Automating the Detection and Correction of Failures in Modern Persistent Memory Systems

by

Ian Glen Neal

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2023

Ian Glen Neal
iangneal@umich.edu
ORCID iD: 0000-0001-9721-781X

For my wife and daughter

ACKNOWLEDGMENTS

A Ph.D. (from the Latin *Philosophiae Doctor*, meaning "Doctor of Philosophy" in English) is usually the highest degree of education one can achieve in a given field, and is awarded based on the successful completion of both a course of study and on the merit of the original research performed in recipient's chosen field. However, there are many alternative definitions of "Ph.D." that are useful for accurately describing the lived experience of earning one: "Piled Higher and Deeper," "Patiently Hoping for a Degree," and "Praying Hard Daily" are among my favorites. This being said, it is abundantly clear to me that I was only able get this far in my education and research career with the continual support of many fellow researchers, friends, and family members.

I would first like to thank all of my mentors and colleagues who I have worked with throughout the years. Of these, I first give thanks to Graham Mitchell, my high school computer science teacher who helped kick-start my passion for the subject. I then thank Emmett Witchel, the professor at the University of Texas at Austin who gave me multiple opportunities to get involved in computer systems research and inspired me to pursue my graduate education. I then thank Baris Kasikci (my official advisor) and Andrew Quinn (my unofficial advisor), who have both given me countless pieces of advice, coupled with plenty of encouragement, that have gotten me to where I am today: finally confident enough to call myself a researcher. I finally thank all of my collaborators on the wide variety of projects that I have lead and contributed to throughout my graduate career for presenting me with all sorts of interesting ideas, broadening my perspectives, and helping me develop into a more mature engineer and researcher.

My research and this dissertation would not have been possible without the support of my close friends who have provided the support and fun times I have needed for at least some semblance of balance in my life. I thank Zack[1] and Sidd[2] for the numerous shenanigans over the years that have given me[3] many opportunities to unwind and be myself (or, myselves?). I thank Justin and Amberly, and (the other) Zach and Syd for the "GEN-Cons" and all of their emotional and moral support they have provided for my wife and I. I finally thank Dennis, my best man, for keeping me sane and laughing throughout the years (choir practice just isn't the same without you).

[1]Bast, Krieg, Finch, Caulipitus Therian Onzden (a.k.a. Cauthon), Draistian, and Red(acted).

[2]Zara, Phineas, Jellaby, Septimus, Algernon, Clifford, Frederick ("Freddie Boy"), Igneous, and Gustavo.

[3]Justice, Jeffrey, Silverwing, Hurin, Julius, Paul Invictus, Damir, and Felwin Felfan Calbis Turin ("Tayken Runn").

Last and certainly not the least, I thank my family. I thank my parents, Danny and Pamela, and my sister, Audrey, for all of their love and support throughout the years, even when I did crazy things like take internships in Seattle and move to Michigan to go to graduate school. I also thank the Western Michigan COGWA congregation and include them in my thanks to my family as well, since they are my family and have treated me as such since I first walked into the meeting hall back in 2018; I sincerely believe a part of the reason I came to Michigan was to meet and get to know all of you. Most importantly of all, I thank my wife, Ginny, who has been my best friend for almost half of my life at this point and the best wife that any man could ask for. All of my successes in graduate school have happened after we got married, and I believe that there is no coincidence there. I tried to write something about what my life would be without her love and support, but such a reality is beyond my reasoning capabilities at this point. I also thank Sneako, the picky little pet axolotl that we keep together, for the levity and goofy antics he has provided for both of us throughout the last half of my graduate career.

This chapter of my life has been more difficult than anything I have done before. As I rapidly enter into the next stage of my life and reflect, I thank God and each one of you for everything you have done for me throughout my graduate career and throughout my life, as I would not have been able to make it to this point without you. Thank you all so very much.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

LIST OF ACRONYMS

API Application Programming Interface

CPU Central Processing Unit

CXL Compute Express Link

DPOR Dynamic Partial Order Reduction

DRAM Dynamic Random-Access Memory

eADR Extended Asynchronous DRAM Refresh

GB gigabyte

IR Intermediate Representation

ISA Instruction Set Architecture

KB kilobyte

LOC Lines of Code

PM Persistent Memory

PMDK Persistent Memory Development Kit

POR Partial Order Reduction

PR Pull Request

MB megabyte

NVM Non-Volatile Main Memory

YCSB Yahoo! Cloud Serving Benchmark

ABSTRACT

Modern software systems are deeply embedded into our daily lives; the failures of these systems can therefore result in massive real-world harm. Consequently, considerable resources are spent finding and fixing bugs in testing. Overall, the software industry spends billions of dollars each year on fixing bugs, and ultimately loses trillions of dollars each year due to poor software quality (as a result of bugs that escape testing and wreak havoc once deployed).

One particularly challenging domain of software development for developers is the area of Persistent Memory (PM) programming, an abstraction where developers write software that accesses and updates long-term storage with direct memory operations. The PM programming abstraction has become popular in recent years due to new hardware advances in low-latency, byte-addressable storage devices. Unfortunately, writing crash-consistent PM applications is challenging, as untimely program crashes can result in data corruption and loss if the application does not carefully order updates to PM, and testing all possible crashes for data consistency is intractable. Furthermore, crash-consistency bugs are difficult to manually debug and repair, taking weeks or months for a developer to correctly fix. Without advancements in PM testing and program repair tools, developers will be unable to effectively write correct and efficient applications for modern PM platforms, hampering the ease of their adoption.

Motivated by these PM software development challenges, this dissertation explores research in developing software techniques that automate difficult and time-consuming PM development tasks. We study PM system design, bugs, and bugs fixes and observe that we can automatically provide scalable and high-coverage bug detection and correction by approximating the reasoning performed by developers as they develop their applications. Based on this insight, we first explore automated bug detection and correction for PM application bugs caused by the misuse of platform-specific PM primitives. We develop a testing technique that prioritizes testing program paths that heavily modify PM, as these paths are more likely to misuse PM. We implement this technique in AGAMOTTO, a symbolic-execution tool that thoroughly explores PM applications to uncover platform-specific bugs, which we use to find 84 new bugs while incurring no false positives. We then develop a technique for generating fixes for PM platform-specific bugs that are provably correct, coupled with heuristic performance optimizations that do not compromise correctness, and implement the technique in a compiler tool, HIPPOCRATES.

Second, this dissertation explores automated bug detection for general crash-consistency bugs in PM applications (i.e., bugs caused by the improper ordering of PM updates). We develop a technique that automatically identifies groups of PM program behaviors that are likely to result in the same crash-consistency bugs and only tests one behavior out of the group, thus providing high testing accuracy (by testing all types of behaviors thoroughly) while also increasing efficiency (by eliminating redundant testing on functionally-similar behaviors). We implement this technique in SQUINT, a model-checking tool that selectively tests groups of PM program behaviors identified from a dynamic program trace, which we use to find 108 PM crash-consistency bugs.

The works presented in this dissertation provide a holistic automated testing and program repair solution for PM software developers. In sum, these tools have been used to find and fix over two hundred PM bugs in real-world PM systems, demonstrating both the need for such tools and the efficacy of the tools presented in this dissertation.

CHAPTER 1

Introduction

We live in an era where computing is truly ubiquitous, where software abounds and technology is embedded into the daily tasks of many people around the globe. Software is crucial to many of us, but unfortunately, software can experience a wide variety of failures. These failures range from simple programming mistakes, which can be relatively easy to correct before impacting users, to complex ordering problems or performance degradations that may only manifest in a non-deterministic fashion, making it challenging for developers to uncover these errors, even though they can have a large impact on users when they do occur. Consequently, software companies spend considerable time and money testing their software for bugs [113] and then fixing those bugs, costing the software industry billions of dollars each year in finding and fixing bugs [17, 171] and trillions of dollars due to overall poor software quality [87].

While developing new software is tedious and error-prone due to the challenges of finding and fixing bugs, developers must keep up with new software and hardware trends in order to deliver high-quality systems and applications. One such emerging trend in the past few years has been the uptick in applications leveraging PM. Recently, the software industry and software researchers alike have taken renewed interest in PM programming [68, 71, 114, 140], which is a programming abstraction that enables applications to address durable storage using direct memory accesses [148]. PM has experienced this renewed interest due to recent hardware advances in low-latency and byte-addressable storage, such as Intel Optane Pmem [34, 36] and Compute Express Link (CXL)-attached non-volatile storage [27, 52], which now enables the development of highly efficient storage systems using PM hardware and software abstractions [32, 77, 164].

Unfortunately, writing correct PM applications is challenging. First of all, modern PM platforms often require developers to correctly use PM-specific memory ordering primitives (e.g., cache flush and memory fence instructions) to ensure that PM updates are written to the backing storage device; the omission or incorrect use of these primitives can lead to data corruption, data loss, or performance issues. Second, even if all PM updates are properly made persistent, those updates may not be persisted in the correct order, leading to data inconsistencies or data loss if a crash interrupts a PM program's execution.

To combat the cost and effort of testing applications and fixing bugs, there has been increased research and development in software methods that developers can use to automate these tasks. However, there are many challenges in building and using such methods. Testing tools cannot feasibly test for all possible bugs and must therefore minimize the number of tests run to expose the most bugs. Furthermore, program repair tools then must carefully repair programs to ensure produced patches are effective and do not induced further program failures or incur high overhead.

The challenges of finding and fixing bugs are more than apparent for developers of PM applications. PM applications have large state-spaces that traditional crash-consistency testing approaches cannot scale to test, as PM applications make fine-grained updates that result in many more crash-states being generated than can be generated by typical, block-based storage systems [90, 116]. Furthermore, fixing PM bugs once they are found is challenging for the same reason that it is challenging to write correct PM code from the beginning: it is difficult to correctly order PM updates such that the application is both correct and efficient, as more strictly ordered update sequences are easier to reason about during development, but are often less efficient at run-time.

The challenges of building efficient and effective methods to help developers handle software defects in PM systems motivates this dissertation. In this dissertation, we investigate the challenges of testing programs and repairing bugs in modern systems, with a focus on systems with added challenges due to their use of new PM platforms.

1.1 Persistent Memory Background and Challenges

Persistent Memory (PM) is a programming abstraction where programs address durable storage (i.e., data that lives beyond a single execution of a program or boot cycle of a machine) via a memory address space, the same manner in which volatile main memory (e.g., temporary stack variables) is modified [9, 144, 148]. PM is an appealing programming abstraction because long-lasting program data structures can be directly modified without requiring serialization and deserialization through other storage abstractions (e.g., file-system IO calls), simplifying storage-interfacing code and allowing greater code reuse between volatile and non-volatile operations.

PM programming is not a new concept [148], but has seen a growth in research popularity due to advancements in Non-Volatile Main Memory (NVM) hardware (e.g., ReRAM [3], STT-MRAM [6], and most importantly, PCM [157]), as NVMs form the basis for efficient PM programming platforms; a minimalism and highly efficient PM implementation can be to simply allow an application to issue memory operations directly against a byte-addressable NVM. Most notably, Intel has commercialized three versions of PCM-based NVMs in its Intel Optane Persistent Memory line of products [4, 34–36, 38, 69], which can offer PM accesses with latencies that are only 2–3$\times$ higher than the latencies of DRAM [147, 164]. With the use of efficient NVMs, the PM abstraction allows applications to directly interact with persistent data in an efficient manner.

Unfortunately, writing PM applications can be challenging, as it is difficult to ensure that the application is both correct and efficient. There are three general types of crash-consistency bugs in PM applications:

1. **Persistence Bugs** (also referred to as *Platform-Specific Bugs* or *Application-Independent Bugs*). Many PM platforms require applications to issue specific instructions (i.e., *persistence* or *durability* instructions) in order for memory writes to PM to become durable. For example, PM applications that use Intel Optane DC Pmem [35, 36, 38] as their backing storage device often require both cache-line flushes and memory fences to be issues to force PM writes to leave the Central Processing Unit (CPU) cache and be written to the NVM hardware [34, 69, 77, 137]. The misuse of these persistence instructions lead to *persistence bugs*. There are two subtypes of persistence bugs: *durability* bugs, where required persistence instructions are omitted, resulting in data inconsistencies; and *performance* bugs, caused by the unnecessary issuance of these instructions, which can degrade performance. These bugs are considered *application-independent* because these persistence instructions must be used correctly regardless of the application's semantics.

2. **Ordering Bugs** (also referred to as *Application-Specific Bugs*). Even when persistence instructions properly ensure that PM updates become persistent, an application can issue updates in an improper order that, when interrupted from a crash, results in data inconsistencies. Ordering bugs are therefore caused by improper ordering between updates to semantically-related PM data and arise regardless of whether applications include appropriate PM ordering instructions. These bugs are *application-specific* because the proper order of PM updates is dependent on how an application uses the persisted data and what the persisted data represents in the program state, which are application-specific properties.

3. **Compiler Bugs**. Certain compiler optimizations assume that certain types of memory operations may be safely reordered or split apart without affecting the correctness of the application. For example, some compilers assume non-atomic stores may be safely split into multiple smaller stores in a procedure referred to as store tearing [58]. However, while these transformations are safe for volatile-memory algorithms, the ordering and granularity of stores does impact the crash-consistency properties of PM-modifying memory operations. Ultimately, this results in code that, when considered in the unoptimized form written by the developer, is not susceptible to crash-consistency bugs, but is once optimized by the compiler.

Each of these three types of PM bugs can have a variety of different consequences. We list four possible types of failures below:

- **Recovery failure.** Recovery failures occur when a PM application crashes in such a way that it failed to enforce consistency within its state contained in a crash-state that is required by its recovery code. Recovery failures can be caused by any of the three general causes of crash-consistency bugs listed above. Finding recovery failures often requires less application-specific knowledge because they can be automatically detected on an arbitrary PM application (i.e., by attempting to restart the application after a crash).

- **Data corruption.** Alternatively, a PM application may initially recover from a crash, but produce incorrect results or perhaps crash later in an execution due to incomplete modification of internal data structures at the time of the crash. Data corruption bugs can also be difficult to detect without application-specific oracles, because finding a data corruption bug requires knowing an application's intended behavior in the event of a crash (e.g., the expected output of a specific operation).

- **Data loss.** Data loss occurs when an unexpected crash causes data to either not become durable or to become unreachable, resulting in an application state where data goes missing. Data loss bugs can also be difficult to detect without application-specific oracles, because finding a data loss bug also requires knowing an application's intended behavior in the event of a crash (i.e., how much data, if any, can be lost).

- **Performance degradation.** Performance persistence bugs in particular can erode the performance of PM applications. Persistence instructions often operate by enforcing specific memory orderings in the CPU's execution, which restricts the CPU's memory parallelism.

These PM bugs can be difficult to both find and fix during the development of PM applications. Due to the complexity and impact that PM bugs can have, we are motivated to investigate solutions for automating the detection and correction of PM bugs in emerging PM storage systems.

1.2 Limitations of Prior Work

Finding PM Bugs While research in software engineering has explored automated bug detection for a wide variety of different types of bugs and types of applications, finding PM crash-consistency bugs requires specialized tooling to detect platform-specific PM and cope with the state space of PM applications. Prior work has created tools and techniques for testing modern PM applications, however existing approaches force developers to chose between **scalability** and **coverage** during testing, limiting the practical usability of these tools. Existing approaches that scale to testing large systems sacrifice testing coverage by dramatically limiting the scope of bugs they can find, resulting in many false negatives (i.e., missing bugs) [31, 41, 50–52, 56, 103, 104, 126]. On the other hand, existing approaches that provide high testing coverage by exhaustively testing all possible ways that a PM application could crash and check if these crashes result in data inconsistencies or the inability to recover [50, 57, 90, 97]. These high-coverage approaches, even with state-of-the-art state generation optimizations, cannot scale to real-world systems, and are thus only usable to developers who want to test extremely small PM applications or embedded libraries [57].

Fixing PM Bugs The challenge of automatically fixing bugs, while practically unexplored specifically for modern PM systems, has been explored in many prior works for repairing commodity software systems [92, 93, 139] and techniques for general automated program repair are increasing being deployed to repair bugs in production systems [60, 109]. These tools often leverage unsound approaches to repair bugs, which could result in patches that do not fix the original bug or even introduce new bugs; due to the challenges of finding and manually repairing PM bugs, such approaches could lead to the creation of subtle errors that are even more difficult for developers to uncover or reason about. A subset of automated program repair tools target narrower classes of bugs and leverage domain-specific insights about the nature of those bugs to provide stronger guarantees about the correctness of the fixes that they provide [67, 80]; however, these approaches still leave something to be desired, as they provide no sound or formal guarantees about the correctness of their fixes.

1.3 Thesis Statement and Contributions

In this dissertation, we introduce new techniques for automating the detection and correction of PM bugs in modern PM systems. Our overarching insight and thesis statement is that **we can efficiently and thoroughly automate bug detection and correction by approximating and automating the reasoning performed by developers as they write, test, and repair their applications**. Our insight is drawn from observations about how developers organize their applications, test their applications, and fix bugs in their applications. Developers often test applications by identifying and testing execution paths which are the most prone to encountering the targeted type of bug; ergo, replicating this approach to set priorities during automated path exploration leads to accelerated discovery of PM bugs (Chapter 2). Developers then often endeavor to fix bugs by trying to find the most efficient fix that does not compromise the correctness of the fix, which inspires our approach towards correctly fixing PM bugs with heuristically-optimized fixes (Chapter 3). Finally, developers often implicitly organize their application in a way that reveals the crash-consistency requirements of the program (e.g., fields that must be consistent before a crash are grouped into the same data structure). So, rather than trying to build testing approaches that generalize crash-consistency requirements from other sample applications in order to scale testing [31, 41, 50–52, 56, 103, 104, 126], we instead focus on automatically uncovering the semantics of the application-under-test to prune unnecessary testing. Overall, our approach results in bug discovery that is simultaneously more scalable and more accurate than prior work (Chapter 4).

We explore our insight in the three projects that are detailed in this dissertation, which provide ample evidence of the effectiveness of our approach in building techniques to automatically detect and correct PM bugs. We summarize the contributions of this dissertation below:

AGAMOTTO Writing PM applications that are simultaneously correct and efficient is challenging, resulting in many PM applications that 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 our aim to build a system for that more thoroughly tests PM applications, we first perform and present a detailed study of 63 bugs from popular PM projects. We identify two application-independent patterns of PM misuse (i.e., persistence bugs) which account for the majority of bugs in our study and can be detected automatically. We then present AGAMOTTO, a generic and extensible system for discovering misuse of PM 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 persistence bugs. We use AGAMOTTO to identify 84 new persistence bugs in 5 different PM applications and frameworks while incurring no false positives.

HIPPOCRATES PM-specific testing and debugging tools can help developers find PM durability bugs, however even with such tools, **fixing** durability bugs can be challenging. To understand why, we extend our original study of durability bugs and find that although durability bug fixes seems simple, the actual reasoning behind the fix can be complicated and time-consuming. Overall, the severity of these bugs coupled with the difficultly of developing fixes for them motivates us to consider automated approaches to fixing durability bugs. We therefore develop HIPPOCRATES, a system that automatically fixes durability bugs in PM systems. HIPPOCRATES automatically performs the complex reasoning behind durability bug fixes, relieving developers of time-consuming bug fixes. HIPPOCRATES's fixes are guaranteed to be *safe*, as they are guaranteed to not introduce new bugs ("do no harm"). We use HIPPOCRATES to automatically fix 23 durability bugs in real-world and research systems. We show that HIPPOCRATES produces fixes that are functionally equivalent to developer fixes and can generate code that is as efficient or more efficient than developer-written durability code.

SQUINT In this work, we introduce *representative testing*: a new PM crash-state reduction strategy that simultaneously achieves high scalability and high coverage. Our key observation is that many crash-states produced by a PM application can be considered *equivalent* because they evince the same crash-consistency bug, even though the crash-states are not themselves equivalent. We design a heuristic that approximates a small set of representative crash-states, or, a set of crash-states that is equivalent to all of the crash-states that an execution can produce. We build SQUINT, which uses representative testing to perform crash-consistency testing on only the small set of representative crash-states. We demonstrate that SQUINT achieves high coverage, since it finds 108 bugs (53 new) across 19 real-world PM applications, and show that it achieves high scalability, since it scales to real-world PM applications more effectively than existing works.

1.4 Dissertation Outline

The rest of this dissertation is organized as follows. We first focus on how to thoroughly test PM systems for PM persistence bugs and present AGAMOTTO, a symbolic-execution approach to thoroughly testing PM applications for persistence bugs (Chapter 2). We then present HIPPOCRATES, a compiler tool that can take PM durability bug reports from PM-testing tools like AGAMOTTO and automatically fix them using provably-correct fixes (Chapter 3). We then present SQUINT, a PM-testing tool that efficiently detects more general crash-consistency bugs in PM systems, and further discusses what work is remaining in this project (Chapter 4). Finally, we conclude and discuss possible future research directions based on the techniques and insights presented in this dissertation (Chapter 5).

CHAPTER 2

AGAMOTTO: How Persistent is your Persistent Memory Application?

Persistent Memory (PM) is a helpful programming abstraction that developers can use to write applications that directly modify persistent data, 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 chapter 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 durability bugs. AGAMOTTO has so far identified 84 new bugs in 5 different PM applications and frameworks while incurring no false positives.

2.1 Introduction

PM is a promising programming abstraction that offers an appealing performance-cost trade-off for application developers. New NVM technologies, such as Intel Optane DC PMem [69], can offer developers a PM abstraction with latencies that are only 2–3$\times$ higher than the latencies of Dynamic Random-Access Memory (DRAM) [147]. Moreover, such NVM technologies are cheaper than DRAM per gigabyte (GB) of capacity [4]. As byte-addressable memory, NVM can also be accessed via processor load and store instructions. Application developers have already started building systems that use NVM directly, without relying on heavyweight system calls to ensure durability, including ports of popular systems such as memcached [39] and Redis [33].

While using PM directly via persistent data structures can offer good performance, it is challenging to write PM-based applications that are simultaneously correct and efficient [18, 26, 63, 106, 108, 119, 152, 165]. PM platforms often require the use of special *persistence* instructions to enforce the crash-consistency of updates. For example, PM writes in the CPU cache may need to be explicitly flushed to PM using specific instructions or Application Programming Interface (API)s, and those PM flush operations may need to be ordered using memory fences to enforce crash consistency. Incorrect usage of these mechanisms can result in *persistence bugs* which break crash-consistency guarantees or degrade application performance. Persistence 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 [90] and `pmemcheck` [31] specifically to test the crash consistency and durability of PMFS (Persistent Memory File System) [43] and PMDK (Persistent Memory Development Kit) [32], 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 Persistence Inspector [126], PMTest [104], and XFDetector [103] 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 persistence bug finding can be automated (i.e., not require program annotations) 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 [12] to thoroughly explore the state space of a program[1]. In addition to expanding path coverage, symbolic execution also allows AGAMOTTO to detect persistence 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 persistence bugs using two *universal persistence 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 [26, 106, 119, 152, 165]).

To identify application-specific persistence bugs, AGAMOTTO allows developers to provide *custom persistence bug oracles*. To demonstrate the versatility of custom oracles, we implemented two such oracles in AGAMOTTO to detect bugs related to misuse of PMDK's Transaction API [32, 103, 104].

Analyzing large PM applications using traditional symbolic execution [12] 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 persistence bugs in real-world systems including PMDK (a mature PM library) [32], memcached-pm [39], Redis-pmem [33], NVM-Direct [11], and RECIPE [95]. In particular, we found 13 new correctness and 70 new performance bugs using the universal persistence bug oracles, and 1 new correctness bug using a custom durability bug oracle. We report all bugs to their authors, and so far 40 have been confirmed and none denied.

In this chapter we make the following contributions:

- We perform a detailed study of persistence bugs in PMDK as well as bugs found by prior work, and present a new taxonomy of persistence bugs.

[1]AGAMOTTO is named after the "Eye of Agamotto" (i.e., the Time Stone in the Avengers franchise), which has the power to allow the user to explore alternative timelines [156], like how AGAMOTTO allows users to symbolically explore many execution paths for PM persistence bugs.

- We build AGAMOTTO, a PM bug detection tool that can test real-world PM programs using a novel state exploration algorithm. AGAMOTTO automatically detects bugs using two universal persistence 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. AGAMOTTO's artifact is publicly available on GitHub [120].
- We use AGAMOTTO to find 84 new bugs in 5 applications and PM libraries, compared to the 6 persistence bugs found in persistent applications by the state of the art (PMTest [104], which finds 3 bugs, and XFDetector [103], which finds 3 bugs). AGAMOTTO does not incur any false positives in our evaluation.

In the rest of this chapter, we first provide background on PM programming and describe the challenges of PM bug finding (§2.2). We then present the results of our PM bug study and provide common patterns of PM misuse that identify PM bugs (§2.3). Then, we discuss the high-level design of AGAMOTTO and detail the persistence bug detection algorithms and search techniques that power AGAMOTTO's bug detection capabilities (§2.4). Next, we briefly discuss AGAMOTTO's implementation (§2.5) and evaluate the system with respect to both the number of bugs found and the impact of these bugs (§2.6). Finally, we discuss related PM bug detection work (§2.7) and conclude (§2.8).

2.2 Background and Challenges

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

2.2.1 Persistent Memory Programming

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

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

Listing 2.1: **PM Programming Example.** An example of PM programming on the x86 CPU architecture.

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 2.1, which allocates two integers in persistent memory and issues ordered writes to the integers. In order to guarantee that the write to `x` (line 4) is ordered before the write to `y` (line 8), a flush and fence must occur between the updates (line 5 and line 6). To ensure that the write to `y` (line 8) is durable, a flush and fence must occur after the write (line 5 and line 6).

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 [7, 135] with similar semantics to x86 flushes and fences.

2.2.2 Challenges of Detecting Persistent Memory Bugs

PM interfaces for durability and performance are easy to misuse [103, 104] and the resulting *persistence* bugs (i.e., misuses of persistence instructions) can be challenging to detect. Persistence bugs exhibit many characteristics that make them difficult to detect. First, finding a persistence 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 persistence bug requires modeling PM state and instrumenting the program for tracking state updates, which is challenging to

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 2.1: **Bug Survey.** The results of our initial PM bug survey conducted on bugs reported in PMDK [32] and on bugs discovered by PMTest [104] and XFDetector [103].

accomplish using traditional debugging tools. Second, in the case of correctness bugs, the root cause and symptoms of a durability bug are often loosely tied together: while the symptoms of a correctness durability 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, persistence 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 durability 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 [32]) to prevent these bugs. As we show in §2.3, many applications use PM libraries incorrectly and even these established libraries themselves may misuse PM.

2.3 Persistent Memory Bug Study and Classification

In this section, we present a study of PM bugs. We construct a corpus of 63 PM bugs from a mature PM library, Intel's PMDK [32], and persistence bugs from PM projects (PMFS [43] and Redis-pmem [33]) that were found by state-of-the-art PM bug detection tools (PMTest [104] and XFDetector [103]). We chose PMDK because it is a mature project with a thorough issue tracker [75] representing a large collection of existing bugs. We use this corpus to identify common patterns of PM bugs.

Table 2.1 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 persistence 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

[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`

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

Listing 2.2: **Missing Flush/Fence Bug Example.** A missing flush/fence correctness bug adapted from PMDK Issue #1103, PR #3907.

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 developers extensively use `pmemcheck` [31] to detect bugs. In the rest of this section, we present examples of these bugs together with more detailed descriptions.

2.3.1 Missing Flush/Fence Pattern

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` [31] 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 2.2. 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 2.2. In our evaluation (§2.6), 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% [42]. There are PM data structures that intentionally include this pattern [106] as a programming simplification. However, in the applications included in our study and evaluation, all instances of the missing flush/fence pattern are persistence bugs.

2.3.2 Extra Flush/Fence Pattern

The other common pattern of persistent memory misuse which we identify in our study is the

```
1  // array is an array of integers in PM with length = size
2
3  // resizes array in-place
4  resize_array(array, new_size);
5
6  // if size >= new_size, no copying occurs
7  for (size_t i = size; i < new_size; i++) {
8      array[i] = 0;
9  }
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 }
15
16 sfence();
```

Listing 2.3: **Extra Flush/Fence Bug Example.** An extra flush/fence performance bug adapted from PMDK Issue #1117, PR #3860.

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 2.3. 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 [26, 119, 152, 165] (e.g., an additional flush and fence can add an average of 250ns (nanoseconds) of latency [105, 154], where the base latency of uncached PM accesses can be as low as 96ns [77]).

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 persistence bugs. However, in the applications in our study and evaluation, all instances of the extra flush/fence pattern are persistence bugs.

2.3.3 Other Bugs

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 2.4 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

```
// store pool's header
/* BUG: header made valid before pool data made valid */
header = ...
clwb(header);
sfence();

pool = ...
clwb(pool);
sfence();
```

Listing 2.4: **Ordering Bug Example.** An example correctness bug adapted from PMDK Issue #14.

an updated value of `header` without an updated value of `pool`.

2.3.4 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 persistence bugs; we hypothesize that this trend will hold for general PM applications. In our evaluation (§2.6), we find that all instances of these patterns are persistence bugs across a variety of PM libraries and applications (i.e., we find no false positives with these patterns).
- The remaining bugs, while less prevalent in our survey, are still potential sources of data inconsistency or application inefficiency. An ideal tool should allow developers to specify application-specific patterns without requiring extensive test cases and significant developer annotations.

2.4 Design

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

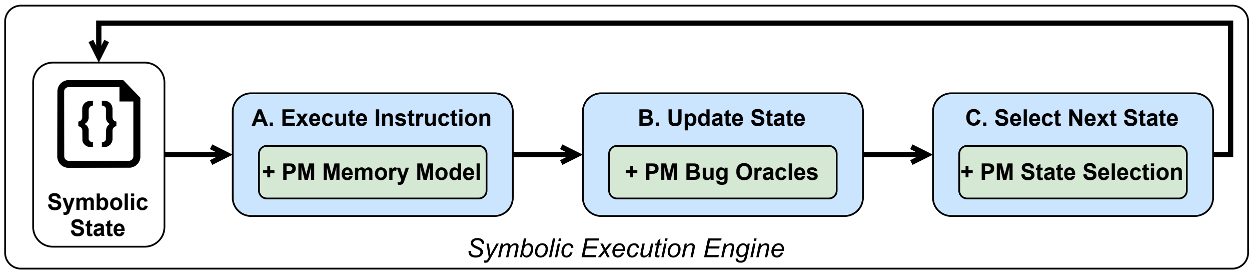


Figure 2.1: **Components of AGAMOTTO.** Green-shaded boxes are AGAMOTTO-specific contributions to the existing symbolic execution engine.

Automation: Bug-finding can take a substantial amount of developer effort [111, 138]; 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, regardless of what PM libraries or APIs that the PM application uses.

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 Figure 2.1 (green-shaded boxes represent the key components unique to AGAMOTTO). AGAMOTTO relies on an existing symbolic execution engine (KLEE [12] in our prototype) to explore the state space of a PM program. During this exploration, AGAMOTTO uses a custom PM memory model (Figure 2.1, Step A) 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 persistence bugs in a PM application. As AGAMOTTO explores the state space of the program, it checks for PM bugs using universal bug oracles (Figure 2.1, Step B), 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 [103, 104].

At the heart of AGAMOTTO lies its PM-aware state space exploration algorithm (Figure 2.1, Step C), 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 persistence bugs (see §2.6).

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.

2.4.1 Persistent Memory Model and Persistent Memory State Tracking

AGAMOTTO facilitates persistence bug detection by tracking the state of persistent memory objects in the program. For each PM allocation, AGAMOTTO tracks constraints on the *durability state* of the allocated cache lines. The durability 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 durability state of PM cache-lines to reflect the behavior of the program. AGAMOTTO uses these constraints to identify execution paths that contain durability bugs (i.e., when redundant flushes are issued, or updates are not properly ordered).

Identifying Persistent Memory 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`) [104]. 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 durability 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.

2.4.2 Persistence Bug Oracles

AGAMOTTO uses the persistent memory state in order to support two types of persistence bug oracles. First, AGAMOTTO provides two built-in *Universal Persistence Bug Oracles*, which check for bugs based on the patterns we identify in our initial study (§2.3). Second, AGAMOTTO allows developers to specify custom, application-specific persistence bug oracles, which we have used to provide two oracles for PMDK's transaction API [32].

2.4.2.1 Universal Persistence Bug Oracles

AGAMOTTO provides two universal persistence 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 2.5. 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 (i.e., `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 that violates the conditions checked by the oracle. In our evaluation (see §2.6), we show that these oracles do not incur any false positives across a variety of PM frameworks and applications.

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

# Extra Flush/Fence Bug Oracles
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(inst.cacheline)
    if inst is fence:
        check_extra_fence(state)
        state.commit_pending()
```

Listing 2.5: **Universal Persistence Bug Oracles.** Pseudo-code for Universal Persistence Bug Oracles and how they are used as AGAMOTTO explores the state space.

2.4.2.2 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 transaction log) or assert that certain 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 that 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 §2.5.

```
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 2.6: **Custom Oracle Example.** A pseudo-code example of a custom oracle, designed to check for redundant PMDK transaction “adds” (i.e., redundant log updates).

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 2.6. PMDK's transaction 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 BTree 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.

2.4.3 Persistent Memory-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 persistence bugs than traditional coverage-guided exploration heuristics.

2.4.3.1 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 [118].

To achieve this, AGAMOTTO first identifies all PM-modifying instructions in the program by leveraging a sound, whole-program (i.e., interprocedural) pointer analysis [5, 21, 61, 62]. 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.

```
char *pbuf = mmap(<PM file>);
...                                // (# of PM-modifying instructions)
do_read = /* user input */;        // (2)
if (do_read) {                     // (0)
    a = pbuf[x];                   // (0)
    foo();                         // (0)
} else {                           // (2)
    a = /* user input */;          // (2)
    pbuf[x] = a;                   // (2)
    clwb(pbuf[x]);                 // (1)
    // BUG: Missing sfence!
}
exit(0);                           // (0)
```

Listing 2.7: **Priority Calculation Example.** 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.

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 §2.6, it accelerates persistence 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`), cache-line 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 2.7, where each instruction is

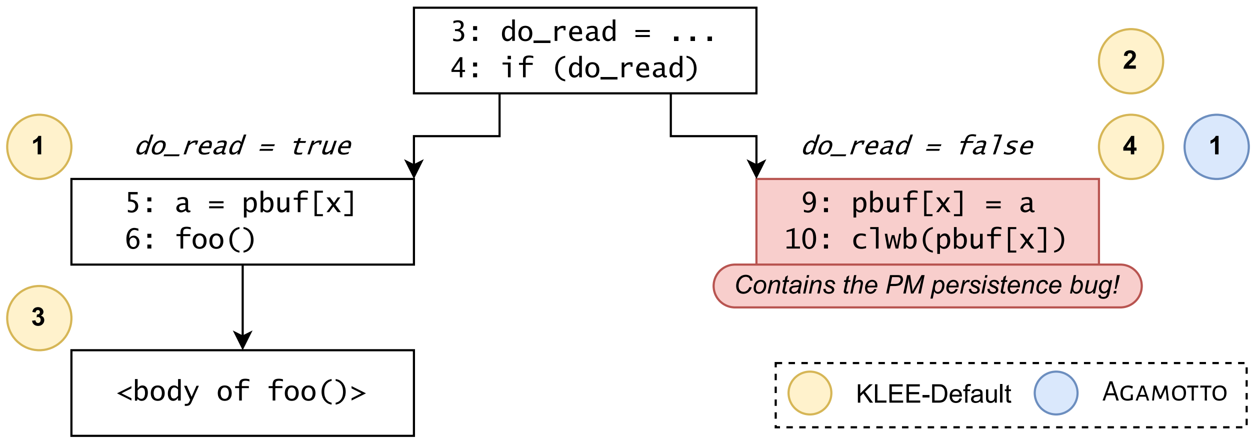


Figure 2.2: **State Space Exploration Comparison.** State space exploration with two strategies: (1) KLEE-Default (based on code coverage), (2) AGAMOTTO's priority-driven exploration. This example corresponds with the bug described in Listing 2.7.

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 back-propagated to the entry point (Line 3).

2.4.3.2 State Exploration Strategy

AGAMOTTO relies on an existing symbolic execution engine, KLEE [12], 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).

Figure 2.2 shows an example of state space exploration for the the example code snippet Listing 2.7, where the state containing `do_read` at the top 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 2.7, 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 §2.6.

2.5 Implementation

AGAMOTTO comprises a persistent memory model (∼400 Lines of Code (LOC) of C++), a static analysis component (∼2600 LOC of C++), and a state space exploration component (∼100 LOC of C++) built atop KLEEE [12]). 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 [64, 86, 128] 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 multi-threading (i.e., POSIX threads) from Cloud9 [24], 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 megabytes (MBs) or GBs of PM, but KLEE is optimized for tracking memory objects that are kilobytes (KBs) in size; treating each PM mapping as a single memory object leads to poor performance when KLEE solves constraints. Instead, AGAMOTTO carefully partitions PM into separate, yet logically adjacent, objects (empirically, we find 16 KB chunks to balance the trade-off between solver time and management overhead). AGAMOTTO also tracks the set of live PM objects to reduce time resolving symbolic addresses for global fence operations.

AGAMOTTO supports custom persistence 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 PM bugs in PMDK and Redis (§2.4.2.2). 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.

2.6 Evaluation

In this section, we evaluate the effectiveness and usefulness of AGAMOTTO. We start by giving an overview of the 84 new bugs AGAMOTTO has found[3] and the insights we gather from them (§2.6.1). We also discuss the positive responses that we have received after reporting bugs to PM application developers (§2.6.2). We then evaluate the performance of AGAMOTTO and how our novel search tactic compares to the default symbolic execution search strategy in KLEE (§2.6.3).

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 [103, 104]. We evaluate AGAMOTTO on two PM libraries. First, we test the PMDK [32] library from Intel, the most active and well-maintained open-source PM project, which has been maintained for over 8 years. Consistent with existing tools [104], we use example data structures provided

[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

with PMDK (e.g., BTree, RBTree and Hashmap implementations) and an application provided by Intel [37] as drivers for our testing. In addition to PMDK, we test NVM-Direct [11], a PM library developed by Oracle. 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 developed 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 [43], an evaluation target that has been considered by prior work [104].

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 §2.5) 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 §2.4.2.2). 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 2.2. Note that none of our experiments use persistent memory hardware since AGAMOTTO symbolically models all interactions with persistent memory.

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 2.2: **Tested Software Versions.** Software configurations we test with AGAMOTTO; note that we tested two different PM versions of Redis-pmem.

	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 2.3: **Bugs Found By AGAMOTTO.** 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**.

2.6.1 Overview

We show a summary of our bug-finding results in Table 2.3[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 persistence 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

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

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.

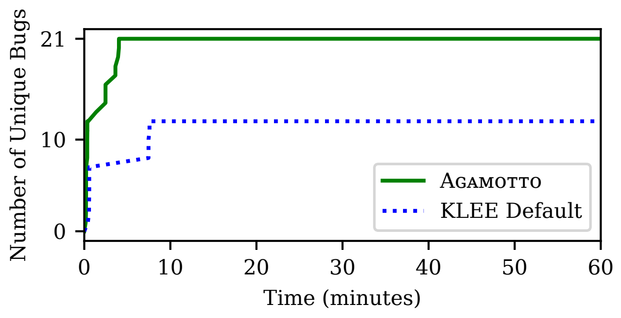
Extra Flush/Fence Bugs We found 22 new bugs using the extra flush/fence bug oracle. Of these bugs, AGAMOTTO found 9 in NVM-Direct, 6 in PMDK library functions and 6 in RECIPE's P-CLHT index.

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 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 BTree 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.

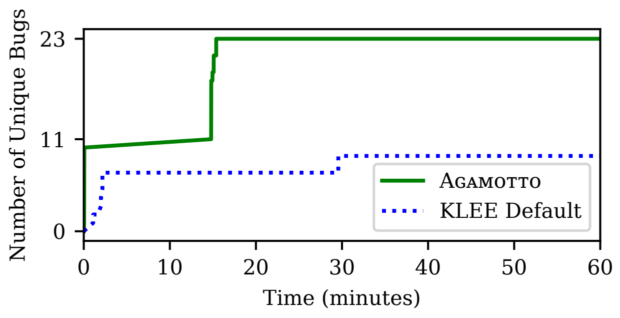
2.6.2 AGAMOTTO Reporting

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."[6] 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 [95]), 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."

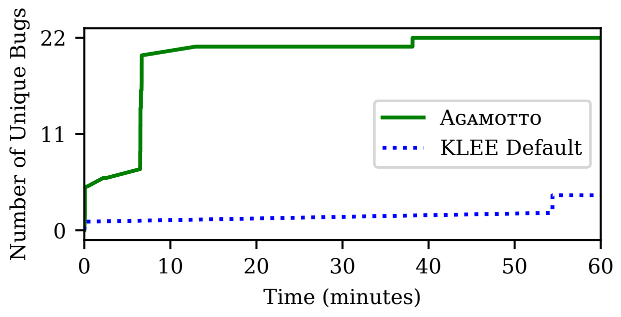
We conclude that AGAMOTTO has been successful in finding bugs that developers care about.



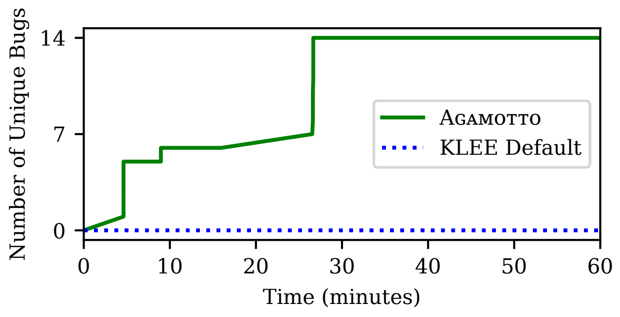
(a) memcached-pm



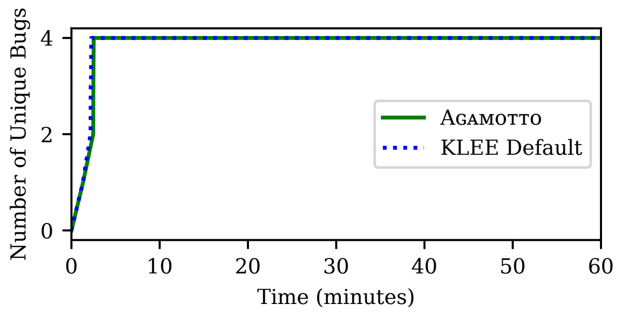
(b) NVM Direct



(c) PMDK



(d) RECIPE



(e) Redis-pmem

Figure 2.3: **Bug Detection Comparison.** Comparison of the KLEE's default search strategy to AGAMOTTO's search strategy.

2.6.3 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 (Figure 2.3a), NVM-Direct (Figure 2.3b), RECIPE's

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

[6]Oracle ultimately archived this project around June 2020.

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 2.4: **Analysis Overhead.** 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.

P-CLHT index (§2.6.3), on PMDK's libpmemobj examples (§2.6.3), and on Redis-pmem (§2.6.3). 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.

Static Analysis Run-time We show the run time of AGAMOTTO's static analysis in Table 2.4. 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.

2.6.4 Case Study: Persistent Memory 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 [128] and a study of memcached-pm found that storing volatile data in PM reduces application performance by roughly 5% [42].

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

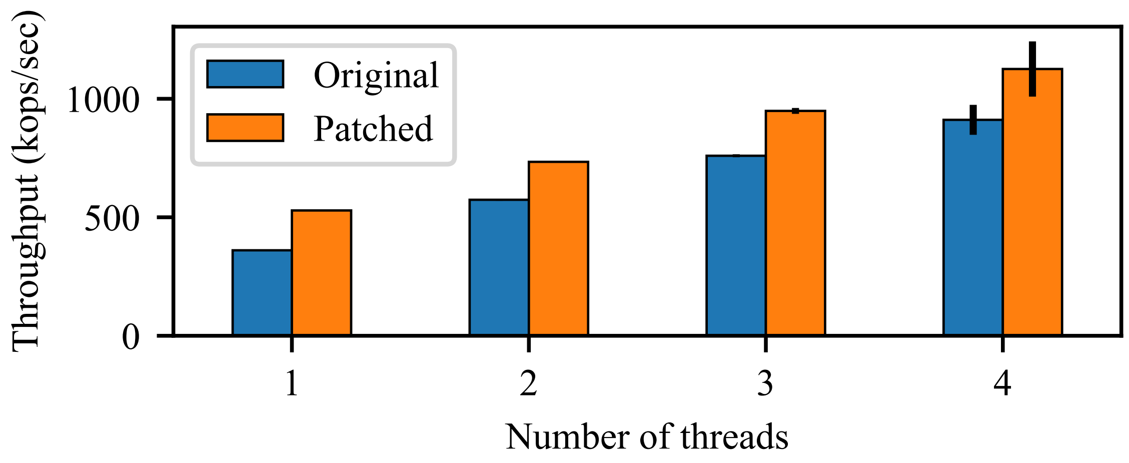


Figure 2.4: **Performance Bug Analysis.** The results of the PM performance bug case study. The write throughput (in kilo-operations per second) of the P-CLHT data structure before and after patching performance bugs. “Original” denotes the unmodified P-CLHT structure and “Patched” denotes P-CLHT after we patch the performance bugs.

	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 2.5: **Prior Work Comparison.** A qualitative comparison between AGAMOTTO and related work, as measured by our design goals (§2.4).

operations, since they stress the update path on which these bugs were found. We report the performance in Figure 2.4. 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.

2.7 Related Work

Persistent Memory Frameworks Crash consistency mechanisms for persistent memory have been considered for years [8, 16, 23, 26, 129]. 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 [32], NV-Heaps [25], LSNVMM [66]), provide language extensions to augment C/C++ with persistent data types (e.g., Mnemosyne [154], NVL-C [40]), or both (e.g., NVM-Direct [11]). Some systems also use transactional hardware mechanisms to provide more efficient updates to persistent memory (NV-HTM [15], Crafty [53]). However, while these mechanisms may make programming easier, they may still contain persistence 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 [30, 43, 88, 153, 159, 160], as many PM-optimized file systems offer full-data consistency, rather than just maintaining metadata consistency [13]. 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 [77].

Tools for Detecting PM Bugs The state-of-the-art tools for detecting PM bugs are PMTest [104] and XFDetector [103]. 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 [104] 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 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 [95] 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` [31] and Persistence Inspector [126], 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 PM bug detection frameworks in Table 2.5.

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 [20, 45, 46, 85, 90, 110, 116].

Many of these tools either test for semantic bugs specific to file systems or are only targeted for block-based storage devices. Yat [90] specifically targets crash consistency testing for Intel's Persistent Memory File System (PMFS [43]). 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 [90].

Bug Taxonomies Many papers taxonomize software bugs in other contexts. In the storage context, JUXTA [115] draws a distinction between shallow (roughly equivalent to application-independent) and semantic (i.e., application-specific) bugs while CrashMonkey [110] studies the effects and number of operations required to induce crash consistency bugs in file systems. More generally, Li et al. [100] and Liu et al. [101] 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, 82, 124] in concurrent programs and semantic crash consistency bugs [95, 103]. Traditional data races result in inconsistent data being read across threads of execution, which many systems have been designed to detect and fix [2, 80, 83, 98, 132, 141]. Principles from data race detection have been adapted to build PM crash consistency mechanisms (i.e., in RECIPE [95]) and PM semantic crash consistency detection tools (i.e., XFDetector [103]). When applied to AGAMOTTO, these principles inform the design of custom bug oracles.

2.8 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 chapter, 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 persistence 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.

CHAPTER 3

HIPPOCRATES: Healing Persistent Memory Bugs Without Doing Any Harm

Programming Persistent Memory (PM) systems is error-prone, as the misuse or omission of the durability mechanisms (i.e., cache-line flushes and memory fences) can lead to durability bugs (i.e., unflushed updates in CPU caches that violate crash consistency). PM-specific testing and debugging tools can help developers find these bugs, however even with such tools, **fixing** durability bugs can be challenging. To determine the reason behind this difficulty, we first study durability bugs and find that although the solution to a durability bug seems simple, the actual reasoning behind the fix can be complicated and time-consuming. Overall, the severity of these bugs coupled with the difficultly of developing fixes for them motivates us to consider automated approaches to fixing durability bugs.

We introduce HIPPOCRATES, a system that automatically fixes durability bugs in PM systems. HIPPOCRATES automatically performs the complex reasoning behind durability bug fixes, relieving developers of time-consuming bug fixes. HIPPOCRATES's fixes are guaranteed to be *safe*, as they are guaranteed to not introduce new bugs ("do no harm"). We use HIPPOCRATES to automatically fix 23 durability bugs in real-world and research systems. We show that HIPPOCRATES produces fixes that are functionally equivalent to developer fixes. We then show that solely using HIPPOCRATES's fixes, we can create a PM port of Redis which has performance rivaling and exceeding the performance of a manually-developed PM-port of Redis.

3.1 Introduction

Non-Volatile Main Memory (NVM) technologies aim to revolutionize the storage-memory hierarchy [119, 129]. NVM technologies such as Intel Optane DC [34, 69] are roughly 8$\times$ less expensive than DRAM [4] and offer disk-like durability with access latencies that are only 2–3$\times$ higher than DRAM latencies [77, 94, 147, 170]. These NVM technologies enable low-latency, high-throughput

Persistent Memory (PM) abstractions that allow applications to access durable storage using the conventional load and store instructions and thus offers persistence without needing heavyweight file-system operations. Since becoming commercially available, popular applications (e.g., memcached [39] and Redis [33]) and companies (e.g., VMware and Oracle [71]) have begun adopting these highly-efficient PM implementations.

Alas, programming PM systems is error-prone [18, 26, 63, 106, 108, 118, 119, 142, 152, 165, 169]. Updates to PM are often cached in volatile CPU caches, and developers must explicitly flush cache lines to guarantee that updates reach PM. Moreover, cache-line flushes are weakly ordered on most architectures (i.e., cache-line flushes do not follow store order), so developers must insert memory fences to order updates as necessary for crash consistency. The misuse of either of these mechanisms results in *durability bugs* and compromises correctness.

To help developers fix durability bugs, a number of useful tools have been built to find such bugs in PM systems [31, 103, 104, 122, 126]. Some tools use developer annotations and existing test suites to find bugs in arbitrary PM programs (e.g., PMTest [104], XFDetector [103], and Persistency Inspector [126]), while others find bugs in specific PM applications and frameworks (e.g., Yat [90] for PMFS [Persistent Memory File System] [43] and `pmemcheck` [31] for applications using PMDK [Persistent Memory Development Kit] [32]). AGAMOTTO [122] (chapter 2) uses symbolic execution to thoroughly discover durability bugs in PM storage systems without the need for developer annotations or test suites.

However, even with effective PM-specific bug finding tools, *fixing* durability bugs in PM systems is challenging. In this chapter, we first analyze 26 bugs reported by Intel's own bug finding tool, `pmemcheck`, and manually fixed by developers. We find that these bugs are arduous to manually debug and fix, even with the help of a state-of-the-art bug finding tool like `pmemcheck`. The PM bugs in our study took on average weeks (23 days) and up to months (66 days) to fix and required numerous attempts (13 commits on average) to correctly fix. We find that these PM bug fixes are complicated due to a trade-off between performance and simplicity. Simple intraprocedural fixes insert a flush or fence in-line with the store that is missing one, making it very easy to reason about the durability of the application. However, if the intraprocedural fix often accesses volatile data (e.g., adding flushes within `memcpy`), the performance of the application may suffer dramatically. Instead, a developer will employ a more complicated interprocedural strategy, in which they add flush operations to other functions in the call stack that result in the missing flush.

One appealing solution is to consider *automated* fixing techniques for PM durability bugs, since PM durability bugs are numerous (e.g., we found 84 new bugs in only 5 PM applications and libraries using AGAMOTTO, see §2.6) and time consuming to fix. Automated bug fixing tools are increasingly being deployed in industry (e.g., at Janus Rehabilitation [60] and Facebook [109]). Existing general purpose program repair tools use heuristics and/or tests suites to modify programs [92, 93, 139]. These tools are best effort, i.e., produced patches may neither fix the bug nor be bug-free, which makes them a poor fit for PM applications. Many of these applications use PM for crash consistency; a buggy patch could lead to irreversible data loss. In contrast, tools which target more specific classes of bugs (e.g., to automate concurrency bug fixing, such as CFix [80]), have been able to provide stronger guarantees.

Our main insight based on our analysis of 26 bugs and their fixes is that PM durability bugs can often be fixed *safely*, meaning the fixes are guaranteed to not incur new correctness bugs (i.e., the fixes "do no harm"). We observe that many durability bug fixes either require adding memory orderings to the program or flushing specific cache lines. We find that durability bugs can be fixed with three kinds of fixes which are guaranteed to not create new bugs: (1) intraprocedural fence insertion; (2) intraprocedural flush insertion; and (3) persistent subprogram creation, which implements interprocedural fixes. Intuitively, these fixes only modify the program by adding memory orderings, which we show cannot violate the original program's memory ordering behavior (§3.4).

Based on our insights, we develop HIPPOCRATES, an automated PM bug fixing tool guaranteed to "do no harm"[1]. HIPPOCRATES uses the output of PM bug finding tools to create safe fixes, thereby fixing durability bugs without introducing new bugs. HIPPOCRATES also uses a safe heuristic that automatically performs the complex reasoning needed to compute an effective location for an interprocedural fix. We show that this heuristic is also guaranteed to "do no harm."

We use HIPPOCRATES to automatically fix all 23 of the durability bugs we find when using `pmemcheck` to test PMDK [32], P-CLHT (from RECIPE [95]), and memcached-pm [39]. We manually verify that HIPPOCRATES is able to correctly fix all the bugs using the bug finding tool that originally found the bugs (`pmemcheck`). For the 11 PMDK bugs we reproduced and fixed, we compare developers' fixes and HIPPOCRATES's automated fixes and find that in most cases (8/11), HIPPOCRATES's fixes are functionally identical to developer fixes. In the remaining cases (3/11), HIPPOCRATES's fixes are functionally equivalent, but the fixes inserted by the PMDK developers are slightly more machine-portable (i.e., PMDK's fix determines which flush instructions are available on the CPU at run-time).

We also show the effectiveness of HIPPOCRATES's interprocedural fix heuristic with a case study of Redis-pmem [33], a developer-created port of Redis designed to use PMDK. We test Redis-pmem against Redis-HIPPOCRATES, a version where all flushes have been inserted by HIPPOCRATES instead of by a developer, and show that Redis-HIPPOCRATES matches or exceeds the performance of Redis-pmem (up to 7% increase in throughput on Yahoo! Cloud Serving Benchmark (YCSB) [28, 29] workloads).

Overall, we make the following contributions:

[1]HIPPOCRATES is named after Hippocrates, the Greek physician who wrote the Hippocratic Oath, a widely known of Greek medical text on the professional ethical standards of physicians [125].

- We provide an analysis of bugs found with a state-of-the-art PM bug finding tool and their associated fixes, which motivates our design of HIPPOCRATES.
- Based on the insights of our analysis of existing durability bug fixes, we develop HIPPOCRATES, a novel automated PM bug fixing tool. HIPPOCRATES uses safe fixes in conjunction with a safe heuristic to safely modify PM programs to eliminate bugs that have been detected by PM bug finding tools.
- We demonstrate that HIPPOCRATES is able to fix all 23 reproduced bugs while not introducing new bugs. HIPPOCRATES also generates fixes which do not incur unnecessary overhead, rivaling and exceeding the performance of manually-developed durability mechanisms.

In the rest of this chapter, we provide background on PM programming and discuss the challenges of automatic PM bug fixing (§3.2). We then discuss our analysis of PMDK bugs and our insights (§3.3). Next, we describe the design of HIPPOCRATES's automated fixes and sketch proofs of their correctness (§3.4) and discuss details of HIPPOCRATES's implementation (§3.5). We then evaluate the effectiveness and performance of HIPPOCRATES (§3.6). Finally, we discuss related topics (§3.7), related prior work (§3.8), and conclude (§3.9).

3.2 Background and Challenges

3.2.1 Persistent Memory Programming

In order to take advantage of byte-addressable PM platforms, developers must modify existing programs to use user-level persistence mechanisms, according to they type of PM being used. These mechanisms are cache-line flushes (or non-temporal stores) and memory fences. The x86 ISA provides 3 cache-line flush instructions (`CLFLUSH`, `CLFLUSHOPT`, and `CLWB`) and 2 memory fence instructions (`MFENCE`, which orders all memory operations including loads; and `SFENCE`, which only orders store-like instructions and cache line flushes) [72, 73]. ARM provides similar instructions with similar semantics (e.g., flush [`DC CVAP`] and fence [`DSB`] [7, 135]). Developers need to ensure that updates destined for PM are flushed from the CPU cache, as the updates are volatile until they leave the CPU cache and reach PM. Furthermore, instructions that flush the CPU cache are generally weakly-ordered (i.e., can be reordered after subsequent memory instructions, with the exception of `CLFLUSH`) with respect to other memory instructions (as are non-temporal stores), so explicit memory-fences must be issued to force the execution of cache-flush instructions at specific points during the program's execution.

Misuse or omission of persistence mechanisms in PM programming can lead to *durability* bugs. A durability bug occurs when an update to PM is not made properly durable, i.e., the update is not flushed from the volatile CPU cache or an update is not properly ordered. Durability bugs in PM can be briefly classified as due to: a lack of a cache-line flush instruction (a "missing-flush bug"), a lack of a memory fence (a "missing-fence bug"), or both (a "missing-flush&fence bug"). When any of these bugs are present in a program, a crash that occurs may cause updates to be missing or partially applied, causing data inconsistencies.

In this work, we specifically examine durability bugs. While there are other classes of PM bugs (e.g., performance bugs resulting from overuse of cache flush instructions), we have demonstrated that PM durability bugs are numerous (see §2.3, §2.6) and existing PM bug finding tools target durability bugs (e.g., `pmemcheck`, Yat, PMTest, and AGAMOTTO). Moreover, durability bugs are the only class of bugs that many existing tools can detect automatically (i.e., without source annotations). We defer further discussion of other bug types to §3.7.

3.2.2 Existing Approaches for Finding Durability Bugs

The challenges and severity of durability bugs in PM systems have spurred many works in automatic detection of PM bugs. PM durability bugs are difficult to detect as they are only observable after a failure has occurred and data has been rendered inconsistent. This is because the durability of updates to PM relies on the state of the CPU cache (i.e., whether or not cache lines have been flushed), which is not directly observable in current CPU architectures. This means that bug-finding tools for PM have to rely on some other mechanism in order to detect bugs (e.g., trace validation [104], binary instrumentation [31, 103, 126], or a symbolic memory model [122]; see §3.8 for further discussion).

One of the advantages of these PM testing tools is that they are all capable of generating a trace of all PM operations that occur over the execution of the application under test, along with the actual list of detected errors encountered during execution. This information is an important feature that allows us to consider automated PM bug-fixing solutions.

3.2.3 Challenges of Automating Fixing Persistent Memory Bugs

One compelling technique for alleviating the challenges of building and debugging PM applications is automated bug fixing. Automated bug fixing tools are becoming an increasingly popular approach, including in industry [60, 109]. Many of these systems are targeted at general-purpose program repair and aim to fix any class of bugs by using heuristics [92, 93, 139] (see §3.8). However, these approaches are not ideal for solving durability bugs as the fixes produced by these approaches may neither fix the bug nor be bug-free. This is problematic for PM applications, as unsafe fixes may result in crash-consistency violations that cause irreversible data loss.

Issue Numbers	Average Commits	Average Days From Open to Close	Max Days From Open to Close	Bug Type
440, 441, 444	-	-	-	Core library/tool bug
442, 446, 447, 448, 449, 450, 452, 458, 459, 460, 461, 463, 465, 466	17	33	66	Core library/tool bug
940, 942, 943, 945	-	-	-	API misuse
545, 585, 949, 1103, 1118	2	15	38	API misuse
Average	13	28	66	

Table 3.1: **Study of PM Bug Fixes.** An overview of the 26 PMDK bugs and their fixes we analyze. The first 17 are bugs with root causes within PMDK library code. The remaining 9 bugs are caused by API misuse within PMDK's unit tests.

In contrast to general bug-fixing approaches, tools which target more specific classes of bugs have been able to provide stronger guarantees. For example, CFix [80] and AFix [79] (which target concurrency bugs, see §3.8) are both able to generate fixes which either do not create new bugs, or, reduce the likelihood of creating bugs. While the guarantees provided by AFix and CFix are not formal, they employ a more principled and rigorous testing approach which inspires this work.

3.3 Study of Durability Bugs and Fixes

We want to investigate how PM bugs are fixed in order to consider methods for safely fixing these bugs automatically. To this end, we first study the difficulty of fixing real bugs in PM systems (§3.3.1), which motivates our desire to create an automated bug fixing solution. We then study the fixes for these bugs (§3.3.2), which provide insights on how we can go about automatically fixing these kinds of bugs. We then present our overall conclusions and insights from this study (§3.3.3).

Study Targets In this section, we present a study of durability bugs and their associated fixes in Intel's PMDK (Persistent Memory Development Kit), which is a mature collection of libraries and tools for accessing Intel PM devices used in real-world systems such as Redis-pmem [33] and memcached-pm [39]. We study all the 26 bugs in PMDK that were found using PMDK's bug detection tool, `pmemcheck` [31], and subsequently fixed. We chose these bugs because they are well-documented and validated in PMDK's issue tracker [70, 75], and `pmemcheck` provides rich information about detected bugs in the form of execution traces[2].

3.3.1 Study of Bugs

We present an overview of our bug study in Table 3.1. Of the 26 bugs we study, 17 have their root cause within the core PMDK libraries (e.g., libpmemobj) or core PMDK tools (e.g., utilities for managing object pools). These bugs are particularly severe as they could lead to data corruption for any application built on top of PMDK's persistent object API. The remaining 9 bugs are caused by the misuse of PMDK's API. These API misuse bugs further demonstrate the difficulty of PM programming.

Moreover, the bugs we study seem to have been arduous to debug and fully fix. In particular, these bugs take a long time to reproduce using `pmemcheck` (as stated by one bug reporter and as confirmed by us in our own testing), which hampers the development and validation of a fix that fixes the bug. We also observe that these bugs were not fixed quickly; each bug required an average of 13 commits to create a passing build, taking an average of 23 days (up to 66) to close the issue.

3.3.2 Study of Bug Fixes

To better understand how developers fix durability bugs, we further study fixes implemented by PMDK developers for the bugs we analyze. To that end, we study the associated commits for all of the 26 bugs. We find that although the verbal descriptions of the fixes are all very similar (e.g., "add missing persists"), the actual fixes of each bug vary in their implementation. We broadly classify these fixes into two categories: *intraprocedural* fixes, which insert flushes and fences in-line with stores to PM, and *interprocedural* fixes, which insert flushes and fences in a separate function context. We provide real examples and describe the difference between these two fix categories below and then discuss why they are challenging to implement.

Intraprocedural fixes We provide an example of an intraprocedural fix in Listing 3.1. These durability fixes are where persistence mechanisms are inserted within the same function (intraprocedurally) as the memory-modifying instruction. In Listing 3.1, the original modification on Line 2 was made durable by inserting a flush and fence immediately after the update to `oid`, rather than in a different function.

Interprocedural fixes We provide an example of an interprocedural fix in Listing 3.2. These durability fixes are where persistence mechanisms are inserted outside of the function context

[2]Providing this information from AGAMOTTO is also possible, but would require more extensive modifications to KLEE (§2.5).

```
if (if_free != 0) {
    *oid = NULL; // oid is a pointer to PM
    // FIX: insert missing flush and fence
    CLWB(oid);
    SFENCE;
}
```

Listing 3.1: **Intraprocedural Fix Example.** An example missing-flush&fence bug with an intraprocedural fix, adapted from PMDK Issue #1103.

```
if (/* condition */) {
    memcpy(pmem_addr, vol_src, nbytes);
    // FIX: insert missing flush and fence
    pmem_persist(pmem_addr, nbytes);
}
```

Listing 3.2: **Interprocedural Fix Example.** An example missing-flush&fence bug with an interprocedural fix, adapted from PMDK Issue #463.

(interprocedurally) of the memory-modifying instruction(s). In Listing 3.2, the original modifications occur within the `memcpy` function, but the fix (`pmem_persist`, which flushes and fences all cache lines in the address range) is deferred until `memcpy` returns. Interprocedural fixes are also employed for non-library functions as well (e.g., an internal checksum function) and can occur multiple frames above the original PM modification (i.e., the original PM update occurs in many nested function calls below the fix).

Challenges of inserting fixes While the scope of these modifications can be small, it is challenging for developers to ensure their fixes simultaneously achieve crash-consistency and good performance. Specifically, the reasoning behind determining whether a fix should be intraprocedural or interprocedural is challenging as this has serious implications on performance (see §3.6.3). For example, inserting intraprocedural fixes into `memcpy` would make reasoning about durability easier, but would incur performance penalties for invocations of `memcpy` on volatile data (residing in DRAM) as well as limiting memory parallelism with the increased number of memory fences. An interprocedural fix (such as in Listing 3.2) can be more efficient, but can be trickier to place correctly in the program such that crash-consistency requirements are not violated (i.e., ensuring an interprocedural fix occurs before an operation which may cause system shutdown). These trade-offs and technical challenges explain why fixing durability bugs is difficult, even though the fixes themselves can be very small. This is also an important trade-off in practice, as over half (16/26, 62%) of the bugs in our study were fixed with interprocedural fixes.

3.3.3 Key Insights

Our primary insight that drives our design is that for PM bugs, we can create *safe* fixes (i.e., the fixes do not introduce new bugs) which are best-effort with respect to performance, rather than making fixes which are best-effort with respect to correctness (like general-purpose automatic bug-fixing tools we discuss in §3.2.3). All PM durability bugs can be fixed using only intraprocedural fixes, which are easy to reason about automatically because they are made of relatively simple operations (i.e., flush and fence insertion). Interprocedural fixes can then be used as a means for improving performance; when they cannot be safely employed automatically, a safe intraprocedural fix can be used instead.

3.4 Algorithms and Design of HIPPOCRATES

Based on our insights, we design HIPPOCRATES, an automated bug fixing tool targeted at safely fixing PM durability bugs. HIPPOCRATES strives to achieve these four design principles:

Ease of use: An automated bug fixing tool should require little developer effort to use. To this end, HIPPOCRATES does not require any input from the developer other than the output of an automated PM bug finding tool.

Do no harm: An automated bug fixing tool should not introduce any new bugs which may impact program correctness. To this end, HIPPOCRATES only introduces bug fixes that are guaranteed to not introduce new bugs.

Performance of fixes: An automated bug fixing tool should strive to achieve best-effort fixes with regard to performance. To this end, HIPPOCRATES employs heuristics that strive to place fixes in optimal locations while provably not impacting the correctness of the inserted fixes.

Offline overhead: Additionally, an automated bug fixing tool should complete its operations in a reasonable amount of time so that it can be used as part of the development cycle for maintaining systems. Existing automated bug fixing solutions are able to produce fixes overnight.

3.4.1 Overview

The system overview of HIPPOCRATES is shown in Figure 3.1. HIPPOCRATES expects a PM-specific execution trace where each event in the trace includes the source line where the event

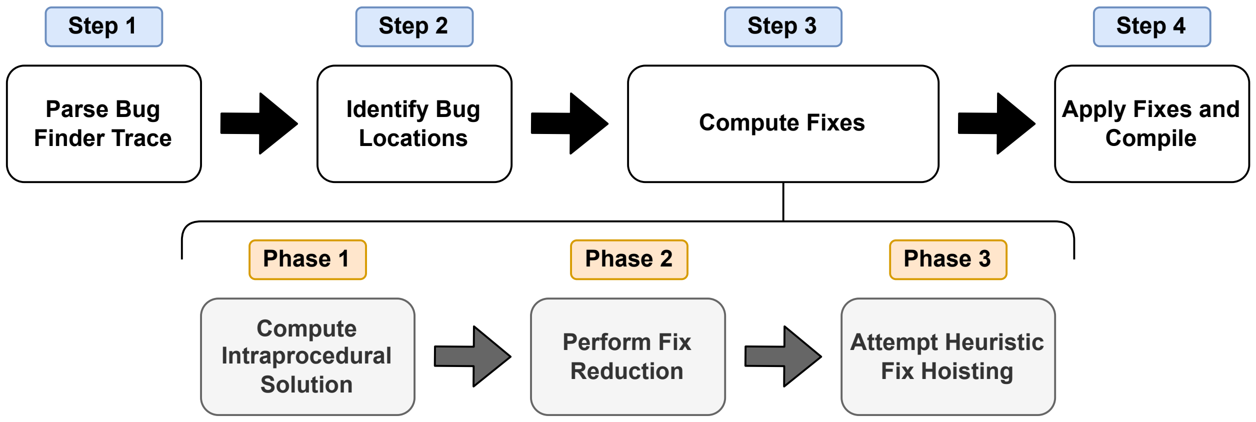


Figure 3.1: **HIPPOCRATES System Overview.**

occurred, the stack trace at the time of the event, and PM-specific information (e.g., the size and location of PM being modified or flushed, or that the instruction is a memory fence, or that a bug has been detected). Many PM-specific tools are capable of generating this information; `pmemcheck` provides a trace with this information by default, and other tools like PMTest and AGAMOTTO can be easily modified to provide the same level of information. This trace is then given to HIPPOCRATES (Step 1) in combination with the application under test so HIPPOCRATES can fix bugs. HIPPOCRATES then uses the trace to locate the original operation which caused each bug detected by the bug finder (e.g., the unflushed store which causes a missing flush bug) (Step 2). HIPPOCRATES then computes all required fixes (Step 3), applies the fixes, and compiles the modified application (Step 4).

HIPPOCRATES goes through a three-phase process to compute fixes (Step 3): first, it computes the simplest possible fix using only intraprocedural fixes; second, HIPPOCRATES performs "fix reduction," where fixes that would create a redundant flush or fence are merged; and third, it performs a heuristic transformation to determine if fixes should be "hoisted," i.e., if any intraprocedural fixes (i.e., fixes in-line with the PM modification) can and should be converted into an interprocedural fix (i.e., in a caller function).

We now discuss the generation of fixes in HIPPOCRATES and provide proof sketches for their correctness.

3.4.2 HIPPOCRATES's Bug Fixes and Proof Sketches

Based on the analysis of our bug study, we identify three code transformations to fix a broad range of durability bugs: (1) the intraprocedural insertion of memory fence instructions, used to fix missing fence bugs; (2) the intraprocedural insertion of CPU cache flush instructions, used to fix missing flush bugs; and (3) interprocedural durability fixes, used to fix missing flush and missing fence bugs when intraprocedural fixes would result in poor performance. These transformations are composable, e.g., a missing-flush&fence bug can be fixed by applying both an intraprocedural flush fix and an intraprocedural fence fix. We first discuss each one of these fixes and sketches of their correctness proofs, before discussing how HIPPOCRATES selects which kind of fix to apply (§3.4.3).

Notation To present our proof sketches, we use a notation similar to prior work [89]. All symbols indicate individual memory instructions or atomic units of memory instructions, i.e., X and Y are separate instructions or blocks of atomic memory instructions (e.g., an Intel TSX transaction [74] is treated as a single atomic unit). We denote an update to PM as X (i.e., a store to X), a flush to PM as $F(X)$ (i.e., a cache-line flush which flushes X), a fence instruction as M, and any other instruction as I. The notation $X \rightarrow Y$ denotes "X happens-before Y." Similarly, $X \not\rightarrow Y$ denotes "X does not happen-before Y." The happens-before relationship is transitive [89]. For all instructions, if X is executed before Y in a given thread, $X \rightarrow Y$.

Definitions We define flushes and fences based on the semantics of flush and fence instructions implemented in current CPU architectures [7, 72, 73], as proposed in previous work [76].

A *cache-line flush* (or just *flush*) $F(X)$ is an instruction which writes update X to PM at some point in time after $F(X)$ is executed, potentially evicting X from the cache hierarchy.

A *memory fence* (or just *fence*) M is an instruction which performs two actions: (1) it causes all memory updates in M's thread of execution to become visible across all threads in a shared memory system (i.e., for all updates W such that $W \rightarrow M$ and all readers R which execute on any thread after the point in time when M is executed, R will read W); and (2) for all instructions I and updates X, such that there exists a flush operation, $F(X)$, with $X \rightarrow F(X) \rightarrow M \rightarrow I$, M causes X to be written to PM before I (i.e., M creates a durability ordering, see below).

An update X to PM has an associated *durability event* X_D. X_D is ordered before another instruction I if and only if X is flushed and fenced before I, formally, $X_D \rightarrow I \iff$ there exists a flush $F(X)$ and fence M such that $X \rightarrow F(X) \rightarrow M \rightarrow I$. We define the ordering of $X_D \rightarrow I$ to be a *durability ordering*. Informally, if $X_D \rightarrow I$, that means that X is durable before I.

We define a *bug* as the *possibility* of incorrect program behavior. For our use case, which does not consider real-time constraints, incorrect behavior is limited to generating incorrect outputs. A bug is *new*, if and only if the *possibility of new incorrect behavior* is introduced into the program.

We define a fix as *safe*, if it can be inserted into a program without incurring any *new* bugs. As cache-line flush and memory fence instructions do not modify the program state (i.e., the values contained in registers or memory), the safety of PM fixes only requires reasoning about modifications to the program's memory ordering and durability behavior.

```
void foo(char *pm_addr) {
    pm_addr[0] = ...
    CLWB(pm_addr);
    // FIX: insert a memory fence
    SFENCE();
    // Without the fence, the system may lose data
    ↯ Crash occurs here ↯
}
```

Listing 3.3: **Missing-Fence Bug Fix Example.** An example missing-fence bug that is fixed by the intraprocedural insertion of a memory fence (i.e., an `SFENCE` instruction).

3.4.2.1 Intraprocedural Memory Fence Insertion

We show an example of a missing-fence bug fixed by an intraprocedural memory fence insertion in Listing 3.3. Without the `SFENCE` instruction inserted on §3.4.2.1, the `CLWB` instruction would not be ordered before the system crashed, potentially leading to data loss or data inconsistencies (see §3.2.1). Inserting a fix for this kind of bug can always be done safely; we provide a proof sketch below.

Definition Formally, a bug $B(X)_{\text{fence}}$, indicating a missing memory fence, occurs when a program requires $X_D \rightarrow I$ for durability, but there does not exist a fence, M, such that $X \rightarrow F(X) \rightarrow M \rightarrow I$.

Lemma 1 *It is safe to insert a memory fence M into a program.* We prove this by contradiction. Assume that inserting a fence M causes a new bug in a program. By definition, M has two actions: (1) it causes all memory updates in M's thread of execution to become visible across all threads in a shared memory system; and (2) for all instructions I, updates X, such that there exists a flush operation, $F(X)$, with $X \rightarrow F(X) \rightarrow M \rightarrow I$, M causes X to be written to PM before I. The new bug must be caused by one of these two actions, which we handle below:

(1) In this case, the bug must be caused by the updates in M's thread of execution becoming visible across all threads in a shared-memory system after the execution of M. Formally, the bug is a result of a memory update on M's thread of execution (W) and a memory read on a different thread (R), such that $W \rightarrow M$, R occurs after M and R observes W (i.e., R reads W). In an execution without M, R may still observe W. For example, after executing W, but before executing R, enough time passes (e.g., due to the execution of other instructions) such that W becomes visible. Thus, M does not introduce the possibility of R observing W, so the bug cannot be caused by memory updates becoming usable across threads.

(2) In this case, the bug is caused by a new durability ordering. Formally, the bug is caused by an instruction, I, and an update X such that $X \rightarrow F(X) \rightarrow M \rightarrow I$. In an execution without M,

```
void foo(char *pm_addr) {
    pm_addr[0] = ...
    // FIX: insert a flush
    CLWB(pm_addr);
    // Without the flush, the system may lose data
    SFENCE();
    ↯ Crash occurs here ↯
}
```

Listing 3.4: **Missing-Flush Bug Fix Example.** An example of a missing-flush that is fixed by a intraprocedural cache-line flush insertion.

X_D may still occur before I due to cache evictions. Therefore, inserting M does not introduce the possibility of $X_D \rightarrow I$, so this is not the cause of the bug.

Thus, the bug cannot be caused by (1) or (2), so M cannot cause the new bug, which is a contradiction. ↯

Theorem 1 *If $B(X)_{fence}$ exists and M is a memory fence inserted into the program such that $X \rightarrow F(X) \rightarrow M \rightarrow I$, then the insertion of M safely fixes $B(X)_{fence}$.*

M fixes $B(X)_{\text{fence}}$ by definition and is safe to insert by Lemma 1. Therefore, inserting M safely fixes $B(X)_{\text{fence}}$. ■

3.4.2.2 Intraprocedural Cache-Line Flush Insertion

Listing 3.4 shows an intraprocedural cache-line flush fix, in which a `CLWB` is inserted (§3.4.2.2) to write the modification of `pm_addr[0]` to PM.

Definition Formally, a bug $B(X)_{\text{flush}}$, indicating a missing (cache-line) flush, occurs when a program requires $X_D \rightarrow I$ for crash-consistency, but there does not exist a flush, $F(X)$, such that $X \rightarrow F(X) \rightarrow M \rightarrow I$.

Lemma 2 *It is safe to insert a flush $F(X)$ into a program.* We prove this by contradiction. Assume that inserting flush $F(X)$ causes a new bug in a program. By definition, $F(X)$ only performs one action: $F(X)$ writes update X to PM at some point in time after $F(X)$ is executed, potentially evicting X from the cache hierarchy, so the new bug must be caused by X either being written to PM or evicted from the cache hierarchy after $F(X)$. However, without executing $F(X)$, X may still be evicted from the cache, and thus also written to PM, due to memory pressure. Thus, $F(X)$ does not introduce the possibility of X getting written to PM or being evicted from the cache, so $F(X)$ does not cause the bug. This is a contradiction. ↯

Theorem 2 *If $B(X)_{\textit{flush}}$ exists and $F(X)$ is a flush inserted into the program such that $X \rightarrow F(X) \rightarrow M \rightarrow I$, then the insertion of $F(X)$ safely fixes $B(X)_{\textit{flush}}$.*

$F(X)$ fixes $B(X)_{\text{flush}}$ by definition and is safe to insert by Lemma 2, so $F(X)$ safely fixes $B(X)_{\text{flush}}$. ■

3.4.2.3 Intraprocedural Flush and Fence Insertion

Listing 3.1 shows an example of a missing-flush&fence bug. These bugs are a composition of the two earlier classes (i.e., a missing-flush bug and a missing-fence bug); we show that they can be safely fixed by applying both intraprocedural fix techniques.

Definition Formally, a bug $B(X)_{\text{flush\&fence}}$, indicating a missing flush and fence, occurs when a program has both $B(X)_{\text{flush}}$ and $B(X)_{\text{fence}}$ bugs, i.e., the program requires $X_D \rightarrow I$ for crash-consistency, but there does not exist a flush $F(X)$ nor a fence M, such that $X \rightarrow F(X) \rightarrow M \rightarrow I$.

Theorem 3 *If $B(X)_{\textit{flush\&fence}}$ exists and $F(X)$ is a flush and M is a fence that are both inserted into the program such that $X \rightarrow F(X) \rightarrow M \rightarrow I$, then the insertion of $F(X)$ and M safely fixes $B(X)_{\textit{flush\&fence}}$.*

Inserting $F(X)$ and M such that $X \rightarrow F(X) \rightarrow M \rightarrow I$, fixes $B(X)_{\text{flush\&fence}}$ by definition and is safe by Lemma 1 and Lemma 2. Therefore, inserting $F(X)$ and M such that $F(X) \rightarrow M$ safely fixes $B(X)_{\text{flush\&fence}}$. ■

3.4.2.4 Interprocedural Fixes

Intraprocedural fixes are often expensive. Consider Listing 3.5, a program in which all PM updates (i.e., the write made by `update` through the call to `modify`) must be durable before Line 30. Fixing this bug intraprocedurally (in `update(...)`) is tempting, but leads to performance issues since `update` frequently operates on volatile memory. Specifically, `modify(vol_addr)` on Line 22 results in a call to `update(vol_addr, ..., ...)`, which modifies volatile memory. So, adding a `CLWB` and `SFENCE` directly in `update(...)` will lead to durability mechanisms being unnecessarily used on non-PM regions of memory. Instead, an interprocedural fix (i.e., outside of `update`) is desirable. Yet, generating an interprocedural fix can be challenging; for example, an interprocedural fix that modifies `foo` must determine the PM updates made by `modify`, which depends on the semantics of `modify` (e.g., local variables used to calculate PM addresses, etc.). For example, a correct interprocedural fix must identify the value of `addr[idx]` in Line 2 in order to flush all modified cachelines, but the value of `idx` passed to `update` from `modify` (Line 6) may depend upon user input and be challenging or even impossible to calculate statically. In practice,

```
 1 void update(char *addr, int idx, char val) {
 2     addr[idx] = val;
 3 }
 4
 5 void modify(char *addr) {
 6     update(addr, ..., ...);
 7 }
 8
 9 // New function generated by HIPPOCRATES
10 void update_PM(char *addr, int idx, char val) {
11     addr[idx] = val;
12     CLWB(&addr[idx]);
13 }
14
15 // New function generated by HIPPOCRATES
16 void modify_PM(char *addr) {
17     update_PM(addr, ..., ...);
18 }
19
20 void foo(char *vol_addr, char *pm_addr) {
21     for (int i = 0; i < INT32_MAX; i++) {
22         modify(vol_addr);
23     }
24
25     ~~modify(pm_addr);~~
26     // The above call is replaced with:
27     modify_PM(pm_addr);
28     SFENCE();
29
30     ↯ Crash occurs here ↯
31 }
```

Listing 3.5: **Persistent Subprogram Transformation Example.** An example interprocedural fix as implemented by HIPPOCRATES as a *persistent subprogram transformation*. The functions labeled "new" (and highlighted) are generated by HIPPOCRATES during the persistent subprogram transformation. Line 25 is replaced with Line 27 during the transformation.

developers use their own semantic knowledge of their software to bridge this gap. However, applying the same approach to HIPPOCRATES breaks the ease-of-use design principle since it would require substantial input from developers.

To obtain the performance benefits of interprocedural fixes without requiring developer annotations, HIPPOCRATES introduces the *persistent subprogram transformation*. This operation reuses the semantic information which already exists in the *subprogram* (defined as a function and all nested functions called by it) to identify which modifications need to be made durable. A persistent subprogram transformation duplicates a subprogram, inserts flushes after every store that modifies persistent memory, and places a single memory fence after the call site to the modified subprogram. The resulting *persistent* subprogram guarantees that all the PM modifications are flushed while minimizing the number of memory fences. Furthermore, since the flushes are based on the subprogram's original semantics, the persistent subprogram only flushes cache lines that are modified.

For example, `modify_PM` (Line 16) is the persistent subprogram of `modify`. The subprogram creates and calls `update_PM`, a copy of `update` in which all PM modifications are immediately flushed (Line 12). In addition, a fence is added to the end of `modify_PM` so that updates becomes durable. By copying the subprogram, `modify_PM()` reuses the semantics of `modify` (e.g., local variables used to calculate PM addresses, etc.) to ensure that all modifications are durable.

HIPPOCRATES reuses subsets of a persistent subprogram to reduce the impact of persistent subprogram transformation on code size. For example, consider if `update` was also called in a function, `permute` (not shown). If HIPPOCRATES performs a persistent subprogram transformation on `permute`, the resulting persistent subprogram (`permute_PM`) would need to be modified to call a persistent version of `update` (`update_PM`). Since a persistent version of `update` was created in an earlier persistent subprogram transformation (i.e., when `modify_PM` was created), HIPPOCRATES modifies `permute_PM` so that it directly calls the existing `update_PM` rather than creating another persistent version of `update` for `permute_PM` to call (e.g., HIPPOCRATES does not have to create `update_PM_2`). In our testing (§3.6.3), we find that the overall code size increase is negligible (only 0.05% increase in the end binary on average).

HIPPOCRATES attempts persistent subprogram transformations for durability bugs that require $X_D \rightarrow I$ where X (the modification to PM) is in a separate function context from I (the instruction by which X must be durable). For example, in Listing 3.5, all modifications must be made durable before §3.4.2.4 (i.e., I is the victim of a system crash). HIPPOCRATES uses a heuristic (described below) to determine which function in the call stack should be the start of the persistent subprogram. HIPPOCRATES considers functions on the call stack between the function containing X (in this case, `update`) and the function on the call stack called by the function containing I (in this case, the function being called by `foo` is `modify`) to be candidates for the start of the persistent subprogram. HIPPOCRATES does not select the function containing I (`foo`) nor functions which call the function containing I (any callers of `foo`) because a separate intraprocedural fence M would need to be inserted before I (such that $X \rightarrow F(X) \rightarrow M \rightarrow I$ would still hold), which would limit the performance benefits of the transformation. In this case, if `foo` were the start of the persistent subprogram, a fence would be needed before the crash and at the end of `foo`, which adds more fences than is needed and is undesirable for performance. HIPPOCRATES uses the bug finder trace (see §3.4.1) to identify I.

We now demonstrate the safety of this transformation.

Theorem 4 *If $B(X)_Q$ is a bug where $Q \in \{fence, flush, flush\&fence\}$ indicating the program requires $X_D \rightarrow I$, for some instruction I outside of the function containing X, $F(X)$ and M are flush and fence operations inserted into the subprogram such that $X \rightarrow F(X) \rightarrow M \rightarrow I$, then a persistent subprogram transformation safely fixes $B(X)_Q$.*

By duplicating the function and replacing the call site with a call to the duplicated function, the memory ordering behavior, durability orderings, and all other semantics are unaltered, rendering the initial duplication safe. Inserting fence M at the end of the duplicated function and flush $F(X)$ after X are both safe (by Lemma 1 and Lemma 2). Furthermore, these both fix $B(X)_Q$ by the definition of the bug: for $Q =$ fence, M fixes $B(X)_Q$; for $Q =$ flush, $F(X)$ fixes $B(X)_Q$; and for $Q =$ flush&fence, $F(X)$ and M such that $F(X) \rightarrow M$ fix $B(X)_Q$. Therefore, the persistent subprogram transformation safely fixes $B(X)_Q$. ■

3.4.3 Optimization of HIPPOCRATES's Fixes

After HIPPOCRATES determines all bug locations and inserts intraprocedural fixes (Step 3, Phase 1 in Figure 3.1), HIPPOCRATES performs "fix reduction" by combining redundant bug fixes (Phase 2) based on source code location and operation. For example, two fixes which introduce flush instructions $F_1(X)$ and $F_2(X)$ which both flush X can be safely reduced to a single fix which creates flush $F(X)$, as this will still satisfy $X \rightarrow F(X) \rightarrow M \rightarrow I$. Likewise, fixes which create memory fences M_1 and M_2 where $X \rightarrow F(X) \rightarrow M_1 \rightarrow I$ and $X \rightarrow F(X) \rightarrow M_2 \rightarrow I$ can be safely reduced to a single fix which creates fence M, as this will still satisfy $X \rightarrow F(X) \rightarrow M \rightarrow I$. After all possible fix reductions are made, HIPPOCRATES determines which fixes should be "hoisted" (Phase 3), i.e., should instead be implemented as interprocedural fixes using the safe heuristic described below.

Heuristic description The heuristic uses a whole-program, interprocedural alias analysis to determine whether to transform an intraprocedural fix into an interprocedural fix, and if so, to determine which level in the function call stack to make the transformation. The heuristic aims to identify the persistent subprogram transformation that is least likely to operate on volatile data in order to avoid using persistency mechanisms on volatile data—the same intuition behind avoiding an intraprocedural fix in `memcpy` (§3.3.2). Listing 3.6 provides an example of the calculation for Listing 3.5.

The heuristic first marks all pointers as "PM" or "not PM" based on whether or not the pointer in the source code is associated with a PM modification event in the bug finder trace. For each bug, the heuristic constructs a list of candidate fix locations. The possible fix locations consist of (1) the original PM-modifying instruction and (2) the call sites of all functions in the call stack of the original PM-modifying instruction. If HIPPOCRATES performs a fix at (1) the original PM-modifying instruction, it will use an intraprocedural fix (i.e., by adding a flush/fence after the PM-modifying instruction). Otherwise (2) the system implements an interprocedural fix (i.e., performing a persistent subprogram transformation on the function called by a call site and updating the call site to call the transformed function). In Listing 3.6, the list of considered sites for the missing flush bug on Line 3 consists of Lines 19, 8, and 3.

The heuristic computes a score for each fix location in the list of possible fix locations. For the pointer argument to the PM-modifying instruction (1) or for each pointer argument to a call site (2), the heuristic calculates the score as the number of PM aliases (i.e., number of aliases to pointers marked as "PM") minus the number of non-PM aliases (i.e., number of aliases to pointers marked as "not PM"). A low score for a particular call site indicates that the call site frequently passes non-PM arguments to the called function, and it is thus more likely that a persistent subprogram transformation would operate on volatile memory. So, the heuristic chooses the fix location that has the highest score as the location to apply the interprocedural fix.

Note that the heuristic assigns a score of $-\infty$ to call-sites that do not pass any arguments along with all parents of this call site in order to prevent unnecessary persistent subprogram transformations. Intuitively, if a function has no parameters, it is either directly allocating PM or PM is being modified through global pointers; in either of these cases, performing a persistent subprogram transformation on the parameterless function or its parents provides no potential reduction in performance penalties (i.e., accidental durability mechanisms on volatile data) and therefore serves no purpose.

In Listing 3.6, the heuristic calculates a score for each candidate fix location. In Line 3, `addr` aliases both `vol_addr` and `pm_addr` from `foo`, so the line has 1 PM and 1 non-PM alias and a score of 0. For the call site on Line 8, the heuristic considers all pointer arguments, in this case only `addr`, which has 1 PM alias, 1 non-PM alias, and a score of 0. Finally, the call to `modify` on Line 19 has 1 PM alias (through `pm_addr`), 0 non-PM aliases, and a score of 1. Since the call to `modify` on Line 19 has the highest score, the heuristic performs a persistent subprogram transformation on `modify` and updates Line 19 to call the updated function, resulting in the transformed program shown in Listing 3.5.

Proof sketch of heuristic correctness Since the persistent subprogram transformation is a safe operation guaranteed to fix bugs (Theorem 4), the heuristic can insert a persistent subprogram at any point in the call stack as long as the durability ordering requirement for I is satisfied. The heuristic will only choose from fix locations which satisfy the durability ordering requirement for I. Therefore, the heuristic will insert safe and correct interprocedural fixes. The heuristic may also insert intraprocedural fixes. Intraprocedural fixes inserted by these heuristic are safe and

```
void update(char *addr, int idx, char val) {
    // non-PM alias: 1, PM alias: 1 = score: 0
    addr[idx] = val;
}

void modify(char *addr) {
    // non-PM alias: 1, PM alias: 1 = score: 0
    update(addr, ..., ...);
}

void foo(char *vol_addr, char *pm_addr) {
    for (int i = 0; i < INT32_MAX; i++) {
        // (This call contributes +1 non-PM alias)
        modify(vol_addr);
    }

    // (This call contributes +1 PM alias)
    // non-PM alias: 0, PM alias: 1 = score: 1
    modify(pm_addr);
    ↯ Crash occurs here ↯
}
```

Listing 3.6: **Heuristic Calculation Example.** An example heuristic calculation performed on Listing 3.5 to determine where to place the interprocedural fix.

guaranteed to fix the bug (Theorems 1, 2, 3). The heuristic therefore inserts safe intraprocedural and interprocedural fixes—as these are the only fixes produced by the heuristic, the heuristic inserts safe fixes. ■

3.5 Implementation

HIPPOCRATES is implemented primarily as an LLVM [91, 149] compiler pass (comprising 3300 LOC [155]) which locates the sources of the bugs, computes the appropriate fixes, and applies them (Figure 3.1, Steps 2–4). Step 1 (parsing bug finder output) is performed by Python scripts, which account for 1100 LOC (including some Python scripts used for orchestrating linking and running PMDK unit tests). We use an implementation of Andersen's alias analysis [5, 21] for the whole-program alias analysis we perform to compute our heuristic.

3.5.1 Collecting Traces and Identifying Bug Locations

Manually parsing the output of bug traces is challenging due to the size of these traces; for example, the `pmemcheck` traces in the Redis experiment are over 350 MB in size. This contributes to the difficulty of manually fixing PM durability bugs. Automating this process (Figure 3.1, Step 1), however, is fairly straightforward.

HIPPOCRATES relies on complete and accurate traces to identify bug locations in the LLVM bitcode, so we disable optimizations and function inlining, This limitation only applies to trace generation—a binary that includes HIPPOCRATES fixes can be fully optimized. Furthermore, compiling applications without optimizations for generating the PM bug trace is a non-issue with regards to performance, as the currently-available PM bug-finding tools are designed as offline testing tools due to their high overhead (ranging from 33% [104] to 400× [103]).

The main engineering challenge is mapping from source lines to LLVM Intermediate Representation (IR) using debug information (Figure 3.1, Step 2); however, HIPPOCRATES only requires this information for instructions that operate on PM, which simplifies the task. In practice, we use whole-program LLVM (WLLVM [136]) and are able to compile our applications into native machine code and into LLVM bitcode without having to make any modifications to the applications.

In principle, HIPPOCRATES can accept input from any PM bug finding tool; it currently supports `pmemcheck` and PMTest. HIPPOCRATES requires an input trace that contains the type, binary location, and call stack of each PM operation. `pmemcheck` provides this by default; we found it easy to port PMTest to provide the same information and expect the porting effort for other PM bug detection tools, such as AGAMOTTO, to be similarly easy.

3.5.2 Implementation of Fixes

As HIPPOCRATES is implemented in LLVM and computes its fixes on LLVM bitcode (Figure 3.1, Step 3), all fixes are generated (Figure 3.1, Step 4) as LLVM IR. Decompiling (mapping assembly or IR instructions back to lines of higher-level language code) is a difficult problem, however there are tools [81] which can convert LLVM IR back into C source code. This problem is made easier for HIPPOCRATES, as the generated fixes are simple; HIPPOCRATES inserts flush and fence instructions and duplicates functions, which are easy changes to automatically perform on source code.

3.6 Evaluation

In this section, we evaluate the effectiveness and usefulness of HIPPOCRATES. We start by validating the *effectiveness* of HIPPOCRATES (i.e., "Can HIPPOCRATES fix bugs?"); we then qualitatively evaluate the *accuracy* of HIPPOCRATES's fixes (i.e., "Are HIPPOCRATES's fixes similar to a developer's fixes?"); we then evaluate the performance of HIPPOCRATES's fixes (i.e., "Does HIPPOCRATES create efficient fixes?"); finally, we discuss the offline overhead of running HIPPOCRATES (i.e., "How expensive is running HIPPOCRATES?").

Evaluation Targets We evaluate HIPPOCRATES by testing representative state-of-the-art PM-applications and libraries. First, we test HIPPOCRATES on PMDK [32] libraries from Intel, as PMDK is the most active and well-maintained open-source PM project, which means it has a large set of validated bugs and fixes that we can use to assess the accuracy of HIPPOCRATES. We additionally evaluate HIPPOCRATES using three real-world PM applications to test the scalability and performance of HIPPOCRATES's fixes. We select memcached-pm [39], a PMDK-port of memcached, a popular high-performance memory caching server, that is maintained by Lenovo. We also test RECIPE's P-CLHT index [151], a state-of-the-art persistent and recoverable index representing a research prototype. Both memcached-pm and P-CLHT contain bugs which are detectable by `pmemcheck`, so we use them to evaluate the effectiveness of HIPPOCRATES on larger systems. Finally, we test Redis-pmem [33], a PMDK-port of Redis, a popular in-memory database and memory caching service, that is maintained by Intel. `pmemcheck` does not detect any bugs in Redis-pmem, so we use Redis-pmem as a baseline to compare the performance of HIPPOCRATES's fixes against a manually-developed bug-free implementation. We selected these targets as they are representative of the state-of-the-art PM-applications and libraries and have been tested in prior work [103, 104, 122].

Evaluation Workloads We evaluate HIPPOCRATES's effectiveness and accuracy on PMDK using the failing unit tests associated with the issues identified in our initial study of durability bugs and fixes (§3.3). We test P-CLHT using an example application used in RECIPE's evaluation, which manipulates the basic structure of the index through standard insertion, deletion, and lookup operations. We use YCSB [28, 29], a popular key-value store set of workloads, to test the Redis-pmem and memcached-pm server daemons.

Experimental Setup We run all of our experiments on a server with a Intel® Xeon® Gold 6230 CPU @ 2.10GHz. The server is equipped with 4 Intel Optane DC Series 100 NVDIMMs, each with 128GB capacity. The server is also equipped with 256 GB of DRAM.

3.6.1 Effectiveness

From our original study of 26 PMDK bugs, we attempt to reproduce the documented bugs by using the specified revision of PMDK specified in the initial bug report along with an up-to-date version of `pmemcheck`. Using this methodology, we are able to reproduce 11 bugs of the bugs in our study. To augment our evaluation, we also find 2 previously undocumented bugs in P-CLHT [95] and 10 previously undocumented bugs in memcached-pm. We are able to find all 23 of these bugs using `pmemcheck`, as all of these systems use PMDK libraries for their persistence mechanisms

Issue Numbers	HIPPOCRATES Fix	Developer Fix	Qualitative Fix Comparison
452, 940, 943	Intraprocedural flush (`CLWB`)	Interprocedural flush	Functionally equivalent; PMDK's fix is more portable
447, 458, 459, 460, 461, 585, 942, 945	Interprocedural flush+fence	Interprocedural flush+fence	Functionally identical

Table 3.2: **Comparison of HIPPOCRATES's Fixes.** Qualitative comparison of HIPPOCRATES fixes and PMDK developer fixes.

(libpmem, libpmemobj) and PMDK is properly instrumented to allow for `pmemcheck` to detect durability bugs.

HIPPOCRATES automatically repairs all 23 bugs we find and reproduce. We validate HIPPOCRATES's fixes by re-running `pmemcheck` against the repaired programs to determine that they no longer contain durability bugs. We further re-run the 11 bugs through PMDK's unit test framework and confirm that all unit tests succeed.

3.6.2 Accuracy

We present a qualitative comparison between HIPPOCRATES's fixes and developer fixes for the PMDK unit tests we were able to reproduce in Table 3.2. 8 of the 11 fixes (73%) were functionally identical to the PMDK developer fixes (issues #447, #458, #459, #460, #461, #585, #942, and #945). In all of these cases, HIPPOCRATES applies an interprocedural fix which functions identically to the developer fix, where the developers either used a persistent version of a function or inserted a specialized flush function to implement the interprocedural fix. We discuss the differences in the other 3 fixes (27%) below.

Direct versus indirect flushing For issues #452, #940, and #943, HIPPOCRATES generates an intraprocedural flush fix, whereas PMDK developers insert a libpmem flush function. HIPPOCRATES's fix produces correct functionality, as the data that needs to be flushed is within the size of a single cache line, however the fix generated by PMDK developers is potentially more machine-portable, as libpmem flush functions determine which kind of cache line function instructions are available at runtime. HIPPOCRATES could be modified to insert more generic fixes with some engineering effort, but some high-performance applications may prefer direct fixes instead.

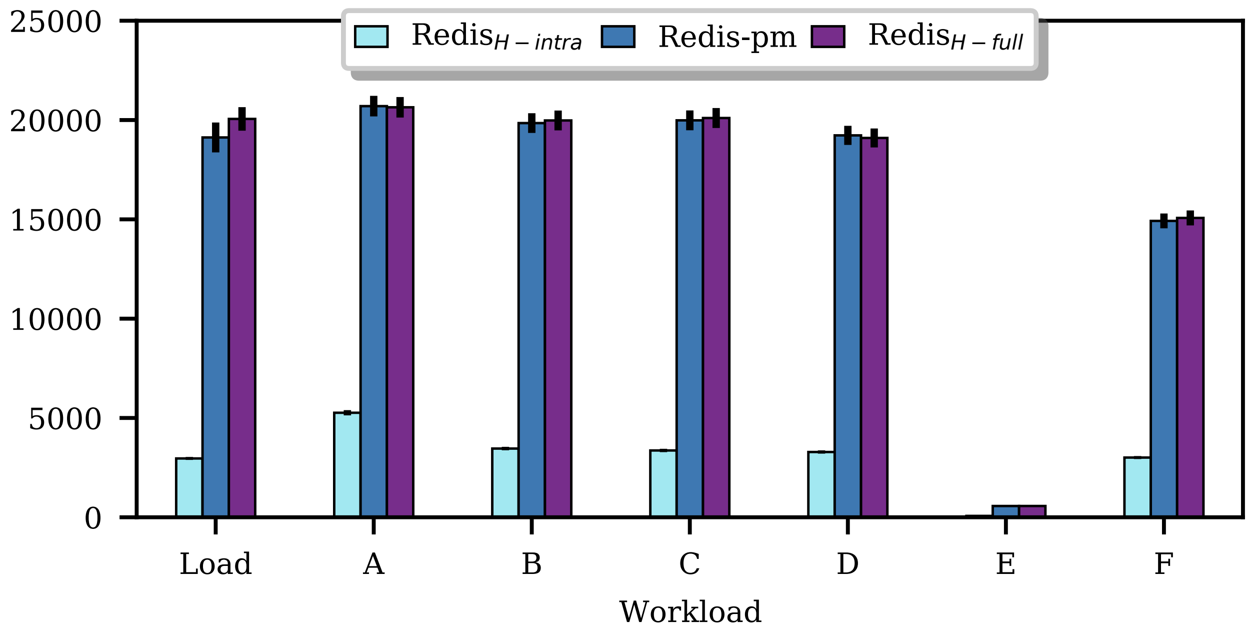


Figure 3.2: **Performance Analysis of HIPPOCRATES's Fixes.** Performance of the three persistent versions of Redis with 95% confidence intervals. HIPPOCRATES is able to provide fixes which are on-par with manual approaches.

3.6.3 Performance of Fixes

We want to ensure that HIPPOCRATES does not incur any undue performance degradation. Through our testing, we found that `pmemcheck` did not detect any bugs in Redis-pmem [33], indicating that this port of Redis had been thoroughly tested and debugged by developers. This makes Redis-pmem a good baseline to compare HIPPOCRATES's fixes against a system with all manually-developed durability mechanisms.

We perform a case study of Redis-pmem to compare HIPPOCRATES's abilites to the hand-tuned fixes of PMDK developers. We first remove all flushes in Redis-pmem. We leave memory fences, however, in order to preserve semantic ordering information which is required for proper crash consistency. We then run `pmemcheck` over this non-persistent version of Redis to generate a bug trace which can be consumed by HIPPOCRATES. We then run HIPPOCRATES over this trace to generate a version of Redis-pmem which has all of its persistence mechanisms auto-generated by HIPPOCRATES (Redis$_{\text{H-full}}$). We confirm that `pmemcheck` does not detect any durability bugs in Redis$_{\text{H-full}}$ (as is the case with our Redis-pmem baseline).

We also create a persistent version of Redis which fixes the persistence problems of the non-persistent Redis without using HIPPOCRATES's heuristic (Redis$_{\text{H-intra}}$). By disabling HIPPOCRATES's heuristic, HIPPOCRATES only applies intraprocedural fixes. While such fixes are sufficient for fixing durability bugs, they may also impact performance. HIPPOCRATES applies 50

	PMDK (Unit Tests)	**P-CLHT (RECIPE)**	**memcached-pm**	**Redis-pmem**
Combined KLOC	37	48	54	203
Time	6s	2s	2.2s	5m09s
Memory	345MB	148MB	147MB	870MB

Table 3.3: **Offline Overhead of HIPPOCRATES.**

fixes to make Redis persistent. In Redis$_{\text{H-intra}}$, all of these fixes are intraprocedural. In Redis$_{\text{H-full}}$, 12/50 (24%) of the fixes are interprocedural (10 are implemented 1 function above the PM modification and 2 are 2 functions above).

To compare the performance of these three persistent versions of Redis, we run each version with YCSB workloads [29] using a popular YCSB driver [28]. We use an entry and operation count of 10 thousand and run 20 trials for each workload. We report the throughput for all standard workloads (A–F) plus the time for the "load" operation (which sets up the initial state of the database for the other workloads). We show the results of this case study in Figure 2.4.

Redis$_{\text{H-full}}$ provides equal or slightly better performance than Redis-pmem (7% higher throughput on the "load" operation, which is the workload with the most durability operations, with the other workloads having equal performance within the 95% confidence intervals). This demonstrates that the fixes provided by Redis$_{\text{H-full}}$ are comparable to manual developer strategies for creating durable PM applications. HIPPOCRATES's ability to provide this quality of fixes is due to its analysis that enables the use of interprocedural fixes, as Redis$_{\text{H-full}}$ is between 2.4–11.7$\times$ faster than Redis$_{\text{H-intra}}$.

3.6.4 HIPPOCRATES's Overhead

Runtime Overhead We measure the overhead of HIPPOCRATES on all of our target systems and present the results in Table 3.3. This overhead is the offline overhead, meaning that it is only experienced during offline testing—HIPPOCRATES itself does not incur additional overhead (other than the overhead of the durability mechanisms it creates, see §3.6.3). This overhead is for fixing all bugs present in each system. HIPPOCRATES has low spatial and temporal overhead (at most taking around 5 minutes to run and less than 1 GB of memory), which allows HIPPOCRATES to be easily integrated into a developer's workflow.

Impact on Binary Size One potential consequence of the persistent subprogram transformation is increased code bloat due to function duplication, which could potentially lead to worse instruction cache (i-cache) performance. To mitigate this effect, Hippocrates performs persistent subprogram transformation once for each function and reuses transformations across interprocedural fixes if possible. The Redis experiment (§3.6.3) shows that Hippocrates creates minimal code bloat: Hippocrates introduces only 105 new lines of LLVM IR to flush-free Redis (an increase of 0.013%), which results in a binary that is only 4 KB larger than the manually-developed Redis-pmem (an increase of 0.05%). The performance results from the Redis experiment (§3.6.3) suggests that the performance benefits from interprocedural fixes outweigh the effect of additional i-cache pressure.

3.6.5 Results Summary

In our evaluation, we showed that HIPPOCRATES is effective, fixing all of the 23 bugs we found and reproduced using `pmemcheck` (§3.6.1). HIPPOCRATES is also accurate, fixing 8 of the 11 PMDK unit test bugs in ways functionally identical to developer fixes, while fixing 3 of the 11 bugs in functionally equivalent ways (§3.6.2). HIPPOCRATES's fixes also yield good performance, equalling or exceeding the performance of manually-developed durability mechanisms (§3.6.3). Finally, we show that HIPPOCRATES has low offline and size overhead (§3.6.4).

3.7 Discussion

Here we discuss some qualitative details about HIPPOCRATES's capabilities.

Fixing other kinds of PM bugs HIPPOCRATES only targets PM durability bugs (i.e., missing flush/fence bugs). By only targeting durability bugs, HIPPOCRATES can take input from the widest variety of PM bug finding tools and fix these critical correctness bugs.

Many PM bug finders report PM performance bugs (i.e., extraneous flush/fence bugs). However, fixing performance bugs (i.e., by removing flushes/fences) requires information about all possible execution paths (e.g., a flush may be extraneous in one execution and required for correctness in another). Existing PM bug detection tools cannot explore all execution paths of a large application, so it would be impossible to *safely* fix PM performance bugs except for in the simplest cases (e.g., redundant flush instructions in the same basic block). We therefore avoided trying to automatically fix PM performance bugs to avoid compromising HIPPOCRATES's "do no harm" design philosophy.

Some PM bug finders can also report PM ordering bugs (e.g., PMTest, XFDetector, and AGAMOTTO), which are crash-consistency bugs caused by the improper ordering of durable updates in PM (e.g., A is persisted before B, but B should have been persisted before A, and so if the program crashes after persisting A, the program is left in an inconsistent state). Fixing such bugs often requires reordering sequences of memory updates (e.g., moving the store to B before A), which can have unintended side effects (e.g., memory races in concurrent programs). *Safely* fixing these PM ordering bugs would require PM bug finders to output information about the safety of reordering memory operations (no existing PM bug finder can do this) or would require developers to provide safety specifications to HIPPOCRATES to encode this information. These approaches violate HIPPOCRATES's design goals (providing safe fixes and providing automatic fixes, respectively), so HIPPOCRATES does not support fixing such bugs.

Automatically providing durability The results of the Redis experiment (§3.6.3) raise the question of whether or not HIPPOCRATES can automatically provide durability to applications. HIPPOCRATES not only "does no harm", but HIPPOCRATES's fixes are provably correct (Theorems 1, 2, 3, and 4). However, HIPPOCRATES cannot currently provide automated durability because HIPPOCRATES can only fix bugs that are identified by an automated PM bug detection tool; current tools struggle to scale to entire programs and provide limited support for identifying all missing-fence durability bugs. HIPPOCRATES can still provide some automation, however—if a developer only specifies ordering points (i.e., memory fences), HIPPOCRATES can automatically inject cache line flushes when used in conjunction with a PM bug finder such as `pmemcheck`. This method of automation is essentially how we performed our experiment on Redis-pmem (§3.6.3).

3.8 Related Work

Persistent Memory Programming Frameworks As programming for PM using CPU primitives can be especially tedious and error-prone, prior work has examined many different frameworks and APIs for making PM programming easier and more intuitive. These range from specialized libraries (such as PMDK [32], NVM-Direct [11], and Pangolin [169]), to modified memory allocators (like Mnemosyne [154] and NV-Heaps [25]) to PM-specific language extensions (such as NVL-C [40] and NVM-Direct's preprocessor [11]). Various works also focus on logging mechanisms [16, 19, 55, 66] in PM to provide low-overhead memory consistency. However, these works do not prevent durability bugs, as APIs can be misused or can contain internal bugs (as we show in §3.3).

Persistent Memory Debugging Tools As discussed in §3.2.2, the challenges of debugging PM durability bugs has spurred many recent works in PM-specific bug detection [142]. PMTest [104] is a trace-validation framework, where each PM operation produces a trace event that is asynchronously validated to detect a durability bug. Some other tools use binary instrumentation to detect durability bugs. `pmemcheck` [31] is a binary instrumentation tool designed by Intel for

PMDK, which is based on Valgrind [123]. Persistency Inspector [126] is another tool developed by Intel, based on proprietary binary instrumentation included in Intel Parallel Studio XE. XFDetector [103] is a fault-injection tool based on Intel PIN, which is specifically tailored at finding crash consistency bugs caused by buggy PM update orderings.

General-Purpose Automated Program Repair Many systems are able to perform general-purpose program repair and solve any class of faults by using heuristics [92, 93, 139]. In particular, genetic programming (or Genetic Improvement [130]) is an increasingly popular method for automatic program repair. GenProg [92, 93], for example, uses a genetic programming method to mutate programs to generate fixes for off-the-shelf programs. Janus Manager [60] and SapFix [109] use genetic programming at industry scale, with SapFix supporting many of Facebook's core systems.

Automated Concurrency Bug Repair Work on automatically repairing concurrency bugs originally inspired us to look for more provably-correct ways to fix PM bugs. AFix [79], for example, specifically targets atomicity violations, and is able to correctly fix a majority of the bugs it targets and reduce the occurrence of bugs in all other cases. CFix [80] targets a wider variety of concurrency bugs—CFix accepts bug reports from a variety of concurrency bug finders [127] and produces fixes which are rigorously tested in a principled manner to provide some correctness guarantees.

3.9 Conclusion

Persistent Memory (PM) technologies aim to revolutionize the storage-memory hierarchy with disk-like durability at near-DRAM access latencies. However, even with specialized PM-bug finding tools, fixing durability bugs is challenging. We studied 26 PM bugs and their fixes and found that PM durability bugs can be fixed with fixes that are guaranteed to be safe. Based on our insights, we developed HIPPOCRATES, an automated PM bug fixing tool guaranteed to "do no harm." We used HIPPOCRATES to automatically fix all 23 durability bugs we found and reproduced. We further showed that HIPPOCRATES creates durability fixes that rival and exceed the performance of manually-developed durable code.

CHAPTER 4

SQUINT: Scaling Persistent Memory Crash-Consistency Testing via Representative Testing

Persistent Memory (PM) is a popular programming abstraction that enables direct memory access to durable data structures. PM provides applications with performance benefits, but makes it more difficult for applications to ensure consistency in the event of an untimely program crash. PM crash-consistency testing helps developers with this task, but are confronted with testing an exponential number of crash-states, or, the durable program state produced when a program crashes. Consequently, prior PM crash-consistency testing work forces developers to sacrifice either coverage (i.e., the tools miss bugs) or scalability (i.e., the tools cannot test large systems).

In this work, we introduce *representative testing*: a new PM crash-state reduction strategy that simultaneously achieves high scalability and high coverage. Our key observation is that many crash-states produced by a PM application can be considered *equivalent* because they evince the same crash-consistency bug, even though the crash-states are not themselves equivalent. We design a heuristic that approximates a small set of representative crash-states, or, a set of crash-states that is equivalent to all of the crash-states that an execution can produce. We build SQUINT, which uses representative testing to perform crash-consistency testing on only the small set of representative crash-states. We demonstrate that SQUINT achieves high coverage, since it finds 108 bugs (53 new) across 19 real-world PM applications, and show that it achieves high scalability, since it scales to real-world PM applications more effectively than existing works.

4.1 Introduction

Persistent Memory (PM) is a programming abstraction that enables developers to address durable storage using direct memory accesses [148]. While the PM abstraction has existed for decades [10, 144], recent hardware advances in low-latency and byte-addressable storage, such as Intel Optane

Pmem [34, 36, 77, 164] and CXL-attached non-volatile storage [27, 52], have spurred renewed interest in PM programming [68, 71, 114, 140].

Unfortunately, writing crash-consistent PM applications is challenging. A *crash-consistency* bug occurs when an untimely program crash during an otherwise correct execution produces a crash-state (i.e., the application state contained in non-volatile storage at the time of the crash) from which the application cannot recover. Software testing is the de facto approach for finding crash-consistency bugs in file systems and the applications that use them, but traditional tools struggle to scale to PM applications due to the massive state-space of unique crash-states that a PM application can produce. The large crash-state space of PM applications can be attributed to two reasons: First, applications update PM at a finer granularity using store instructions when compared to typical block-based storage systems [116]. Second, most modern PM platforms reorder the PM stores that are issued between explicit ordering instructions (§4.2.1). In sum, a single execution of a PM application can produce an exponential number of unique crash states, parameterized by the number of PM stores issued during the execution.

The research community has developed PM-specific crash-consistency testing tools to aid developers in building correct PM applications, but existing tools either sacrifice scalability (i.e., they cannot test large systems) or bug coverage (i.e., they miss crash-consistency bugs) in the face of the enormous PM crash-state space. One class of PM testing tools scales to large systems by either only testing for a narrow class of crash-consistency bugs [31, 41, 56, 103, 104, 122, 126] or by only testing crash-states that are created by patterns of PM updates that match the patterns of PM updates of bugs from other PM applications [50–52]. Unfortunately, such tools have low coverage as they miss crash-consistency bugs that fall outside the tested bug classes or that are not caused by known and tested buggy patterns (§4.2.4). Some of these tools even produce false positives (i.e., reporting correct behavior as a bug) when the tested application can recover from a crash-state produced by a "buggy" pattern found in previously-studied applications.

Alternatively, another class of PM testing tools [50, 57, 90, 97] achieve high coverage by employing model-checking techniques that generate all possible crash-states and testing them for crash consistency using a testing oracle (e.g., an application's built-in consistency checks). The state-of-the-art in PM model checking uses Dynamic Partial Order Reduction (DPOR) techniques [44, 57], an advanced technique from the testing community, to reduce the number of crash states that it must test. Nevertheless, existing model checking tools fail to scale to real-world PM systems [57] due to the volume and fine-granularity of PM updates in these applications.

In this work, we propose a new state-space reduction technique, *representative testing*, that builds on DPOR to scale PM model checking to real-world applications. Our key insight is that the crash-consistency of crash states is often highly correlated, even when those crash states are *not* identical. For example, if c_1 and c_2 are crash-states produced by a program crash immediately before the same instruction i in a function `foo`, but from `foo`'s invocations in different contexts, then the crash consistency of c_1 is likely to be the same as the crash-consistency of c_2, even though the two crash-states are independent. Representative testing would limit its testing to only c_1 or c_2, while existing PM DPOR techniques would test c_1 and c_2 since they are produced by different sequences of PM stores.

Representative testing employs this insight as a new equivalence relation on top of DPOR: the technique considers crash-states to be equivalent if the crash-states are likely to exhibit the same crash-consistency bugs or lack thereof (i.e., the crash-states have highly-correlated crash consistency). The approach only tests a single representative crash-state from each equivalence class. Representative testing scales strictly better than DPOR since it produces at most the equivalence classes from DPOR (representative testing always places equivalent crash states, as determined by DPOR, into the same equivalence class). In practice, we observe that representative testing prunes the crash-state state space to scale to significantly larger PM applications than prior model checking tools (§4.6).

Alas, soundly identifying crash-states that have correlated crash-consistency is intractable, as a naive approach requires testing all crash-states from a given execution. So, we analyze PM use (and misuse) in existing applications to identify an *update behavior*-based heuristic that approximates correlated crash-states without performing crash-consistency testing on each crash-state. The heuristic identifies sequences of PM updates that produce crash-consistency correlated crash states. First, it splits the PM updates from the execution into *update behaviors*, sequences of PM updates to the same data object that occur with high temporal locality. Then, the heuristic identifies one update behavior, u_1, as representing another update behavior, u_2 if u_1 and u_2 operate over an object of the same type (regardless of whether the operations assign the same value or when u_1 and u_2 are executed relative to one another) and u_1 imposes a subset of the ordering constraints imposed by u_2. Intuitively, u_1 represents u_2 because each of the crash-states producible by crashing the program after each update in u_2 is correlated with a crash-state producible by crashing the program after some updates in u_1. The heuristic is unsound, so it reduces the bug coverage of representative testing. Nevertheless, in practice, we observe that the heuristic achieves higher bug coverage than existing tools given practical testing restrictions (§4.6).

We build SQUINT, a model-checking tool for testing PM crash-consistency that uses representative testing and the update behavior heuristic. SQUINT traces the execution of a PM application to identify all PM updates. Then, the system uses the update behavior heuristic to identify the set of representative update behaviors. It applies DPOR to create the crash-states of only the representative update behaviors rather than the entire program trace [57]. If the application cannot recover from a crash-state produced by an update behavior, u_1, SQUINT also tests the update behaviors that were represented by u_1 to identify all instances of a particular type of crash-consistency bug throught the application.

Experimentally, we show that representative testing is effective by using SQUINT to find PM crash-consistency bugs across 19 well-tested PM applications. SQUINT finds 108 crash-consistency bugs, including 53 new bugs. We found 52 of these new bugs in applications that were rigorously tested by prior work [41, 50, 51, 102–104, 122], demonstrating the efficacy of representative testing. We also compare SQUINT and representative testing against Jaaru [57], the state-of-the-art DPOR-based model checking tool for PM, and find that SQUINT is able to find 8.5× as many bugs as Jaaru can within a 2 hour testing time limit.

In summary, we make the following contributions:

- We design representative testing, a state-space reduction method that reduces the crash-state testing space by eliminating the testing of crash-states that are likely to evince the same crash-consistency bugs.
- We build SQUINT, a PM testing tool that implements representative testing by automatically finding update behaviors that approximate the characteristic set of crash-states in an execution of a PM application.
- We use SQUINT to evaluate representative testing and find 108 PM crash-consistency bugs (53 new) in real-world PM applications, demonstrating the efficacy of representative testing over prior PM crash-consistency testing approaches.

4.2 Background

4.2.1 Persistent Memory and Persistent Memory Technologies

Persistent Memory (PM) is a programming abstraction in which programs address durable storage using the same memory instructions that they use for main memory [9, 144, 148]. When using PM, an application maps persistent data into its address space rather than using expensive file-system IO calls to update block-storage devices [77, 164]. The PM abstraction can be implemented in many ways. For example, PM can be implemented as a software abstraction on top of a file system (e.g., `mmap`) with libraries [32, 169], or language run-times [40, 158]. PM can also be implemented via direct access to non-volatile main memory (e.g., Intel Optane Pmem [34, 36, 77, 164]) or other non-volatile hardware mechanisms (e.g., CXL.mem with non-volatile storage [27, 52]).

Most PM platforms require explicit memory ordering instructions (e.g., memory fence instructions, CXL's Global Persistent Flush, or `msync`) to enforce a specific ordering between PM updates. Without these memory ordering instructions, the underlying PM implementation is free to

```
1  // Assume entry spans multiple cache lines
2  typedef struct entry {
3      uint8_t key[KEY_LEN];
4      uint8_t value[VALUE_LEN];
5      bool valid;
6  } entry_t;
7
8  void insert(key, value) {
9      entry_t *new_entry = ...;
10     new_entry->key = key;
11     new_entry->value = value;
12     new_entry->valid = true;
13     ϟ Crash occurs here ϟ
14     // Bug: updates to key, value, and valid fields may be
15     // reordered, causing valid=true with empty key or value
16     PERSIST(entry);
17 }
18
19 void check_consistency(entry_t *entry) {
20     if (entry->valid) {
21         assert(strlen(entry->key) > 0);
22         assert(strlen(entry->value) > 0);
23     }
24 }
```

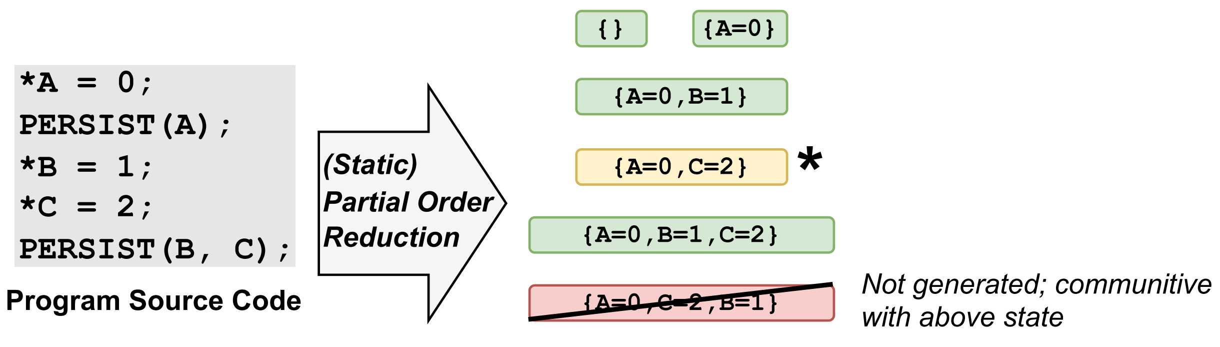
Listing 4.1: **Crash-Consistency Bug Example #1.** A simplified excerpt of a crash-consistency bug in Level hashing [172].

reorder updates, which improves memory throughput but may violate crash-consistency requirements [106, 121]. In this paper, we focus on such platforms that reorder PM stores since they are more common due to their increased efficiency.

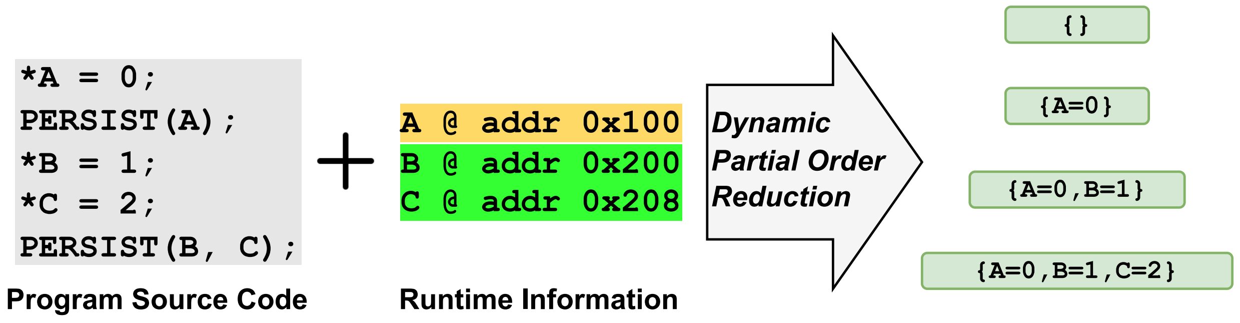
4.2.2 Persistent Memory Crash-Consistency Bugs

An application is *crash-consistent* if it operates as intended even if there is an application crash. For example, a crash-consistent application could restart and resume processing new user requests without losing previously input data. A *crash-consistency bug* occurs when an application's state becomes inconsistent due to the interruption of a sequence of updates that cannot be recovered through recovery mechanisms (e.g., rolled back, re-executed, or ignored). Crash-consistency bugs can arise due to programmer error (e.g., updates written in the wrong order) or due to compiler bugs that alter the intended order of persistent updates [58].

Consider the example crash-consistency bug in Listing 4.1. The application inserts a new key-value pair into a hash table entry and sets the `valid` field to `true` (lines 9–12). This update order *would* ensure crash-consistency, but the updates to these fields are not guaranteed to be persisted



(a) Partial Order Reduction



(b) Dynamic Partial Order Reduction

Figure 4.1: **POR versus DPOR.** A comparison of how DPOR compares to POR when performing model checking on PM applications.

due to the lack of `PERSIST`[1] calls and therefore may be persisted in any order. Therefore, if a crash occurs after the `valid` field is set (line 12) but before the `key` and `value` are guaranteed to be persisted (line 16), the updates to `key` and `value` may be lost, resulting in inconsistent data and crash-consistency violations (e.g., an assertion failure on line 20).

4.2.3 (Dynamic) Partial Order Reduction

Partial Order Reduction (POR) is a technique to reduce a testing tool's state space. POR uses static program analysis to identify commutative operations and elide testing states that differ only the the order of such operations. We show an example of how POR works when generating crash-states to test the program in Figure 4.1a. Since PM updates cannot be reordered around `PERSIST` calls, POR identifies that any crash state containing `B=1` or `C=2` must contain `A=0`. Furthermore, since stores to different addresses are commutative (e.g., applying `B=1` and `C=2` in either order will result in identical crash-states), POR elides crash-states that differ only in the ordering of these stores.

[1]The details of persistence functions vary on the underlying PM platform.

While POR reduces redundant crash-states, *Dynamic* Partial Order Reduction (DPOR) provides further savings by leveraging ordering constraints that are only observable at runtime. DPOR executes the program multiple times and traces the execution, collecting information on ordering constraints and re-executing the program to generate new, non-redundant states [44]. Figure 4.1b shows an example of DPOR applied to PM. DPOR identifies the same constraints and commutative relationships as POR, but also determines that `B` and `C` are allocated on the same cache line. Thus, on Intel CPUs, which ensure total store ordering, `C=2` cannot be persisted unless `B=1` is also persisted [134]. Therefore, DPOR will not generate the state marked with an asterisk in Figure 4.1a, since `C=2` cannot be persisted without `B=1` in this execution. With such runtime information, DPOR generates fewer states that POR.

4.2.4 Prior PM Crash-Consistency Testing Approaches

PM crash-consistency testing is challenging due to the wide variety of ways that crash-consistency bugs can occur in applications and the vast number of possible crash states. Ergo, researchers have developed numerous PM crash-consistency testing tools. In general, there are two main approaches: exhaustive testing and application-specific/bug-specific pruning.

Exhaustive testing tools use *stateless model checking* [54] to test for PM crash-consistency bugs by testing all possible PM crash states [57, 90, 97] and thus achieve high crash-state coverage. The state-of-the-art tool, Jaaru [57], uses PM-specific DPOR methods to avoid testing equivalent crash-states. For example, crash-states c_1 and c_2 may not be byte-for-byte equal, but the recovery procedure of the PM application only reads from a small set of memory locations that are equal in both c_1 and c_2, making the recovery (i.e., the post-crash execution) identical and thus makes c_1 and c_2 functionally equivalent [57]. However, even DPOR testing tools cannot scale to real-world systems (§4.6).

To overcome the scalability problems of exhaustive testing, most prior approaches eschew POR/DPOR techniques and instead aim to prune the crash-state testing space by testing crash-states or executions that match *patterns* extrapolated from known PM bugs and well-studied applications [22, 31, 41, 50–52, 58, 102–104, 122, 126]. However, extrapolating patterns from known bugs and applications leads to false negatives when the crash-consistency requirements of a PM application do not conform to extrapolated patterns or when new, unseen patterns of bugs are encountered (as we demonstrate by finding new bugs in applications that have been extensively tested by prior work; §4.6.3).

```
1  void replace(old_key, new_value) {
2      entry_t *new_entry = ...;
3      entry_t *old_entry = ...;
4      new_entry->key = old_key;
5      new_entry->value = new_value;
6      new_entry->valid = true;
7      old_entry->valid = false;
8
9      PERSIST(new_entry);
10     PERSIST(old_entry);
11 }
```

Listing 4.2: **Crash-Consistency Bug Example #2.** A simplified excerpt from another crash-consistency bug in Level hashing [172]. `entry_t` is defined in Listing 4.1.

4.3 Representative Testing

Prior work in PM crash-consistency testing demonstrates the limitations of pure DPOR-powered model checking in testing real-world PM applications. The goal of this work is to develop a new state-space reduction technique that can improve upon DPOR-based approaches without the application-specific or bug-specific optimizations introduced by other prior works, as they introduce many false negatives and/or false positives.

We analyze PM use across existing applications and observe that many crash-states that are not equivalent according to existing DPOR relations *are* equivalent with regard to the crash-consistency bugs that they manifest. Consider the example in Listing 4.2. In the `replace` function (Line 1), the level hash table updates an existing hash table entry by inserting the new value into a new entry, setting it as valid (Line 6), and invalidating the old entry (Line 7). The `replace` function should be atomic, but it does not enforce orderings in the updates (between lines 4–6). Consequently, the update may become visible (i.e., `valid` set to true) before `key` and `value` are persisted and cause a crash inconsistency. This update to a level hash entry is the same crash-consistency bug as the `insert` function in Listing 4.1: the ordering constraints on the updates to `new_entry` are the same in both the `insert` and `replace` functions (i.e., no ordering constraints are enforced between updates to `key`, `value`, and `valid`) which causes the same consistency violation in the `check_consistency` function (Listing 4.1, Line 19). We can therefore say that crash-states created by crashes in `insert` are equivalent to crash-states created by crashes in `replace`.

This observation is comparable to observations made in concurrency and distributed system testing works [96, 117, 133]. These works overcome the limitations of DPOR by further limiting state-space exploration using either global exploration bounds (e.g., number of thread preemptions in concurrent systems), or domain-specific knowledge (e.g., ordering constraints between different

```
void insert_ordered(key, value) {
    entry_t *new_entry = ...;
    new_entry->key = key;
    PERSIST(&new_entry->key);
    new_entry->value = value;
    new_entry->valid = true;
    // A crash here exposes some, but not all, of the bugs
    // found in the original insert function (see Listing 4.1).
    ⚡ Crash occurs here ⚡
    PERSIST(new_entry);
}
```

Listing 4.3: **Synthetic Crash-Consistency Bug with Added Ordering Constraints.** `insert_ordered` is a synthetic version of `insert` (Listing 4.1) with an added ordering constraint. With the added ordering constraint, `insert_ordered` exposes some, but not all, of the crash-consistency bugs that `insert` can expose.

messages or failure events in distributed systems). While these approaches are often unsound (i.e., can miss bugs) or are only sound within a set bound, these techniques are able to find deep bugs in real-world systems that cause state-space explosion for DPOR techniques.

Based on our observation, we propose *representative testing*, an approach to crash-state reduction that identifies and tests a small set of crash-states, C, from an execution such that every crash-state that can be generated by the execution is equivalent to a crash-state in C. In other words, representative testing aims to test a small set of crash-states that *represent* the other crash-states in the execution.

Unfortunately, it is intractable to identify such a representative crash-state set as such a brute-force approach requires generating and exhaustively testing every crash-state in an execution. Therefore, we derive a method to *approximate* this characteristic crash-state set to employ representative testing.

We further observe that many PM applications create equivalent crash-states from *update behaviors*—updates to single PM data structures with high temporal locality. Consider the example in Listing 4.2. The crash-consistency violation that occurs is dependent on a single data structure—`entry_t`. Furthermore, the set of updates that cause the `entry_t` to become inconsistent happen only within the context of the `replace` function, regardless of the history of other updates that may or may not have happened to that `entry_t` instance. We observe that the crash-consistency of many PM applications rests on the consistency of individual data structures, which implies that the crash-consistency of update behaviors is crucial and central to the crash-consistency of PM applications.

This observation leads us to the *update behavior heuristic*, a heuristic for approximating true representative testing. Since multiple different updates can expose the same crash-consistency

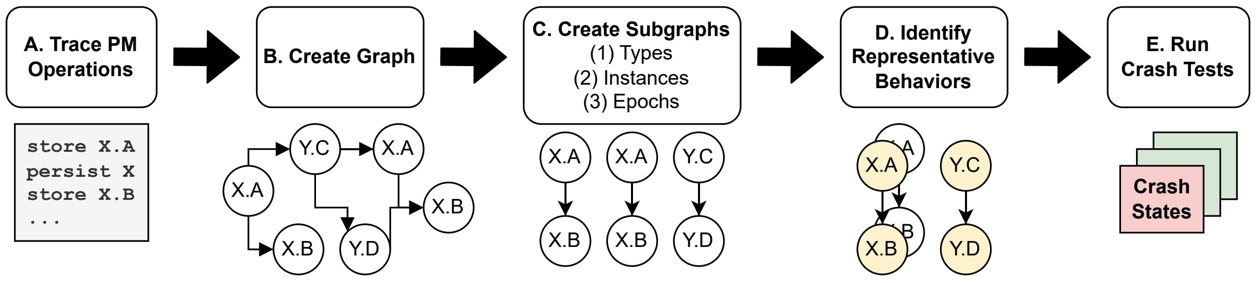


Figure 4.2: **Functional Overview of SQUINT.**

bugs, our insight is that one, *representative* update behavior to a particular type of data structure can be tested on the behalf of the other updates. If a crash-consistency bug is found due to the ordering constraints enforced (or not enforced) in the representative update, it is then likely that the represented behaviors can create crash states that are also crash inconsistent. Since this heuristic only requires the analysis of a program trace rather than all possible crash-states resulting from that trace, this heuristic is an efficient approximation for true representative testing.

The update behavior heuristic selects representative behaviors that update the same or more fields of the data type while enforcing the same or fewer ordering constraints on that object than other update behaviors. Consider the example in Listing 4.3, which contains a version of the `insert` function (called `insert_ordered`) that attempts to fix the crash-consistency bugs of the original function, but only inserts an additional ordering constraint between the `key` and `value` update (Line 4). Therefore, for a particular update (e.g., `insert` or `replace` operation), the *representative* update is the largest update (i.e., updates the most fields) that has the fewest ordering constraints, as this update will generate the most crash-states that can be tested for representative crash-consistency violations (fewer ordering constraints allows for more possible orderings and therefore more possible crash states). For example, in the case of the level hash table entry, either `insert` or `replace` could be the representative update for the update behavior in `insert_ordered`, as `insert_ordered` has more ordering constraints and thus creates fewer possible crash-states.

4.4 Design of SQUINT

We now discuss how we design and implement representative testing in SQUINT. SQUINT implements representative testing by identifying update behaviors in the execution of a PM application, grouping them together by the ordering constraints they enforce, and then testing a representative update behavior from each group for crash consistency (i.e., the update behavior heuristic). In order to create and test a characteristic set of crash-states, SQUINT needs to identify which PM stores and ordering instructions constitute a single update behavior. Therefore, SQUINT employs a heuristic approach that approximates if PM operations are related based on (1) the data types modified, (2) the instance of the data types, and (3) the temporal locality of the the PM operations (§4.4.3) in order to automatically (i.e., without developer effort) identify update behaviors.

SQUINT is based on PM crash-consistency model-checking workflows (see Figure 4.2), which first execute a PM application to generate a trace of PM operations (i.e., stores, cache-line flushes, and memory fences) and then perform crash-consistency testing by testing the PM application's ability to recover from crash-states that result from all possible update orderings. However, unlike prior exhaustive testing approaches, SQUINT only tests crash-states generated by representative update behaviors rather than all possible crash-states. Furthermore, for each update behavior that SQUINT tests, SQUINT further reduces the number of generated crash-states by leveraging DPOR state-reduction techniques.

Specifically, SQUINT first traces PM operations during execution (Step A). SQUINT then converts this trace into a *persistence graph* that contains all PM stores in the execution as nodes and all ordering constraints between PM stores as edges (Step B). SQUINT divides the graph into subgraphs by analyzing the data structures and temporal locality of PM instructions from the execution (Step C); each subgraph represents an update behavior from the original execution. SQUINT then places subgraphs (i.e., update behaviors) into groups by adding a subgraph to a group if the group's representative (i.e., the largest subgraph in a group) includes a superset of the PM updates and orderings of the other subgraph (Step D). SQUINT finally uses *crash-consistency model checking* (Step E) to test all possible crash-states from each representative subgraph.

Crash-consistency model checking, as also described in prior work [57, 90, 97], validates crash states using a consistency-checking routine (e.g., a specialized function or application recovery code). Since SQUINT uses crash-consistency model checking, it does not report false positives (i.e., all bugs are true bugs), but may report false negatives (e.g., if the checking routine cannot detect the bug).

We now discuss each step of SQUINT's testing process in detail (§4.4.1–4.4.5), then discuss SQUINT's limitations (§4.4.6).

4.4.1 Tracing PM Operations (Step A)

SQUINT traces PM operations during an execution of the PM application under test, comprising a list of PM update operations (§4.2.1) in the order in which they are executed by the CPU. (A) in Figure 4.3 provides an example PM program snippet and (B) in Figure 4.3 shows the resulting trace. Different program executions may generate traces that follow different paths through the program that perform different PM updates—we discuss trace coverage in §4.4.6. SQUINT can

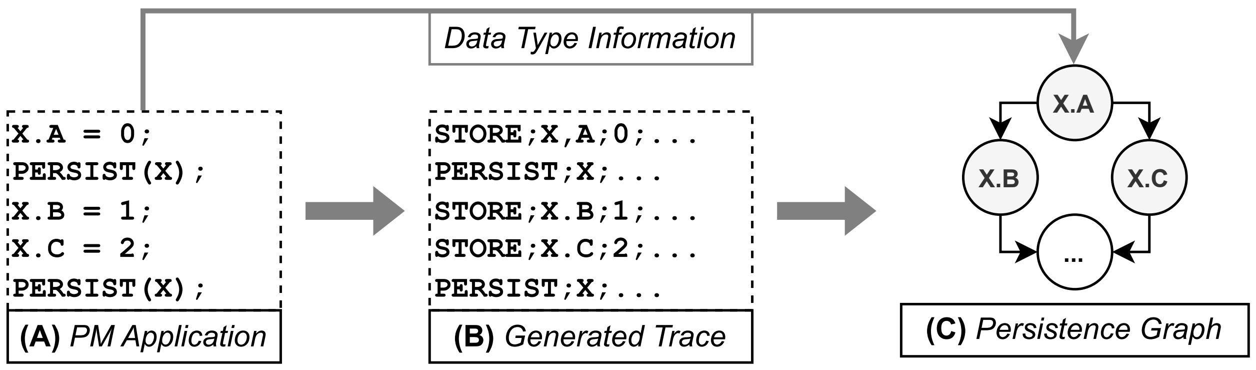


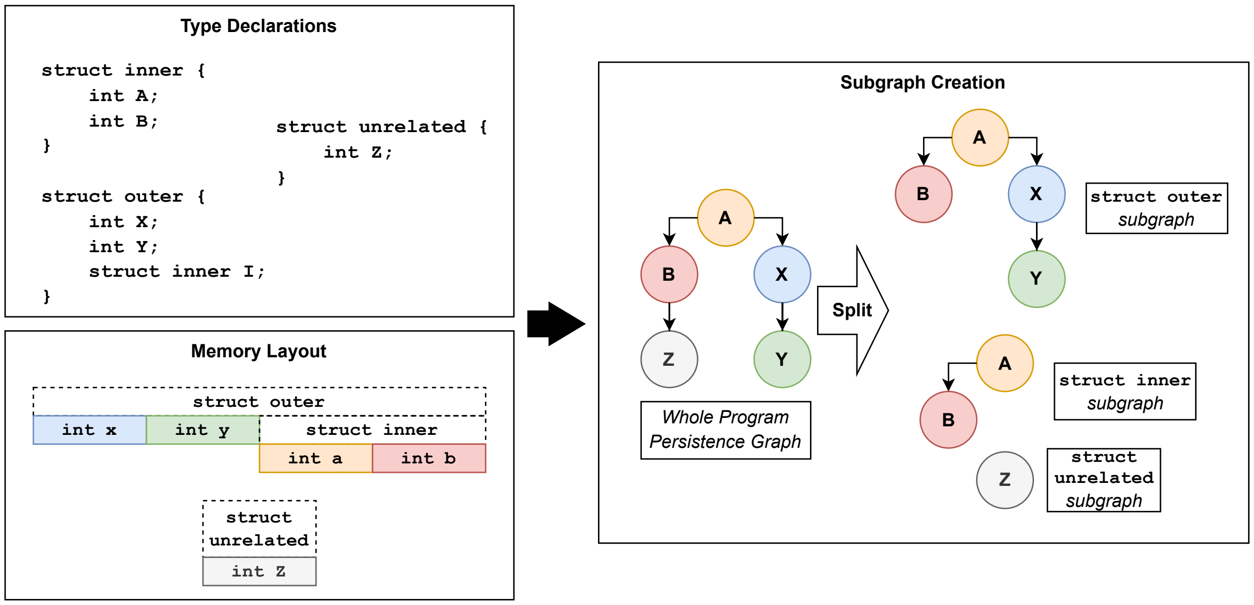
Figure 4.3: **Persistence Graph Construction (Step B).**

use a single PM operation trace to reason about all of the ways that PM updates in the trace could be reordered at run-time, since operations in the trace describe all memory ordering constraints on PM updates for the given program input (§4.4.2).

4.4.2 Persistence Graph Construction (Step B)

Next, SQUINT uses the PM operation trace (Step A) to create a persistence graph, which contains all of the ordering constraints that the application enforced between all of the PM stores in the execution. A persistence graph represents each PM store in the trace as a node, and represents the ordering constraints between PM stores as edges. (C) in Figure 4.3 shows the persistence graph generated by the application, (A), and the execution trace, (B). In this example, the store to $X.A$ is persisted before either store to $X.B$ or $X.C$, and so the node representing the store to $X.A$ has outgoing edges to $X.B$ and $X.C$. The updates to $X.B$ and $X.C$ are not ordered with respect to each other, so there is no path from either $X.B$ to $X.C$ or $X.C$ to $X.B$. Since the execution persists both $X.B$ and $X.C$ before the end of the code snippet, the nodes have outgoing edges to subsequent nodes in the graph.

The PM operation trace only contains run-time information about each operation (e.g., memory addresses, binary code addresses, and stack traces) and does not contain the data-type information required for identifying update behaviors (§4.4.3). So, SQUINT augments the PM operations in the trace with data-types by using debug information from the recorded binary code address to identify which fields are updated by each update in the persistence graph. To improve accuracy, SQUINT identifies the original data-type of each PM update by following the recorded stack traces backwards to determine the original type of all type-casted data types (e.g., when updates are made indirectly via `memset`). We discuss the details and limitations of this process in §4.5.



```
struct inner {
    int A;
    int B;
}

struct outer {
    int X;
    int Y;
    struct inner I;
}

struct unrelated {
    int Z;
}
```

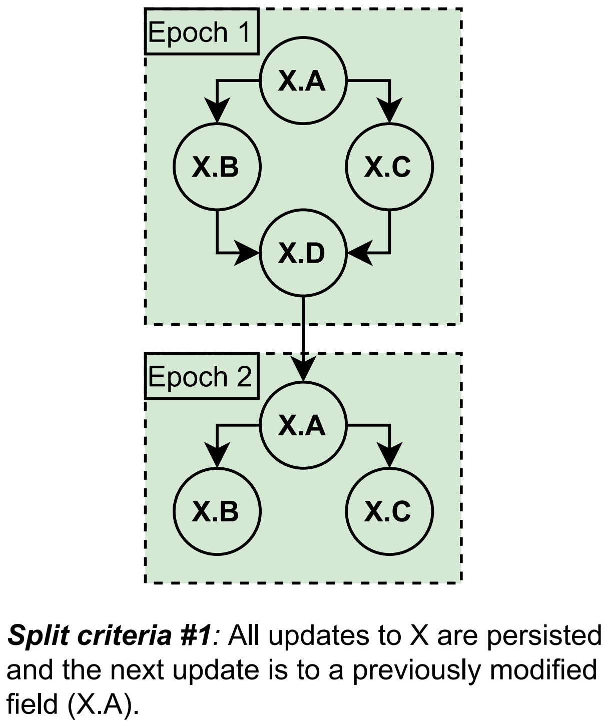
Figure 4.4: **Type Subgraph Creation.** An example of how the persistence graph is grouped into type subgraphs (Step C1).

4.4.3 Subgraph Creation (Step C)

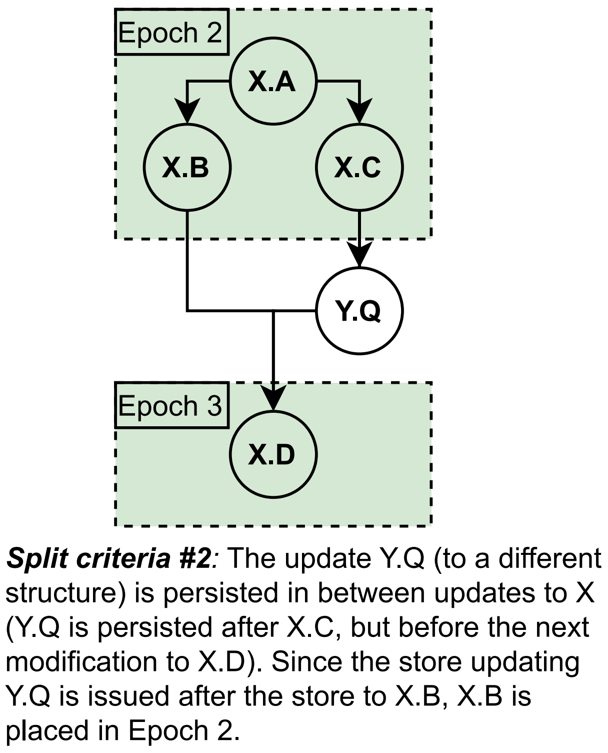
After constructing the persistence graph (Step B), SQUINT splits the persistence graph into subgraphs to identify the program's update behaviors. Alas, there are an exponential number of possible subsets of PM operations, and thus an exponential number of ways to group PM operations together to describe an update behavior. Nevertheless, two properties of PM programs allow SQUINT to automatically identify the update mechanisms in an execution: first, PM programs typically place semantically-related data fields in the same data structure, and second, semantically-related PM updates usually occur with high temporal locality.

SQUINT uses the aforementioned observations to build a heuristic method to identify update behaviors in the persistence graph. First, SQUINT groups updates together based on data-type (Step C1) and data-type instance (Step C2). Then, SQUINT groups together updates that occur close together temporally, by splitting the persistence graph into *epoch*s (Step C3). We outline these three steps below:

(C1) Split the persistence graph into *type subgraphs* First, SQUINT splits the persistence graph into subgraphs by data type since crash-consistency is a property of data types. SQUINT must reason about subgraphs that capture ordering requirements among the constituent types of a composite type (e.g., a struct that has another struct as a field) since constituent types in a composite type may have crash-consistency ordering requirements. Ergo, a node may be placed in multiple type sub-


Epoch 1
X.A
X.B
X.C
X.D
Epoch 2
X.A
X.B
X.C
***Split criteria #1**: All updates to X are persisted and the next update is to a previously modified field (X.A).*

(a) Split Criteria #1.


Epoch 2
X.A
X.B
X.C
Y.Q
Epoch 3
X.D
***Split criteria #2**: The update Y.Q (to a different structure) is persisted in between updates to X (Y.Q is persisted after X.C, but before the next modification to X.D). Since the store updating Y.Q is issued after the store to X.B, X.B is placed in Epoch 2.*

(b) Split Criteria #2.

Figure 4.5: **Epoch Splitting Example.** An example of how an instance subgraph (all shaded nodes) is split into epoch subgraphs (Step C3) by using information from the full persistence graph.

graphs if the type corresponding to the nodes is used in a composite type. Figure 4.4 provides an example, where nodes of type `struct inner` are included in the type graph of both `struct inner` and `struct outer`. In contrast, node Z is of type `struct unrelated`, which is not a composite type, and therefore node Z only appears in one, single-node subgraph. SQUINT performs this analysis for all, non-primitive data types, which includes types likes structures, classes, and arrays.

(C2) Split type subgraphs into *instance subgraphs* Next, SQUINT splits each type subgraph produced in Step C1 into an *instance subgraph*, since the individual object is the unit of crash consistency. Each node in the type subgraph refers to a specific object (i.e., *instance* of a datatype) based on the address and field offset of the node's PM store. SQUINT assigns each node in a type subgraph to an instance subgraph based on the object the node references.

(C3) Split instance subgraphs into *epoch subgraphs* Finally, SQUINT splits each instance subgraph produced in Step C2 into an *epoch subgraph*. We only need to test individual update behaviors for crash consistency. We therefore split the instance subgraphs (which span the entire program execution) into epoch subgraphs that represent individual update behaviors, with operations that have high temporal locality.

For example, consider the persistence graph in Figure 4.5. The graph contains the instance subgraphs of X (represented by the shaded nodes for updates to X) and Y ($Y.Q$). The three distinct update regions for X (Epochs 1–3) can only all be distinguished when considering nodes outside of X's instance subgraph. While Epochs 1 and 2 are split based on a data-structure specific criteria (see below; Figure 4.5a), the boundary between Epochs 2 and 3 can only be drawn when considering the update to $Y.Q$ from Y's instance subgraph, as update $Y.Q$ sits between the two Epochs (Figure 4.5b). Ergo, SQUINT breaks each instance subgraph temporally into *epoch*s by using information from the full persistence graph (§4.4.1) to place PM updates that occur close-together in time into the same *epoch subgraph*.

SQUINT uses two criteria to choose the boundary of an epoch subgraph. First, SQUINT splits the subgraph for instance X into epochs at all PM instructions I in the execution such that all stores to X that were issued before I are persisted before I and I updates a previously updated field of X. Once an epoch is created this way, the tracking for repeated fields resets at the beginning of the new epoch. In Figure 4.5a, this criteria demarcates Epoch 1 from Epoch 2, since the execution again updates $X.A$ (instruction I in the definition) after it persists all prior updates to X ($\{X.A, X.B, X.C, X.D\}$). Since the field tracking is reset at the beginning of Epoch 2, another epoch is not created for the updates to $X.B$ and $X.C$, even though they were previously modified in Epoch 1.

Second, SQUINT splits the instance subgraph into epochs between updates in the subgraph of U, U_1 and U_2, if there exists a PM update V in the execution such that V does not update U and U_1 is persisted before V and V is persisted before U_2. We also show an example of this splitting criteria in Figure 4.5b in how Epochs 2 and 3 are divided. Here, the update to $Y.Q$ is persisted between two updates to X ($X.C$ and $X.D$). Since $X.C$ is persisted before $Y.Q$ and $Y.Q$ is persisted before $X.D$, this breaks the flow of updates to X, and so SQUINT splits this part of X's instance subgraph into Epochs 2 and 3.

The resulting epoch subgraphs are the update behaviors in the execution.

4.4.4 Grouping Update Behaviors (Step D)

After identifying the program's update behaviors (Step C), SQUINT groups behaviors by their representatives. SQUINT considers one behavior to represent another if one behavior contains a

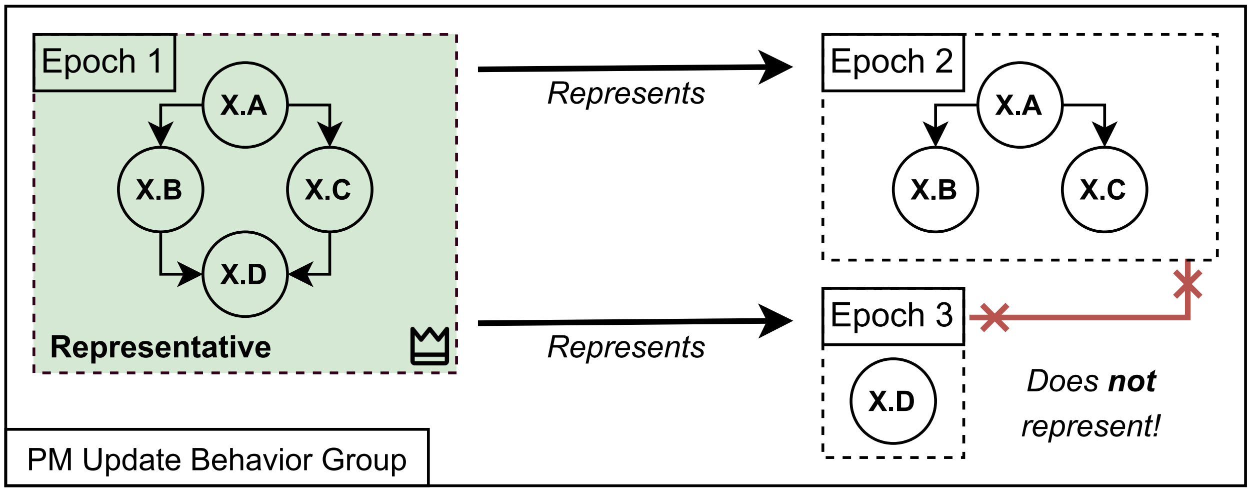


Figure 4.6: **Update Behavior Grouping Example.** An example of how the update behaviors found in Figure 4.5 are grouped by similarity and how Epoch 1 is chosen as the representative (Step D).

superset of the PM updates as the other while enforcing the same or fewer ordering constraints as the other. By this definition, the representative behavior will contain a superset of possible update orderings and therefore create a superset of equivalent crash-states. SQUINT uses the largest update behavior (i.e., allows for the most unique update orderings) in each group as the representative. Using the largest update behavior as the representative ensures that the representative for each group represents all group members by the transitivity of representative relation (e.g., if U_1 represents U_2 and U_2 represents U_3, U_1 represents U_3).

SQUINT forms groups as follows. SQUINT iterates through all identified update behaviors in sorted order, from largest update behavior to smallest. It adds each update behavior to every existing group that represents it, where a group represents an update behavior if the behavior has a subset of updates and possible update orderings of the largest behavior in the group (i.e., the representative of the group). Formally, if R_G is the representative update behavior of group G, then an update behavior A is represented by R_G if A contains a subset of nodes in R_G and the subset of nodes in R_G have the same or fewer ordering constraints as A. SQUINT considers two nodes equal if they update the same field in the same data-type, but does not consider which instance is updated since crash-consistency is a property of data-types, not individual instances. If there is no group that represents the update behavior, then SQUINT forms a new group for the update behavior. By adding each update behavior to all groups that represent it, SQUINT will not incur false negatives by failing to test an update behavior that is represented by two groups, only one of which is determined to be buggy (i.e., the representative of only one group creates crash-inconsistent states).

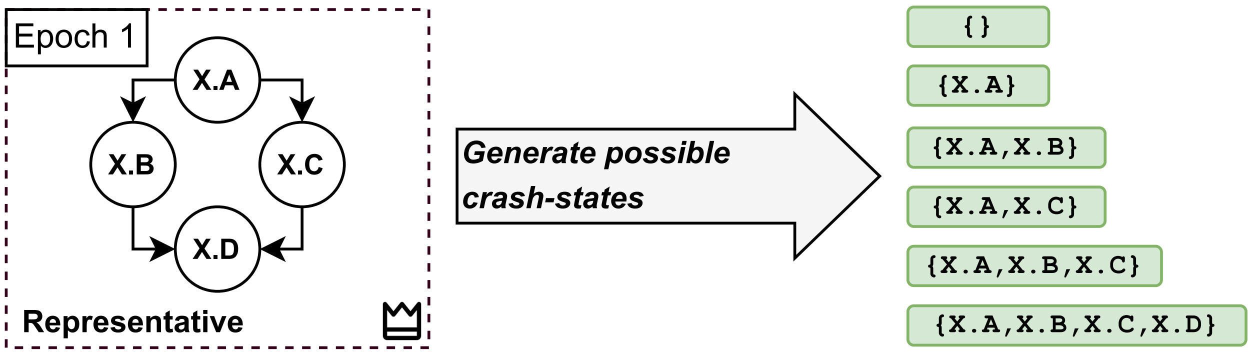


Figure 4.7: **Crash-Consistency Test Generation.** An example of how SQUINT would perform testing (Step E) of Epoch 1 (as found in Figure 4.6).

Figure 4.6 provides an example of how the epochs in Figure 4.5 are placed into a group. Since Epoch 1 is the largest (i.e., has the most nodes) and SQUINT traverses epochs in sorted order, Epoch 1 is the first to be placed into the group. Then, Epoch 2 is added to the group because Epoch 1 represents Epoch 2; Epoch 2 contains a subset of the nodes in Epoch 1 ($\{X.A, X.B, X.C\} \subset \{X.A, X.B, X.C, X.D\}$) and the ordering constraints in Epoch 2 are a superset of the ordering constraints among equivalent nodes in Epoch 1 (for the subset of nodes $\{X.A, X.B, X.C\}$, Epoch 1 has the same ordering constraints as Epoch 2). Lastly, Epoch 3 is added to the group; Epoch 3 also contains a subset of the nodes in Epoch 1 (as $\{X.D\} \subset \{X.A, X.B, X.C, X.D\}$) and the ordering constraints in Epoch 3 are the same as the ordering constraints among equivalent nodes in Epoch 1.

Epoch 1 is the representative of the group because only Epoch 1 represents all other members in the group. Note that not all group members represent one another; Epoch 2 does not represent Epoch 3 because Epoch 3 does not have a subset of the nodes in Epoch 2 ($\{X.D\} \not\subset \{X.A, X.B, X.C\}$). However, since Epoch 1 represents both Epochs 2 and 3 (i.e., Epoch 1 can create crash-states that are equivalent to all of the crash-states of Epochs 2 and 3), they are in the same group.

4.4.5 Model Checking (Step E)

Finally, SQUINT tests the representative update behavior for each group using a model checker (Step E). The model checker performs crash testing on each representative update behavior by generating all possible crash-states that can result from crashes that occur during the execution of the behavior (i.e., all possible permutations of updates that comply with the update ordering constraints in the behavior according to PM DPOR methods [57]) and testing each crash-state for crash consistency. For each crash-state, SQUINT runs the application in recovery mode and checks if the application detects an inconsistency in the persistent data; all applications we test are able to restart using an existing crash-state and can therefore be run in a recovery mode. For example, if an inconsistent state is detected, SQUINT's model checker expects the application to terminate with an error code or throw an exception (i.e, a fail-stop bug oracle), thus indicating that a crash-consistency bug has been found (§4.2.2).

If any of the crash-states generated by the update behavior cause a crash-consistency bug, SQUINT uses a delta-debugging inspired algorithm [168] to iteratively identify the most recent non-buggy update and isolate the bug location and produces a report for the user. If a representative update behavior contains a bug, SQUINT then tests the remaining update behaviors in the group that contain PM updates from source code locations that have yet to be tested. This follow-up testing of represented update behaviors explicitly enumerates all source code locations that contain the crash-consistency bugs that representative testing already identified, helping developers fix each instance of crash-consistency bugs in their application.

Figure 4.7 shows how SQUINT tests the representative update behavior from Figure 4.6. SQUINT passes the update behavior to the model checker along with the original PM operation trace (Step A, §4.4.1). The model checker replays and persists all updates before the representative update behavior, since these updates are required to produce a valid crash image (e.g., the updates in Epoch 1 are not reachable without executing the preceding trace). After reaching the store associated with the $X.A$ node, the model checker tests all possible orderings that the stores in Epoch 1 could be persisted, shown in Figure 4.7.

4.4.6 Limitations

False Negatives Since representative testing and the update behavior heuristic are unsound reduction techniques (§4.3), SQUINT will have false negatives. Specifically, SQUINT has false negatives when: (1) SQUINT's update behavior heuristic does not work well for a given application, (2) our representative testing observations do not apply well to the tested PM application; or (3) an application silently fails in the presence of a crash-consistency bug (e.g., it produces erroneous output instead of stopping). However, we experimentally find that SQUINT incurs fewer false negatives than prior work (§4.6.3).

PM Concurrency Bugs Like most prior PM testing tools, SQUINT targets single-threaded PM crash-consistency bugs and does not track update orderings across threads (due to the limitations of our implementation; §4.5). So, while SQUINT can and does test multi-threaded systems (e.g., memcached-pm [39], Redis-pm [33], and HSE [114]), and can find crash-consistency bugs within a single thread's execution (§4.6.1), it misses PM crash-consistency bugs that only arise due to multi-threading. In the future, SQUINT could incorporate thread interleavings into its update behavior inference (§4.4.3) as additional PM ordering operations. Representative testing is complementary to existing PM concurrency testing tools [22, 52], as representative testing could be used to reduce the crash-state space needed to test concurrent PM applications.

4.5 Implementation

We implement our prototype of SQUINT in ∼6500 LOC of C++ (as counted by SLOCCount [155]). SQUINT performs program analysis using LLVM-10 [91, 150]. We compile all of our test applications with WLLVM [136], a drop-in LLVM replacement, which simplifies the task of compiling large systems into executables with accompanying LLVM bitcode.

Since SQUINT analyzes programs after optimizations have been applied, SQUINT is capable of detecting PM compiler bugs (§4.2.2) in a tested program. For example, if a single store at the source-code level is transformed into multiple PM stores, our tracing tool will detect multiple stores, allowing SQUINT to perform testing and expose bugs that can occur within orderings of these stores.

SQUINT uses `pmemcheck` [31], a valgrind [123] tool for PM tracing, to generate PM traces. SQUINT traces PM applications as they operate on unit test program inputs—these inputs could also be driven by fuzzing [102, 107, 161] or symbolic execution [12, 122]). Since `pmemcheck` works on Intel's PMDK library and PMDK works for applications with or without access to PM hardware (e.g., PMDK can run on block file systems with `mmap`), SQUINT can be used to test a wide variety of PM applications with different PM implementations.

Toolkit Limitations Our implementation inherits three limitations from `pmemcheck`. First, `pmemcheck` only supports user-space applications, which prohibits SQUINT from testing PM file systems. Replacing `pmemcheck` with a kernel-tracing tool would allow SQUINT to add kernel support. Second, `pmemcheck` does not support tracing concurrency operations (§4.4.6). Third, `pmemcheck` is tailored to testing PM applications that use Intel's PMDK library for low-level persistence primitives (e.g., `PERSIST`) calls and PM file initialization. All but one of our test application's supported `pmemcheck` already, with HSE [114] requiring fewer than ten lines of source code modification in order to function with `pmemcheck`.

4.6 Evaluation

In this section, we evaluate the effectiveness and efficiency of SQUINT. First, we present an overview of the crash-consistency bugs found by SQUINT (§4.6.1) and discuss our interaction with developers in reporting the new bugs discovered by SQUINT. We then analyze SQUINT's scalability (§4.6.2) and compare SQUINT's results to prior works (§4.6.3).

Evaluation Targets We evaluate SQUINT with a variety of persistent data structures and real-world applications, all of which were tested by prior work with the exception of HSE [114]. We test 6 persistent data-structures provided by PMDK [32] and implemented using PMDK's persistent object allocation and transaction update API (libpmemobj [143]). We also evaluate 5 persistent data structures from RECIPE [49, 95, 151] as well as 5 PM key-value store indices that were tested by WITCHER [49, 50]. Finally, we use SQUINT to test 3 server applications: we test memcached-pm [39], a PM port of memcached (a popular memory caching service [146]) developed by Lenovo; a PM port of Redis [33] (another popular memory caching service [14]) ported by Intel; and HSE (Heterogeneous-Memory Storage Engine) [114], a key-value storage engine developed by Micron.

Evaluation Setup We evaluate our test targets using existing inputs (e.g., unit tests and coverage tests), similar to prior work [50, 51, 104, 122]. We test targets from WITCHER using the exact random inputs provided in WITCHER's artifact [49]. We run our experiments on a server with a Intel Xeon Gold 6230 CPU (2.10 GHz), four 128 GB Intel Optane Series 100 Pmem DIMMs, and 256 GB of DDR4 DRAM (2667 MHz).

4.6.1 Bugs Detected by SQUINT

We present the results of SQUINT's testing in Table 4.1. SQUINT found 108 bugs across the 19 systems. In our testing, we find 53 new bugs: 6 in PMDK's *Array* data structure, 26 in the persistent key-value indices, 17 in RECIPE data structures, 2 in memcached-pm, 1 in Redis-pm, and 1 in HSE. With the exception of HSE, all of these systems were previously tested by prior work [50, 51, 102, 103, 122], yet these new bugs were not found by any prior tool. We reported the 53 new bugs to their project maintainers. Of these 53 bugs, 49 have been confirmed by the project maintainers so far: all except for the 2 memcached-pm bugs and 1 Redis-pm bugs.

4.6.2 Scalability of Representative Testing

We evaluate the scalability of representative testing by comparing how quickly it finds bugs to Jaaru [57], the state-of-the-art PM model checker that leverages DPOR crash-state reduction techniques. In the original artifact, Jaaru is configured to skip library and data structure initialization—however, we find some early crash-consistency bugs that occur during the initialization of data structures that can cause the first few entries in a key-value store to be lost on a crash. Therefore, we evaluate Jaaru both including and excluding initialization code ("Jaaru" and "Jaaru-NoInit",

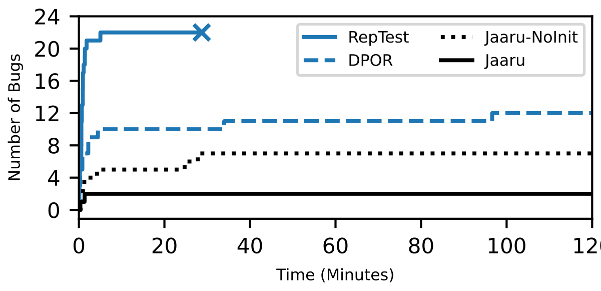
Category	Application	Total Bugs	New Bugs
PMDK Data Structures	Array	7	6
	BTree	6	0
	CTree	1	0
	Hashmap (Atomic)	4	0
	Hashmap (TX)	2	0
	RBTree	2	0
Key-Value Indices	CCEH	4	2
	Fast Fair	7	3
	Level Hash	34	17
	WOART	3	2
	WORT	2	2
RECIPE Indices	P-ART	4	2
	P-BwTree	3	2
	P-CLHT	13	7
	P-HOT	7	5
	P-Masstree	3	1
Server Applications	HSE	2	1
	Memcached	3	2
	Redis	1	1
Total		**108**	**53**

Table 4.1: **SQUINT's Testing Results.** SQUINT finds 108 bugs (53 new); 52 of the new bugs are in systems tested by prior work.

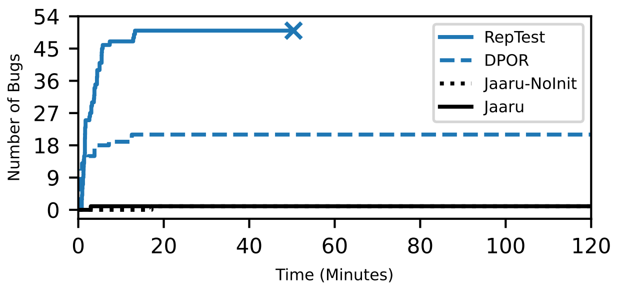
respectively). Furthermore, due to compatibility issues of running Jaaru on newer versions of PMDK, we also implemented Jaaru's DPOR algorithm ("DPOR") in SQUINT's model checker to more clearly compare DPOR and representative testing. We use SQUINT and the three baseline testing strategies on each of our testing targets and count the number of bugs found by each approach over time. We set a maximum time limit of 2 hours for all application categories.

Figure 4.8 shows the bugs found over time for three of the four categories of testing targets: PMDK Data Structures (Figure 4.8a), Key-Value Indices (Figure 4.8b), and RECIPE Indices (Figure 4.8c). We omit comparison against the server benchmarks because we were unable to run Jaaru on these systems; furthermore, the 6 bugs occur early in these application's executions, so both "RepTest" and "DPOR" configurations of SQUINT find all 6 bugs within 11 minutes. Each graph sums the results of the individual tests (e.g., Figure 4.8a shows that SQUINT finds 22 bugs after crash-testing each of the PMDK indices in parallel for $\sim$28 minutes).

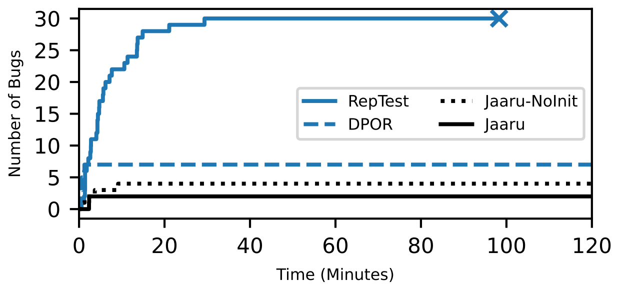
Representative testing outperforms DPOR approaches in all benchmarks. SQUINT finds all of the reported bugs before the time limit (28 minutes for PMDK data structures, 50 for Key-Value indices, and 98 minutes for RECIPE indices). In contrast, none of the baselines finds all



(a) PMDK Data Structures



(b) Key-Value Indices



(c) RECIPE Indices

Figure 4.8: **SQUINT versus DPOR Baselines.** A comparison of SQUINT ("RepTest") to three baselines: Jaaru [57], configured to test the entire application ("Jaaru") and to skip program initialization ("Jaaru-NoInit"); and Jaaru's algorithm implemented in SQUINT's model checker ("DPOR"). The $\times$ marker indicates the completion of testing if it occurred before the time limit, which only occurred for SQUINT.

of the bugs within the time limit: The DPOR baseline outperforms both Jaaru configurations, but finds roughly half of the bugs found by representative testing. Jaaru performs even worse, the Jaaru-NoInit and Jarru baselines find 12% and 5% of the bugs found by representative testing, respectively. Note, given an infinite testing budget, the baselines would find all of the bugs found by SQUINT; however, SQUINT allows developers to scale PM crash-consistency testing to large applications and find crash-consistency bugs quickly.

4.6.3 Coverage Comparison to Pattern-Based Approaches

We evaluate the coverage of representative testing by comparing the number of PM crash-consistency bugs found by SQUINT to those found using pattern-based tools. For each prior tool, we sum the number of crash-consistency bugs found in each application that was tested by both the tool and by SQUINT, ensuring to test the same version of each application with SQUINT as was tested with the prior tool. We do not compare with PMFuzz [102], which does not report the number of crash-consistency bugs it found, nor Yashme [58], which uses different compiler and

Application	**Version(s)**	**Also Tested By**
Array	v1.4, v1.8	PMDebugger, WITCHER
BTree	v1.4	AGAMOTTO, PMTest, XFDetector, WITCHER
BTree	v1.8	Jaaru, WITCHER
CTree	v1.4	PMTest, XFDetector, WITCHER
CTree	v1.8	Jaaru, WITCHER
Hashmap (Atomic)	v1.4	PMTest, WITCHER
Hashmap (Atomic)	v1.8	Jaaru, PMDebugger, WITCHER
Hashmap (TX)	v1.4	AGAMOTTO, PMTest, XFDetector, WITCHER
Hashmap (TX)	v1.8	Jaaru, WITCHER
RBTree	v0.3	AGAMOTTO, PMTest, XFDetector, WITCHER
RBTree	v1.8	Jaaru, WITCHER
WOART, WORT, Fast Fair, CCEH, Level Hash, P-ART, P-BwTree, P-CLHT, P-HOT, P-Masstree	ad69038	WITCHER
P-CLHT	fc508dd	AGAMOTTO, WITCHER
Memcached	8f121f6	AGAMOTTO, PMDebugger, XFDetector, WITCHER
Redis	3.2-nvml	AGAMOTTO, PMDebugger, XFDetector, WITCHER
HSE	v2.1	-

Table 4.2: **Tested Application Versions.** The versions of applications we tested and the prior works which also test those versions. All applications were tested using PMDK 1.8.

benchmark versions. Finally, we do not compare to testing tools that only detect PM concurrency bugs (PMRace [22], DURINN [52]), as SQUINT only detects single-threaded bugs (§4.4.6).

SQUINT finds between 7 and 42 more crash-consistency bugs than five pattern-based prior works. Specifically, SQUINT finds 7 more crash-consistency bugs than AGAMOTTO, 7 more than PMDebugger, 10 more than PMTest, 42 more than WITCHER, and 9 more than XFDetector. Furthermore, SQUINT finds all the crash-consistency bugs reported in prior works with the exception of two bugs: SQUINT fails to detect two bugs from WITCHER, Bug #28 [47] and Bug #36 [48], which WITCHER found in BwTree and P-HOT, respectively. Both of these bugs cause the recovery code to leak memory, but do not cause the recovery code to crash. Ergo, SQUINT tests the erroneous PM updates but does not identify them as failures.

Overall, we find that representative testing provides higher scalability than existing DPOR approaches while providing high coverage relative to pattern-based approaches.

4.7 Discussion

eADR-enabled PM Intel Optane Pmem 200 Series [36] (an updated version of the 100 Series [34]) includes a new feature called Extended Asynchronous DRAM Refresh (eADR). eADR ensures that stores to PM are persisted after executing, eliminating the need for explicit cache-line flushes and eliminating missing-flush bugs (§4.2.2). PM applications must still issue memory fences on eADR platforms to force weakly-ordered to become persistent (e.g., non-temporal stores on x86-64) [137], and memory fences are also required for preventing stores from being reordered on architectures that do not provide total-store-ordering (TSO). Ergo, while eADR platforms eliminate missing-flush bugs, they are still susceptible to missing-fence bugs, compiler bugs [58], and application-specific bugs (§4.2.2) Overall, PM crash-consistency testing tools are still needed, as the crash-state space of PM applications is still large even for eADR-enabled PM.

Test Case Generation SQUINT can only detect a crash-consistency bug if the PM operation trace (Step A, §4.4.1) contains an update behavior that evinces the bug. SQUINT currently uses unit tests, example inputs, and randomly generated inputs [50] to drive its PM testing and analysis. Adopting test generation approaches (e.g., fuzzing [102, 107, 161], symbolic execution [12, 122]) could increase the number and size of the PM operation traces for SQUINT to analyze. This would increase the number of update behaviors that SQUINT could test and could potentially reduce false negatives.

4.8 Related Work

Persistent Memory Programming Libraries Rather than creating custom implementations of common durability techniques (e.g., undo logging), many PM developers use PM programming libraries in their PM applications [25, 32, 112, 143, 154, 163, 169]. While small PM libraries can be exhaustively tested [57], crash-consistency bugs can arise from API misuse or through sequences of API calls which may not be tested in library-only model checking. Since SQUINT effectively prunes the testing space for PM applications, SQUINT can find crash-consistency bugs in application code, programming library code, and in the interface between the two.

Persistent Memory Language Extensions One method of implementing the PM abstraction is to incorporate PM concepts directly into programming language design [40, 65, 145, 158]. PM language extensions allow developers to directly define and operate on PM objects without having to insert ordering instructions (§4.2.1), as they are automatically inserted by the compiler [40, 158]. However, these language extensions cannot automatically prevent crash-consistency bugs, as they are unaware of application-specific ordering requirements. Ultimately, PM language extensions do not eliminate the need for PM crash-consistency testing.

File-System Crash-Consistency Testing Many prior works explore crash-consistency testing in file systems [20, 45, 46, 59, 78, 85, 110, 116, 131, 166, 167]. These approaches predominately leverage either bounded pruning [110, 116] or application/bug-specific customization [20, 59, 78, 84, 85, 162, 166, 167]. Even though crash-consistency testing is well explored for file systems, crash-consistency testing for PM applications requires special attention, as PM applications face exponentially larger testing spaces due to the finer granularity of PM updates.

Copy-paste bugs Copy-paste bugs are bugs caused when developers incorrectly copy-and-paste code, which can introduce bugs when the duplicated code does not operate correctly outside of its original context [99]. The key insight and observations behind representative testing, however, do not rely on code duplication in update behaviors to generate equivalent crash-states. Overall, copy-paste bugs are orthogonal to PM crash-consistency bugs. In our evaluation, we do not find copy-paste errors as the cause of any of the PM crash-consistency bugs found by SQUINT.

4.9 Conclusion

Prior Persistent Memory (PM) crash-consistency testing tools fail to adequately help developers overcome the challenges of writing crash-consistent PM applications, as they either fail to scale to real-world applications or incur false negatives by extrapolating from application- or bug-specific patterns. To overcome these challenges, we developed *representative testing*, a new PM crash-state reduction strategy that considers crash-states equivalent if they produce the same crash-consistency bugs. We subsequently implemented the *update behavior heuristic* to approximate and test a small set of representative crash-states that evince a representative set of crash-consistency bugs in the application. We implemented representative testing and the update behavior heuristic in SQUINT and used it to find 108 bugs (53 new) across 19 PM applications, demonstrating the coverage of representative testing over prior approaches. We also showed that SQUINT is scalable, as it finds 8.5$\times$ as many bugs as the state-of-the-art DPOR model-checking tool. We conclude that representative testing is a scalable and accurate testing approach for PM applications.

CHAPTER 5

Conclusion and Future Work

In this dissertation, we proposed our novel insights on how to design automated bug detection and correction tools that are scalable, thorough, and easy for developers to use. We observed that by approximating the reasoning performed by developers during the development life cycle of their applications, we were able to build automated tools and techniques that find and fix bugs without forcing developers to chose between scalability and coverage.

We applied this observation to the area of modern PM application development, which provides exciting opportunities for developers looking to design low-latency storage systems, but comes with many pitfalls in the difficulty of designing simultaneously efficient and correct systems. We first presented AGAMOTTO and HIPPOCRATES, which we designed to help developers find and fix platform-specific PM bugs by prioritizing the exploration of execution paths most prone to these bugs and then automatically fixing these bugs with provably-correct and heuristically-optimized fixes. We then presented SQUINT, a tool which prunes the exploration of crash-states in crash-consistency testing by automatically uncovering semantic similarities in bugs that would be exposed in different crash-states in order to scalably and thoroughly uncover application-specific PM ordering bugs. These tools have been used to find and fix over two hundred PM bugs across a wide variety of modern PM applications, systems, and programming libraries, and thus demonstrate the efficacy of our methodology in designing automated tools to help developers write correct and efficient systems.

In the rest of this section, we describe some future work directions that could leverage the insights provided in this dissertation to explore bug detection and correction in non-PM applications. We first describe how our insights could drive further detection and correction of platform-specific bugs in non-PM platforms. We wrap up by describing how our insights could advance the state-of-the-art in state-space reduction policies for model-checking approaches in other applications domains.

5.1 Automatically Finding and Fixing Platform-Specific Bugs

In our work on AGAMOTTO (Chapter 2) and HIPPOCRATES (Chapter 3), we demonstrated the efficacy of building tools to target platform-specific (and application-independent) bugs in modern PM systems. We further demonstrated how needed such tools are to uncover developer misconceptions and errors when developing applications for new platforms based on the results from our evaluations of AGAMOTTO and HIPPOCRATES (§2.6 and §3.6, respectively). We argue that this approach applies broadly, as emerging hardware platforms (e.g., new CPUs with new memory models, hardware-accelerated networking) and software platforms (e.g., blockchain programming, cloud computing platforms, domain-specific APIs) can require platform-specific operations to be performed for applications to function properly, and platform-specific debugging tools would help developers identify and correct platform misuses before their applications are deployed. As the diversity of programming platforms continues to expand, we believe that this approach to finding and fixing platform-specific bugs will encourage the adoption of new technologies and increase trust in the correctness of early systems built for these new technologies.

5.2 Automating Semantic State-Space Reduction Policies

In our work on SQUINT (Chapter 4), we demonstrated the efficacy of accurately reducing the number of program states explored by leveraging automatically-discovered application semantics, resulting in both accurate and efficient crash-consistency testing for modern PM applications. As we discussed previously (§4.3), somewhat similar approaches have been taken in model-checking works targeting concurrent systems [117, 133] and distributed systems [96]. However, these works generally either enforce global exploration bounds that do not sufficiently leverage application semantics or require developer input in the form of specifications or annotations in order to leverage more application-specific information, which increases the burden on developers. We believe that building automatic techniques for uncovering application semantics, akin to how SQUINT automatically discovers update behaviors (§4.4), can be leveraged to incorporate more semantic information into future model-checking tools while simultaneously increasing the ease of use of these tools. We further believe that advances in Artificial Intelligence (AI)-powered program analysis will enable tool developers to at least partially generate algorithms for pattern discovery; e.g., assist in the generation of techniques like SQUINT's persistence graph splitting procedure (§4.4).

BIBLIOGRAPHY

[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] H. Akinaga and H. Shima. Resistive Random Access Memory (ReRAM) Based on Metal Oxides. *Proceedings of the IEEE*, 98(12):2237–2251, Dec 2010.

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

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

[6] Dmytro Apalkov, Alexey Khvalkovskiy, Steven Watts, Vladimir Nikitin, Xueti Tang, Daniel Lottis, Kiseok Moon, Xiao Luo, Eugene Chen, Adrian Ong, Alexander Driskill-Smith, and Mohamad Krounbi. Spin-transfer Torque Magnetic Random Access Memory (STT-MRAM). *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 9(2):13:1–13:35, May 2013.

[7] 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.

[8] 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.

[9] Malcolm P. Atkinson and Ronald Morrison. Orthogonally Persistent Object Systems. *VLDB J.*, 4(3):319–401, 1995.

[10] A. Bensoussan, C. T. Clingen, and Robert C. Daley. The multics virtual memory: Concepts and design. *Commun. ACM*, 15(5):308–318, 1972.

[11] Bill Bridge. NVM-Direct library. https://github.com/oracle/nvm-direct, 2015.

[12] 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.

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

[14] Josiah L Carlson. *Redis in action*. Manning Publications Co., Shelter Island, NY, 2013.

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

[16] 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.

[17] R.N. Charette. Why software fails [software failure]. *IEEE Spectrum*, 42(9):42–49, 2005.

[18] 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)*, pages 1–7, Savannah, GA, November 2016. USENIX Association.

[19] Cheng Chen, Jun Yang, Qingsong Wei, Chundong Wang, and Mingdi Xue. Fine-grained metadata journaling on NVM. In *2016 32nd Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–13. IEEE, IEEE, 2016.

[20] 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, Monterey, CA, USA, oct 2015. ACM.

[21] Jia Chen. Andersen's pointer analysis. https://github.com/grievejia/andersen, 2019.

[22] Zhangyu Chen, Yu Hua, Yongle Zhang, and Luochangqi Ding. Efficiently detecting concurrency bugs in persistent memory programs. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 873–887, Lausanne, Switzerland, 2022. ACM.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] CXL Consortium. Compute Express Link™: The Breakthrough CPU-to-Device Interconnect, 2022.

[28] Brian Cooper. YCSB. https://github.com/brianfrankcooper/YCSB, 2019.

[29] Brian Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

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

[31] Intel Corporation. An introduction to pmemcheck. https://pmem.io/2015/07/17/pmemcheck-basic.html, 2015.

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

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

[34] Intel Corporation. Revolutionary memory technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html, 2018.

[35] Intel Corporation. Intel Optane Persistent Memory 100 Series Product Brief. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-dc-persistent-memory-brief.html, 2019.

[36] Intel Corporation. Intel Optane Persistent Memory 200 Series Product Brief. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-persistent-memory-200-series-brief.html, 2020.

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

[38] Intel Corporation. Intel Optane Persistent Memory 300 Series Product Brief. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/300-series-brief.html, 2022.

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

[40] 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, Kyoto, Japan, 2016. ACM.

[41] Bang Di, Jiawen Liu, Hao Chen, and Dong Li. Fast, flexible, and comprehensive bug detection for persistent memory programs. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 503–516, Virtual Event, USA, April 2021. ACM.

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

[43] 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.

[44] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. *ACM Sigplan Notices*, 40(1):110–121, 2005.

[45] 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.

[46] 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.

[47] Xinwei Fu. P-BwTree, Bug #28. https://github.com/cosmoss-vt/witcher/blob/sosp21-ae/bugs/sosp21-correctness-bugs.md#bug-28, 2021.

[48] Xinwei Fu. P-HOT, Bug #36. https://github.com/cosmoss-vt/witcher/blob/sosp21-ae/bugs/sosp21-correctness-bugs.md#bug-36, 2021.

[49] Xinwei Fu. SOSP'21 Witcher Artifact. https://github.com/cosmoss-vt/witcher, 2021.

[50] Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny Wadkar, Dongyoon Lee, and Changwoo Min. Witcher: Systematic Crash Consistency Testing for Non-Volatile Memory Key-Value Stores. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principle*, pages 100–115, Virtual Event / Koblenz, Germany, oct 2021. ACM.

[51] Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny Wadkar, Changwoo Min, and Dongyoon Lee. WITCHER: Detecting Crash Consistency Bugs in Non-volatile Memory Programs. *CoRR*, abs/2012.06086, 2020.

[52] Xinwei Fu, Dongyoon Lee, and Changwoo Min. DURINN: Adversarial Memory and Thread Interleaving for Detecting Durable Linearizability Bugs. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design*

and Implementation, OSDI 2022, pages 195–211, Carlsbad, CA, USA, jul 2022. USENIX Association.

[53] Kaan Genç, Michael D. Bond, and Guoqing Harry Xu. Crafty: Efficient, HTM-Compatible Persistent Transactions, 2020.

[54] Patrice Godefroid. Model Checking for Programming Languages using Verisoft. In Peter Lee, Fritz Henglein, and Neil D. Jones, editors, *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium*, pages 174–186, Paris, France, jan 1997. ACM Press.

[55] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M Chen, and Thomas F Wenisch. Persistency for synchronization-free regions. *ACM SIGPLAN Notices*, 53(4):46–61, 2018.

[56] Hamed Gorjiara, Weiyu Luo, Alex Lee, Guoqing Harry Xu, and Brian Demsky. Checking robustness to weak persistency models. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 490–505, 2022.

[57] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. Jaaru: efficiently model checking persistent memory programs. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 415–428, Virtual Event, apr 2021. ACM.

[58] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. Yashme: detecting persistency races. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 830–845, Lausanne, Switzerland, 2022. ACM.

[59] Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. Storage Systems are Distributed Systems (So Verify Them That Way!). In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 99–115, Virtual Event, November 2020. USENIX Association.

[60] Saemundur O Haraldsson, John R Woodward, Alexander EI Brownlee, and Kristin Siggeirsdottir. Fixing bugs in your sleep: how genetic improvement became an overnight success. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1513–1520, 2017.

[61] 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.

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

[63] 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.

[64] 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.

[65] Morteza Hoseinzadeh and Steven Swanson. Corundum: statically-enforced persistent memory safety. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 429–442, Virtual Event, USA, apr 2021. ACM.

[66] 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.

[67] Hanxian Huang, Zixuan Wang, Juno Kim, Steven Swanson, and Jishen Zhao. Ayudante: A Deep Reinforcement Learning Approach to Assist Persistent Memory Programming. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 789–804. USENIX Association, July 2021.

[68] Intel. Intel Optane DC Persistent Memory Partner: Google Cloud. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/google-partner-video.html, 2018.

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

[70] Intel. Old issues repo for PMDK. https://github.com/pmem/issues/issues, 2019.

[71] Intel. Intel Optane Persistent Memory Workload Solutions. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-persistent-memory-solutions.html, 2020.

[72] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, volume 2, chapter 4, pages 1198,1787. Intel Corporation, 2020.

[73] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, volume 2, chapter 3, pages 739–742,748–749. Intel Corporation, 2020.

[74] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, volume 1, chapter 16, pages 391–398. Intel Corporation, 2020.

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

[76] Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. Brief announcement: Preserving happens-before in persistent memory. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 157–159, 2016.

[77] 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.

[78] Yanyan Jiang, Haicheng Chen, Feng Qin, Chang Xu, Xiaoxing Ma, and Jian Lu. Crash Consistency Validation Made Easy. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 133–143, New York, NY, USA, 2016. Association for Computing Machinery.

[79] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 389–400, 2011.

[80] 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.

[81] LLVM-CBE: Resurrected LLVM "C Backend", with improvements. https://github.com/JuliaComputing/llvm-cbe, 2020.

[82] 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.

[83] 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.

[84] Gabriele Keller, Toby C. Murray, Sidney Amani, Liam O'Connor, Zilin Chen, Leonid Ryzhyk, Gerwin Klein, and Gernot Heiser. File systems deserve verification too! *ACM SIGOPS Oper. Syst. Rev.*, 48(1):58–64, 2014.

[85] 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 2019*, pages 147–161, Huntsville, ON, Canada, October 2019. ACM.

[86] 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.

[87] Herb Krasner. The cost of poor software quality in the US: A 2020 report. *Consortium for Information and Softwware Quality (CISQ)*, 2021.

[88] 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.

[89] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*, pages 179–196. Association for Computing Machinery, New York, NY, United States, 2019.

[90] 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.

[91] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, Palo Alto, CA, mar 2004. IEEE.

[92] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 3–13. IEEE, 2012.

[93] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1):54–72, 2011.

[94] E. Lee, H. Bahn, S. Yoo, and S. H. Noh. Empirical Study of NVM Storage: An Operating System's Perspective and Implications. In *2014 IEEE 22nd International Symposium on Modelling, Analysis Simulation of Computer and Telecommunication Systems*, pages 405–410, Sep. 2014.

[95] 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 2019*, pages 462–477, Huntsville, ON, Canada, October 2019. ACM.

[96] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. SAMC: Semantic-Aware model checking for fast discovery of deep bugs in cloud systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 399–414, Broomfield, CO, October 2014. USENIX Association.

[97] Weronika Lewandowska. Pmreorder basics. https://pmem.io/2019/02/04/pmreorder-basics.html, feb 2015.

[98] 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.

[99] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *IEEE Trans. Software Eng.*, 32(3):176–192, 2006.

[100] 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.

[101] 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.

[102] Sihang Liu, Suyash Mahar, Baishakhi Ray, and Samira Manabi Khan. PMFuzz: test case generation for persistent memory programs. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 487–502, Virtual Event, apr 2021. ACM.

[103] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas F. Wenisch, Aasheesh Kolli, and Samira Manabi Khan. Cross-Failure Bug Detection in Persistent Memory Programs. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems*, pages 1187–1202, Lausanne, Switzerland, March 16-20, 2020, 2020. ACM.

[104] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Manabi 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, ASPLOS 2019*, pages 411–425, Providence, RI, USA, April 13-17, 2019, 2019. ACM.

[105] 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.

[106] Pratyush Mahapatra, Mark D. Hill, and Michael M. Swift. Don't persist all : Efficient persistent data structures. *CoRR*, abs/1905.13011, 2019.

[107] Dominik Maier, Heiko Eißfeldt, Andrea Fioraldi, and Marc Heuse. AFL++ : Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies, WOOT 2020*, Virtual Event, aug 2020. USENIX Association.

[108] 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.

[109] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. Sapfix: Automated end-to-end repair at scale. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 269–278. IEEE, 2019.

[110] 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*, Santa Clara, CA, 2017. USENIX Association.

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

[112] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic In-place Updates for Non-volatile Main Memories with Kamino-Tx. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017*, pages 499–512, Belgrade, Serbia, apr 2017. ACM.

[113] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. Taming Google-scale continuous testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 233–242. IEEE, 2017.

[114] Micron. Heterogeneous-Memory Storage Engine. https://www.micron.com/products/advanced-solutions/heterogeneous-memory-storage-engine, 2022.

[115] 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.

[116] 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 2018*, pages 33–50, Carlsbad, CA, USA, October 8-10, 2018, oct 2018. USENIX Association.

[117] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. *ACM Sigplan Notices*, 42(6):446–455, 2007.

[118] 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.

[119] 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.

[120] Ian Neal. OSDI'20 AGAMOTTO Artifact. https://github.com/efeslab/agamotto, 2020.

[121] Ian Neal, Andrew Quinn, and Baris Kasikci. HIPPOCRATES: healing persistent memory bugs without doing any harm. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 401–414, Virtual Event, USA, apr 2021. ACM.

[122] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. AGAMOTTO: How Persistent is your Persistent Memory Application? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1047–1064, Virtual Event, November 2020. USENIX Association.

[123] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[124] 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.

[125] National Library of Medicine. The Hippocratic Oath. https://www.nlm.nih.gov/hmd/greek/greek_oath.html, 2012.

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

[127] Soyeon Park, Shan Lu, and Yuanyuan Zhou. 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*, pages 25–36, 2009.

[128] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014*, pages 265–276, Minneapolis, MN, USA, jun 2014. IEEE Computer Society.

[129] 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.

[130] Justyna Petke, Saemundur O Haraldsson, Mark Harman, William B Langdon, David R White, and John R Woodward. Genetic improvement of software: a comprehensive survey. *IEEE Transactions on Evolutionary Computation*, 22(3):415–432, 2017.

[131] Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Application Crash Consistency and Performance with CCFS. *ACM Trans. Storage*, 13(3), September 2017.

[132] 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.

[133] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In *TACAS*, volume 5, pages 93–107. Springer, 2005.

[134] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. Persistency semantics of the Intel-x86 architecture. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–31, 2019.

[135] 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.

[136] Tristan Ravitch. Whole Program LLVM. https://github.com/travitch/whole-program-llvm, 2020.

[137] Andy Rudoff. Questions about eADR, sfence and TSX. https://groups.google.com/g/pmem/c/_DJCFGylfVE/m/L0oyltg8BAAJ, 2020.

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

[139] Seemanta Saha, Ripon K. Saha, and Mukul R. Prasad. Harnessing Evolution for Multi-Hunk Program Repair. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 13–24. IEEE Press, 2019.

[140] SAP. SAP HANA & Persistent Memory. https://blogs.sap.com/2018/12/03/sap-hana-persistent-memory/, 2018.

[141] 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.

[142] Steve Scargall. Debugging Persistent Memory Applications. In *Programming Persistent Memory*, pages 207–260. Springer, 2020.

[143] Steve Scargall. libpmemobj: A Native Transactional Object Store. In *Programming Persistent Memory*, pages 81–109. Springer, Santa Clara, CA, USA, 2020.

[144] Jonathan S. Shapiro and Jonathan Adams. Design evolution of the EROS single-level store. In Carla Schlatter Ellis, editor, *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 59–72, Monterey, California, USA, 2002. USENIX.

[145] Thomas Shull, Jian Huang, and Josep Torrellas. Autopersist: an easy-to-use java NVM framework based on reachability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019*, pages 316–332, Phoenix, AZ, USA, 2019. ACM.

[146] Ahmed Soliman. *Getting Started with Memcached*. Packt Publishing Ltd, Birmingham, UK, 2013.

[147] Steven Swanson. Early Measurements of Intel's 3DXPoint Persistent Memory DIMMs, Apr 2019.

[148] Satish M. Thatte. Persistent Memory: A Storage Architecture for Object-Oriented Database Systems. In Klaus R. Dittrich and Umeshwar Dayal, editors, *1986 International Workshop on Object-Oriented Database Systems*, pages 148–159, Asilomar Conference Center, Pacific Grove, California, USA, Proceedings, sept 1986. IEEE Computer Society.

[149] The LLVM Project. LLVM 8 documentation. https://releases.llvm.org/8.0.0/docs/index.html, 2019.

[150] The LLVM Project. LLVM 10 documentation. https://releases.llvm.org/10.0.0/docs/index.html, 2020.

[151] UT Systems and Storage Lab. RECIPE: high-performance, concurrent indexes for persistent memory (SOSP 2019). https://github.com/utsaslab/RECIPE/tree/pmdk, 2019.

[152] 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.

[153] 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.

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

[155] David Wheeler. SLOCCount. http://www.dwheeler.com/sloccount/, 2001.

[156] Marvel Cinematic Universe Wiki. Eye of Agamotto. https://marvelcinematicuniverse.fandom.com/wiki/Eye_of_Agamotto.

[157] H-S Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E Goodson. Phase Change Memory. *Proceedings of the IEEE*, 98(12):2201–2227, Dec 2010.

[158] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan. Espresso: Brewing Java For More Non-Volatility with Non-volatile Memory. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018*, pages 70–83, Williamsburg, VA, USA, 2018. ACM.

[159] 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.

[160] 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.

[161] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2313–2328, Dallas, Texas, USA, 2017. ACM.

[162] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing File Systems via Two-Dimensional Input Space Exploration. In *2019 IEEE Symposium on Security and Privacy, SP 2019*, pages 818–834, San Francisco, CA, USA, May 2019. IEEE.

[163] Yi Xu, Joseph Izraelevitz, and Steven Swanson. Clobber-NVM: log less, re-execute more. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 346–359, Virtual Event, USA, apr 2021. ACM.

[164] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, USA, 2020. USENIX.

[165] 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.

[166] Junfeng Yang, Can Sar, and Dawson R. Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 131–146, Seattle, WA, USA, 2006. USENIX Association.

[167] Junfeng Yang, Paul Twohey, Dawson R. Engler, and Madanlal Musuvathi. Using Model Checking to Find Serious File System Errors. In *6th Symposium on Operating System Design and Implementation (OSDI 2004)*, pages 273–288, San Francisco, California, USA, December 2004. USENIX Association.

[168] Andreas Zeller and Ralf Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Software Eng.*, 28(2):183–200, 2002.

[169] Lu Zhang and Steven Swanson. Pangolin: A fault-tolerant persistent memory programming library. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 897–912, Renton, WA, USA, 2019. USENIX.

[170] Yiying Zhang and Steven Swanson. A study of application performance with non-volatile main memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, May 2015.

[171] Michael Zhivich and Robert K. Cunningham. The real cost of software errors. *IEEE Secur. Priv.*, 7(2):87–90, 2009.

[172] Pengfei Zuo, Yu Hua, and Jie Wu. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 461–476, Carlsbad, CA, USA, 2018. USENIX.