhead of a file system. However, writing PM applications that are simultaneously efficient and correct is challenging. In this paper, we presented a system for more thoroughly testing PM applications. We informed our design using a detailed study of 63 bugs from popular PM projects. We then identify two application-independent (i.e., universal) patterns of PM misuse which are widespread in PM applications and can be detected automatically.

We then presented AGAMOTTO, a generic and extensible system that leverages symbolic execution for discovering misuse of persistent memory in PM applications. We introduced a new symbolic memory model that is able to represent whether or not PM state has been made persistent, as well as a state space exploration algorithm which can drive AGAMOTTO towards program locations that are susceptible to persistency bugs. We used AGAMOTTO to identify 84 new bugs in 5 different applications and frameworks, all without incurring any false positives and not requiring any source code modifications or extensive test suites.

9 Acknowledgements

We thank the anonymous reviewers and our shepherd, Michael Swift, for their valuable feedback. We also thank Bill Bridge and the Oracle team behind NVM-Direct; Andy Rudoff and the whole PMDK team at Intel; as well as Sekwon Lee, Vijay Chidambaram, and the authors of RECIPE. This work is supported by Applications Driving Architectures (ADA) Research Center (a JUMP Center co-sponsored by SRC and DARPA), the National Science Foundation under grants CNS-1900457 and DGE-1256260, the Texas Systems Research Consortium, the Institute for Information and Communications Technology Planning and Evaluation (IITP) under a grant funded by the Korea government (MSIT) (No. 2019-0-00118), and a Microsoft Ph.D. Fellowship. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

A Artifact Appendix

A.1 Abstract

We provide the public repository for AGAMOTTO, which is a fork of KLEE available on GitHub. AGAMOTTO's artifact includes instructions for building and running AGAMOTTO, as well as a pre-installed VM and scripts used to reproduce the core results from our paper.

A.2 Artifact check-list

- **Public repository link**: https://github.com/efeslab/agamotto/tree/artifact-eval-osdi20/artifact
- **Data:** Links to our bug study findings and to a table describing new bugs found with AGAMOTTO: https://github.com/efeslab/agamotto/tree/artifact-eval-osdi20/artifact#resources
- **Code licenses:** AGAMOTTO inherits KLEE's open source license, which can be read in the repository here: https://github.com/efeslab/agamotto/blob/artifact-eval-osdi20/LICENSE.TXT.

A.3 Description

All information is available at our public GitHub repository. We have written a README specifically for the Artifact Evaluation process, which can be found here: https://github.com/efeslab/agamotto/tree/artifact-eval-osdi20/artifact

A.3.1 How to access

We provide information on how to access our repository and all relevant resources here: https://github.com/efeslab/agamotto/tree/artifact-eval-osdi20/artifact#agamotto-osdi-20-artifact

A.4 Installation

The instructions for compiling AGAMOTTO and installing the prerequisites can be found here: https://github.com/efeslab/agamotto/tree/artifact-eval-osdi20/artifact#artifacts-functional-criteria

A.5 Evaluation and expected result

We provide instructions for reproducing the main results from our paper along with the expected results here: https://github.com/efeslab/agamotto/tree/artifact-eval-osdi20/artifact#results-reproduced.

A.6 Notes

We are endeavoring to maintain AGAMOTTO as an open-source tool for debugging PM applications and hope to encourage its use for a wide variety of applications. Any issues that are found with the available artifact or any needed clarifications can be submitted as GitHub issues on our repository (https://github.com/efeslab/agamotto/issues).

References

[1] Sarita V Adve and Mark D Hill. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and distributed systems*, 4(6):613–624, 1993.

[2] Sarita V Adve, Mark D Hill, Barton P Miller, and Robert HB Netzer. Detecting data races on weak memory systems. *ACM SIGARCH Computer Architecture News*, 19(3):234–243, 1991.

[3] Paul Alcorn. Intel Optane DIMM Pricing. https://www.tomshardware.com/news/intel-optane-dimm-pricing-performance,39007.html, 2019.

[4] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[5] Arm Limited. *Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile*, 2019. https://developer.arm.com/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile.

[6] Joy Arulraj and Andrew Pavlo. How to build a non-volatile memory database management system. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1753–1758, 2017.

[7] Bill Bridge. Nvm-direct library. https://github.com/oracle/nvm-direct, 2015.

[8] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 209–224, USA, 2008. USENIX Association.

[9] Mingming Cao, Suparna Bhattacharya, and Ted Ts'o. Ext4: The next generation of ext2/3 filesystem. In *LSF*, 2007.

[10] Daniel Castro, Paolo Romano, and Joao Barreto. Hardware transactional memory meets memory persistency. *Journal of Parallel and Distributed Computing*, 130:63–79, 2019.

[11] Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. *ACM SIGPLAN Notices*, 49(10):433–452, 2014.

[12] Himanshu Chauhan, Irina Calciu, Vijay Chidambaram, Eric Schkufza, Onur Mutlu, and Pratap Subrahmanyam. NVMOVE: Helping programmers move to byte-based persistence. In *4th Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 16)*, Savannah, GA, November 2016. USENIX Association.

[13] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 18–37, 2015.

[14] Jia Chen. Andersen's pointer analysis. https://github.com/grievejia/andersen.

[15] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 228–243, 2013.

[16] Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. Cloud9: A software testing service. *ACM SIGOPS Operating Systems Review*, 43(4):5–10, 2010.

[17] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *ACM SIGARCH Computer Architecture News*, 39(1):105–118, 2011.

[18] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 133–146. ACM, 2009.

[19] Jonathan Corbet. Supporting filesystems in persistent memory, September 2014.

[20] Intel Corporation. Persistent Memory Programming. https://pmem.io/pmdk/, 2018.

[21] Intel Corporation. Redis. https://github.com/pmem/redis/tree/3.2-nvml, 2018.

[22] Intel Corporation. PMDK Examples for libpmemobj. https://github.com/pmem/pmdk/tree/master/src/examples/libpmemobj, 2020.

[23] Intel Corporation. PMDK Issues. https://github.com/pmem/pmdk/issues, 2020.

[24] Lenovo Corporation. Memcached. https://github.com/lenovo/memcached-pmem, 2018.

[25] Joel E Denny, Seyong Lee, and Jeffrey S Vetter. Nvl-c: Static analysis techniques for efficient, correct programming of non-volatile main memory systems. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 125–136, 2016.

[26] Dormondo. The Volatile Benefit of Persistent Memory: Part Two, May 2019.

[27] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

[28] Daniel Fryer, Mike Qin, Jack Sun, Kah Wai Lee, Angela Demke Brown, and Ashvin Goel. Checking the integrity of transactional mechanisms. *ACM Transactions on Storage (TOS)*, 10(4):1–23, 2014.

[29] Daniel Fryer, Kuei Sun, Rahat Mahmood, TingHao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. Recon: Verifying file system consistency at runtime. *ACM Transactions on Storage (TOS)*, 8(4):1–29, 2012.

[30] Kaan Genç, Michael D. Bond, and Guoqing Harry Xu. Crafty: Efficient, htm-compatible persistent transactions, 2020.

[31] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–299, 2007.

[32] Ben Hardekopf and Calvin Lin. Exploiting pointer and location equivalence to optimize pointer analysis. In *International Static Analysis Symposium*, pages 265–280. Springer, 2007.

[33] Swapnil Haria, Mark D Hill, and Michael M Swift. Mod: Minimally ordered durable datastructures for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 775–788, 2020.

[34] Swapnil Haria, Sanketh Nalli, Michael M Swift, Mark D Hill, Haris Volos, and Kimberly Keeton. Hands-off persistence system (hops). In *Nonvolatile Memories Workshop*, 2017.

[35] Qingda Hu, Jinglei Ren, Anirudh Badam, Jiwu Shu, and Thomas Moscibroda. Log-structured non-volatile main memory. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 703–717, 2017.

[36] Intel. Intel® Optane™ DC Persistent Memory. http://www.intel.com/optanedcpersistentmemory, 2019.

[37] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module, 2019.

[38] Guoliang Jin, Wei Zhang, and Dongdong Deng. Automated concurrency-bug fixing. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 221–236, 2012.

[39] Baris Kasikci, Cristian Zamfir, and George Candea. Data races vs. data race bugs: telling the difference with portend. *ACM SIGPLAN Notices*, 47(4):185–198, 2012.

[40] Baris Kasikci, Cristian Zamfir, and George Candea. Racemob: crowdsourced data race detection. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pages 406–422, 2013.

[41] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 147–161, New York, NY, USA, 2019. Association for Computing Machinery.

[42] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M. Chen, and Thomas F. Wenisch. Delegated persist ordering. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49. IEEE Press, 2016.

[43] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 460–477, New York, NY, USA, 2017. ACM.

[44] Philip Lantz, Subramanya Dulloor, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. Yat: A validation framework for persistent memory software. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 433–438, Philadelphia, PA, June 2014. USENIX Association.

[45] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. RECIPE: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*, Ontario, Canada, October 2019.

[46] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. Efficient scalable thread-safety-violation detection: Finding thousands of concurrency bugs during testing. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 162–180, New York, NY, USA, 2019. Association for Computing Machinery.

[47] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now? an empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06, page 25–33, New York, NY, USA, 2006. Association for Computing Machinery.

[48] Haopeng Liu, Shan Lu, Madan Musuvathi, and Suman Nath. What bugs cause production cloud incidents? In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 155–162, 2019.

[49] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. 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*, pages 1187–1202, 2020.

[50] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. 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*, pages 411–425, 2019.

[51] Youyou Lu, Jiwu Shu, Long Sun, and Onur Mutlu. Loose-ordering consistency for persistent memory. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, pages 216–223. IEEE, 2014.

[52] Pratyush Mahapatra, Mark D. Hill, and Michael M. Swift. Don't persist all : Efficient persistent data structures, 2019.

[53] Pratyush Mahapatra, Mark D. Hill, and Michael M. Swift. Don't persist all : Efficient persistent data structures, 2019.

[54] Virendra J Marathe, Margo Seltzer, Steve Byan, and Tim Harris. Persistent memcached: Bringing legacy code to byte-addressable persistent memory. In *9th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, 2017.

[55] Ashlie Martinez and Vijay Chidambaram. Crashmonkey: A framework to systematically test file-system crash consistency. In *Proceedings of the 9th USENIX Conference on Hot Topics in Storage and File Systems*, pages 6–6. USENIX Association, 2017.

[56] Steve McConnell. *Code Complete*. Microsoft Press, 2004.

[57] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 361–377, 2015.

[58] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding crash-consistency bugs with bounded black-box crash testing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 33–50, 2018.

[59] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. An analysis of persistent memory use with whisper. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 135–148, New York, NY, USA, 2017. Association for Computing Machinery.

[60] Dushyanth Narayanan and Orion Hodson. Whole-system persistence. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, pages 401–410, 2012.

[61] Robert HB Netzer and Barton P Miller. What are race conditions? Some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(1):74–88, 1992.

[62] Kevin Oleary. How to Detect Persistent Memory Programming Errors Using Intel® Inspector - Persistence Inspector, 2018. https://software.intel.com/en-us/articles/detect-persistent-memory-programming-errors-with-intel-inspector-persistence-inspector.

[63] Steven Pelley, Peter M Chen, and Thomas F Wenisch. Memory persistency. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 265–276. IEEE, 2014.

[64] Steven Pelley, Thomas F Wenisch, Brian T Gold, and Bill Bridge. Storage management in the nvram era. *Proceedings of the VLDB Endowment*, 7(2):121–132, 2013.

[65] PMDK. An introduction to pmemcheck. https://pmem.io/2015/07/17/pmemcheck-basic.html.

[66] Eli Pozniansky and Assaf Schuster. Efficient on-the-fly data race detection in multithreaded c++ programs. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 179–190, 2003.

[67] Azalea Raad, John Wickerson, and Viktor Vafeiadis. Weak persistency semantics from the ground up: formalising the persistency semantics of armv8 and transactional models. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–27, 2019.

[68] Capegmini S.A. Capgemini world quality report 2015-2016. https://www.uk.capgemini.com/thought-leadership/world-quality-report-2016-17, 2015.

[69] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[70] Steven Swanson. Early measurements of intel's 3dx-point persistent memory dimms, Apr 2019.

[71] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, pages 5–5. USENIX Association, February 2011.

[72] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 14:1–14:14, New York, NY, USA, 2014. ACM.

[73] Haris Volos, Andres Jaan Tack, and Michael M Swift. Mnemosyne: Lightweight persistent memory. *ACM SIGARCH Computer Architecture News*, 39(1):91–104, 2011.

[74] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, pages 323–338, Berkeley, CA, USA, 2016. USENIX Association.

[75] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. Nova-fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 478–496, 2017.

[76] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 167–181, 2015.