HIPPOCRATES: Healing Persistent Memory Bugs without Doing Any Harm

Ian Neal
iangneal@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

Andrew Quinn
arquinn@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

Baris Kasikci
barisk@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

ABSTRACT

Persistent memory (PM) technologies aim to revolutionize storage systems, providing persistent storage at near-DRAM speeds. Alas, programming PM systems is error-prone, as the misuse or omission of the durability mechanisms (i.e., cache 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.

CCS CONCEPTS

• **Hardware → Memory and dense storage**; • **Software and its engineering → Formal software verification**; *Automatic programming*.

KEYWORDS

persistent memory, program repair

ACM Reference Format:
Ian Neal, Andrew Quinn, and Baris Kasikci. 2021. HIPPOCRATES: Healing Persistent Memory Bugs without Doing Any Harm. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3445814.3446694

ASPLOS '21, April 19–23, 2021, Virtual, USA

ACM ISBN 978-1-4503-8317-2/21/04…$15.00
https://doi.org/10.1145/3445814.3446694

1 INTRODUCTION

Persistent memory (PM) technologies aim to revolutionize the storage-memory hierarchy [49, 53]. PM technologies such as Intel Optane DC [16, 24] are roughly 8× less expensive than DRAM [1] and offer disk-like durability with access latencies that are only 2–3× higher than DRAM latencies [32, 41, 60, 68]. PM can be accessed using the conventional load and store instructions and thus offers persistence without needing heavyweight file-system operations. Since becoming commercially available, popular applications (memcached [17] and Redis [15]) and companies (e.g., VMware and Oracle [26]) have begun using PM.

Alas, programming PM systems is error-prone [7, 11, 22, 45, 46, 48, 49, 59, 63, 66, 67]. Updates to PM are 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 [43, 44, 50, 51, 55]. Some tools use developer annotations and existing test suites to find bugs in arbitrary PM programs (e.g., PMTest [44], XFDetector [43], and Persistency Inspector [51]), while others find bugs in specific PM applications and frameworks (e.g., Yat [37] for PMFS [Persistent Memory File System] [19] and pmemcheck [55] for applications using PMDK [Persistent Memory Development Kit] [14]). AGAMOTTO [50] 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 paper, 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 tradeoff 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 (Agamotto [50] found 84 new bugs in only 5 PM applications and libraries) and time consuming to fix. Automated bug fixing tools are increasingly being deployed in industry (e.g., at Janus Rehabilitation [21] and Facebook [47]). Existing general purpose program repair tools use heuristics and/or tests suites to modify programs [39, 40, 58]. 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 [34]), 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 (§4).

Based on our insights, we develop Hippocrates, an automated PM bug fixing tool guaranteed to "do no harm." 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 which 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 [14], P-CLHT (from RECIPE [42]), and memcached-pm [17]. 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 [15], 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 YCSB workloads).

Overall, we make the following contributions:

- 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 bugs we reproduce 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 paper, we provide background on PM programming and discuss the challenges of automatic PM bug fixing (§2); we then discuss our analysis of PMDK bugs and our insights (§3); we describe the design of Hippocrates's automated fixes and sketch proofs of their correctness (§4); we discuss details of Hippocrates's implementation (§5); we evaluate the effectiveness and performance of Hippocrates (§6); finally, we discuss related work (§8) and conclude (§9).

2 BACKGROUND AND CHALLENGES

2.1 Persistent Memory Programming

In order to take advantage of byte-addressable PM, developers must modify existing programs to use user-level persistence mechanisms. 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) [27, 28]. ARM provides similar instructions with similar semantics (e.g., flush `DC CVAP` and fence `DSB` [3, 56]). 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.

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 persistency bugs (e.g., performance bugs resulting from overuse of cache flush instructions), recent studies show that PM durability bugs are numerous [50] 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 §7.

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 [44], binary instrumentation [43, 51, 55], or a symbolic memory model [50]; see §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.

2.3 Automated PM Bug Fixing Challenges

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 [21, 47]. Many of these systems are targeted at general-purpose program repair and aim to fix any class of bugs by using heuristics [39, 40, 58] (see §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, resulting in irreversible data loss or corruption.

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 [34] and AFix [33] (which target concurrency bugs, see §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 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 (§3.1), which motivates our desire to create an automated bug fixing solution. We then study the fixes for these bugs (§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).

Issue #s	Average Commits	Average Days from Open to Close	Max Days from Open to Close	Kind
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
535, 585, 949, 1103, 1118	2	15	38	API Misuse
Average	**13**	**28**	**66**	

Figure 1: The 26 PMDK bugs that 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.

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 [15] and memcached-pm [17]. We study all the 26 bugs in PMDK that were found using PMDK's bug detection tool, pmemcheck [55], and subsequently fixed. We chose these bugs because they are well-documented and validated in PMDK's issue tracker [25, 30], and pmemcheck provides rich information about detected bugs in the form of execution traces.

3.1 Study of Bugs

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 persistent memory 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.

3.2 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 1. These durability fixes are where persistence mechanisms are inserted within the same function (intraprocedurally) as the memory-modifying instruction. In Listing 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.

```
1 if (if_free != 0) {
2     *oid = NULL; //oid is a pointer to PM
3     // FIX: insert missing flush and fence
4     CLWB(oid);
5     SFENCE;
6 }
```

Listing 1: A missing-flush&fence bug with an intraprocedural fix, adapted from PMDK Issue #1103.

Interprocedural fixes. We provide an example of an interprocedural fix in Listing 2. These durability fixes are where persistence mechanisms are inserted outside of the function context (interprocedurally) of the memory-modifying instruction(s). In Listing 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).

```
1 if (/* condition */) {
2   memcpy(pmem_addr, vol_src, nbytes);
3   // FIX: insert missing flush and fence
4   pmem_persist(pmem_addr, nbytes);
5 }
```

Listing 2: A missing-flush&fence bug with an interprocedural fix, adapted from PMDK Issue #463.

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 §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 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 tradeoffs and technical challenges explain why fixing durability bugs is difficult, even though the fixes themselves can be very small. This is also an important tradeoff in practice, as over half (16/26, 62%) of the bugs in our study were fixed with interprocedural fixes.

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 §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.

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 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.

4.1 Overview

The system overview of Hippocrates is shown in Fig. 2. Hippocrates expects a PM-specific execution trace where each event in the trace includes the source line where the event 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

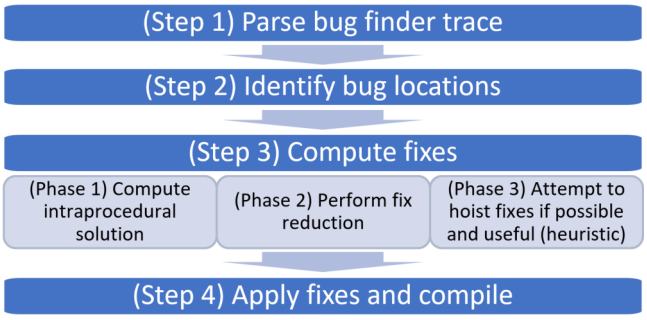

Figure 2: An overview of Hippocrates.

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.

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 (§4.3).

Notation. To present our proof sketches, we use a notation similar to prior work [36]. 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

```
1 void foo(char *pm_addr) {
2     pm_addr[0] = ...
3     CLWB(pm_addr);
4     // FIX: insert a memory fence
5     SFENCE();
6     // Without the fence, the system may lose data
7     ***CRASH***
8 }
```

Listing 3: An example missing-fence bug.

(e.g., an Intel TSX transaction [29] is treated as a single atomic unit). We denote an update to persistent memory as X (i.e., a store to X), a flush to persistent memory 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 [36]. 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 [3, 27, 28], as proposed in previous work [31].

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 flush and 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.

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. Without the `SFENCE` instruction inserted on Line 5, the `CLWB` instruction would not be ordered before the system crashed, potentially leading to data loss or data inconsistencies (see §2.1). Inserting a fix for this kind of bug can always be done safely; we provide a proof sketch below.

```
1 void foo(char *pm_addr) {
2     pm_addr[0] = ...
3     // FIX: insert a flush
4     CLWB(pm_addr);
5     // Without the flush, the system may lose data
6     SFENCE();
7     ***CRASH***
8 }
```

Listing 4: An example intraprocedural cache-line flush instruction insertion.

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, 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 $B(X)_{\text{fence}}$. fl

4.2.2 Intraprocedural Flush Insertion. Listing 4 shows an intraprocedural cache-line flush fix, in which a `CLWB` is inserted (Line 4) to write the modification of `pm_addr[0]` to PM.

Definition Formally, a bug $B(X)_{\text{flush}}$, indicating a missing 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. ↯

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

Listing 5: An example interprocedural fix as implemented by Hippocrates as a *persistent subprogram transformation.* The functions labeled "new" (and colored in blue) are generated by Hippocrates during the persistent subprogram transformation. Line 19 is replaced with line 21 during the transformation.

Theorem 2. If $B(X)_{flush}$ 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 $B(X)_{flush}$.

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

4.2.3 Intraprocedural Flush and Fence Insertion. Listing 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 $B(X)_{flush\&fence}$ 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 $B(X)_{flush\&fence}$.

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 $B(X)_{\text{flush\&fence}}$. ■

4.2.4 Interprocedural Fixes. Intraprocedural fixes are often expensive. Consider Listing 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 23. 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 18 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 5) may depend upon user input and be challenging or even impossible to calculate statically. In practice, 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 13) 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 10). 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 (§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 5, all modifications must be made durable before Line 23 (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 §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 $B(X)_Q$. ■

4.3 Optimization of Hippocrates's Fixes

After Hippocrates determines all bug locations and inserts intraprocedural fixes (Step 3, Phase 1 in Fig. 2), 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` (§3.2). Listing 6 provides an example of the calculation for Listing 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 6, the list of considered sites for the missing flush bug on line 3 consists of lines 15, 7, 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 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 7, 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 15 has 1 PM alias (through `pm_addr`), 0 non-PM aliases, and a score of 1. Since the call to `modify` on line 15 has the highest score, the heuristic performs a persistent subprogram transformation on `modify` and updates line 15 to call the updated function, resulting in the transformed program shown in Listing 5.

```
1  void update(char *addr, int idx, char val) {
2      // non-PM alias: 1, PM alias: 1 = score: 0
3      addr[idx] = val;
4  }
5  void modify(char *addr) {
6      // non-PM alias: 1, PM alias: 1 = score: 0
7      update(addr, ..., ...);
8  }
9  void foo(char *vol_addr, char *pm_addr) {
10     for (int i = 0; i < INT32_MAX; i++)
11         // (This call contributes +1 non-PM alias)
12         modify(vol_addr);
13     // (This call contributes +1 PM alias)
14     // non-PM alias: 0, PM alias: 1 = score: 1
15     modify(pm_addr);
16     ***CRASH***
17 }
```

Listing 6: An example heuristic calculation performed on Listing 5 to determine where to place the interprocedural fix.

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 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. ■

5 IMPLEMENTATION

Hippocrates is implemented primarily as an LLVM [38, 61] compiler pass (comprising 3300 SLOC [65]) which locates the sources of the bugs, computes the appropriate fixes, and applies them (Fig. 2, Steps 2–4). Step 1 (parsing bug finder output) is performed by Python scripts, which account for 1100 SLOC (including some Python scripts used for orchestrating linking and running PMDK unit tests). We use an implementation of Andersen's alias analysis [2, 9] for the whole-program alias analysis we perform to compute our heuristic.

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 350MB in size. This contributes to the difficulty of manually fixing PM durability bugs. Automating this process (Fig. 2, 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% [44] to 400× [43]).

The main engineering challenge is mapping from source lines to LLVM IR using debug information (Fig. 2, 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 [57]) 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.

5.2 Implementation of Fixes

As Hippocrates is implemented in LLVM and computes its fixes on LLVM bitcode (Fig. 2, Step 3), all fixes are generated (Fig. 2, Step 4) as LLVM intermediate representation (IR). Decompiling (mapping assembly/IR instructions back to lines of higher-level language code) is a difficult problem, however there are tools [35] 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.

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 [14] 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 [17], 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 [62], a state-of-the-art persistent and recoverable index representing a research prototype[1]. 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 [15], 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 [43, 44, 50].

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). 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 [13], 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 NVDIMMs, each with 128GB capacity. The server is also equipped with 256 GB of DRAM.

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 [42] 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 (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.

Effectiveness of the heuristics. We also compare the Full-AA heuristic to the Trace-AA heuristic. Both of these heuristics produced the same set of fixes on all the systems we test, resulting in identical end binaries and thus resulting in fixes with identical performance.

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 Fig. 3. 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.

[1] RECIPE contains multiple persistent indices. However, we only test the P-CLHT index because the other indices are not persistent.

Issue #s	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

Figure 3: Qualitative comparison of Hippocrates fixes and PMDK developer fixes.

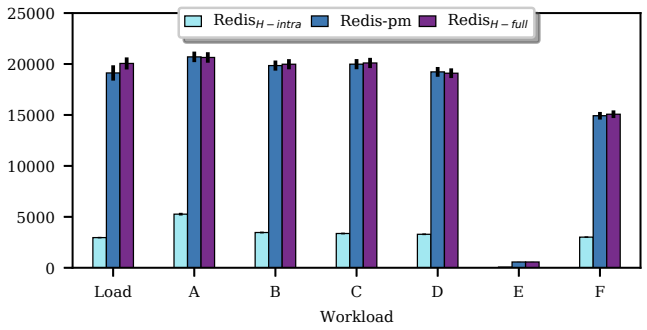

Figure 4: Performance of the three persistent versions of Redis with 95% confidence intervals. Hippocrates is able to provide fixes which are on-par with manual approaches.

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.

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 [15], 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

	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

Figure 5: Offline Overhead of Hippocrates.

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 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 [13] using a popular YCSB driver [12]. 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 Fig. 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× faster than Redis$_{\text{H-intra}}$.

6.4 Hippocrates's Overhead

Runtime Overhead. We measure the overhead of Hippocrates on all of our target systems and present the results in Fig. 5. 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 §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 1GB 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 (§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 4kB larger than the manually-developed Redis-pmem (an increase of 0.05%). The performance results from the Redis experiment (§6.3) suggests that the performance benefits from interprocedural fixes outweigh the effect of additional i-cache pressure.

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` (§6.1). Hippocrates is also accurate, fixing 8/11 PMDK unit test bugs in ways functionally identical to developer fixes, while fixing 3/11 bugs in functionally equivalent ways (§6.2). Hippocrates's fixes also yield good performance, equalling or exceeding the performance of manually-developed durability mechanisms (§6.3). Finally, we show that Hippocrates has low offline and size overhead (§6.4).

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 (§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 (§6.3).

8 RELATED WORK

PM 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 [14], NVM-Direct [4], and Pangolin [67]), to modified memory allocators (like Mnemosyne [64] and NV-Heaps [10]) to PM-specific language extensions (such as NVL-C [18] and NVM-Direct's preprocessor [4]). Various works also focus on logging mechanisms [6, 8, 20, 23] 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).

PM Debugging Tools. As discussed in §2.2, the challenges of debugging PM durability bugs has spurred many recent works in PM-specific bug detection [59]. PMTest [44] is a trace-validation framework, where each PM operation produces a trace event which is asynchronously validated to detect a durability bug. Some other tools are based on binary instrumentation. `pmemcheck` [55] is a binary instrumentation tool designed by Intel for PMDK, which is based on valgrind. Persistency Inspector [51] is another tool developed by Intel, based on proprietary binary instrumentation included in Intel Parallel Studio XE. XFDetector [43] is a fault-injection tool based on Intel PIN, which is specifically tailored at finding crash consistency bugs caused by buggy PM update orderings. Finally, Agamotto [50] is a symbolic-execution tool based on KLEE [5]. Agamotto builds a symbolic model of PM on top of KLEE to provide thorough and automated discovery of PM durability bugs across a variety of PM applications and PM libraries.

General-Purpose Automated Program Repair. Many systems are able to perform general-purpose program repair and solve any class of faults by using heuristics [39, 40, 58]. In particular, genetic programming (or Genetic Improvement [54]) is an increasingly popular method for automatic program repair. GenProg [39, 40], for example, uses a genetic programming method to mutate programs to generate fixes for off-the-shelf programs. Janus Manager [21] and SapFix [47] 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 [33], 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 [34] targets a wider variety of concurrency bugs—CFix accepts bug reports from a variety of concurrency bug finders [52] and produces fixes which are rigorously tested in a principled manner to provide some correctness guarantees.

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.

ACKNOWLEDGMENTS

We first thank Andrew Loveless (University of Michigan and Avionic Systems Division at NASA Johnson Space Center), who helped us refine our proof language for demonstrating the safety of Hippocrates's fixes. We also thank our shepherd, Steven Swanson, and the anonymous reviewers for their valuable feedback. This work is supported by Applications Driving Architectures (ADA) Research Center (a JUMP Center co-sponsored by SRC and DARPA), a Microsoft Ph.D. Fellowship, and the National Science Foundation under grant DGE-1256260 and CAREER award 1942218. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

A ARTIFACT APPENDIX

A.1 Abstract

We provide the public repository for Hippocrates, which is available on GitHub. Hippocrates's artifact includes instructions for building and running Hippocrates, as well as scripts and instructions used to reproduce the core results from our paper.

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

- **Program:** python3.6, llvm-8
- **Compilation:** WLLVM, clang-8, clang++-8
- **Run-time environment:** Ubuntu 20.04.1 LTS
- **Hardware:** One Intel x86 machine with Intel Optane DC Persistent Memory Modules.
- **Experiments:** Fixing PMDK, RECIPE, and memcached-pmem bugs; Redis-pmem performance analysis
- **How much disk space required (approximately)?:** 2 GB on a PM file system partition
- **How much time is needed to prepare workflow (approximately)?:** <1 hour
- **How much time is needed to complete experiments (approximately)?:** ~10 hours
- **Publicly available?:** Yes (GitHub repository: https://github.com/efeslab/hippocrates/)
- **Code licenses (if publicly available)?:** MIT License
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.4464894

A.3 Description

The artifact contains all of Hippocrates's source code as well as the instructions required to build and run Hippocrates and its dependencies. The **README.md** file at the root of the repository contains all of the instructions used in the Artifact Evaluation process.

A.3.1 How to Access. Hippocrates is available at our public GitHub repository (see appendix A.2). The archived version, which was used in the Artifact Evaluation process is available from Zenodo (see appendix A.2).

A.3.2 Hardware Dependencies. Running the performance experiments (§6.3) requires access to Intel Optane DC Persistent Memory modules. This hardware is not required to build and run Hippocrates, but it is required for accurate performance results.

A.4 Installation

Users first install all external dependencies (Python dependencies are located in the `requirements.txt` file in the root directory of the archive and all other dependencies are located in `install-deps.sh`). Users then install all submodule dependencies and then compile Hippocrates and all of the test applications. The detailed instructions for this process are in the main README under the "artifact functional criteria" (located here: https://github.com/efeslab/hippocrates/blob/artifact-evaluation/README.md#artifacts-functional-criteria).

A.5 Experiment Workflow

Users first install and build Hippocrates and all dependencies. The test applications are compiled using WLLVM so that a native binary and LLVM bitcode are generated. The test applications are then run under `pmemcheck` to generate traces and PM bug information (this is later used by Hippocrates to fix the PM bugs found in the programs). The trace and the LLVM bitcode from the application are used by Hippocrates to generate new LLVM bitcode with added fixes for all the detected bugs. The new bitcode is then compiled into a native binary. Detailed instructions are provided in the main README.

A.6 Evaluation and Expected Results

The artifact provides instructions and utilities for reproducing the main components from Hippocrates's evaluation (§6):

- Fixing all previously reported bugs (§6.1)
- The Redis-pmem performance experiments (§6.3)
- Hippocrates's overhead (§6.4)

Detailed instructions for reproducing this instructions are included in the main README under the "results reproduced" section (https://github.com/efeslab/hippocrates/blob/artifact-evaluation/README.md#results-reproduced). These instructions are also available in the same location in the Zenodo archive.

A.7 Notes

We are endeavoring to maintain HIPPOCRATES as an open-source tool. Any issues that are found with the available artifact or any needed clarifications can be submitted as GitHub issues on our repository (https://github.com/efeslab/hippocrates/issues).

REFERENCES

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

[2] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language.* Ph.D. Dissertation. University of Cophenhagen.

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

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

[5] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. 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* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.

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

[7] Himanshu Chauhan, Irina Calciu, Vijay Chidambaram, Eric Schkufza, Onur Mutlu, and Pratap Subrahmanyam. 2016. NVMOVE: Helping Programmers Move to Byte-Based Persistence. In *4th Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 16)*. USENIX Association, Savannah, GA, 1–7. https://www.usenix.org/conference/inflow16/workshop-program/presentation/chauhan

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

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

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

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

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

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

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

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

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

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

[18] Joel E Denny, Seyong Lee, and Jeffrey S Vetter. 2016. 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*. 125–136.

[19] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems* (Amsterdam, The Netherlands) *(EuroSys '14)*. ACM, New York, NY, USA, Article 15, 15 pages. https://doi.org/10.1145/2592798.2592814

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

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

[22] Swapnil Haria, Mark D Hill, and Michael M Swift. 2020. 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*. 775–788.

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

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

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

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

[27] Intel. 2020. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Vol. 2. Intel Corporation, Chapter 4, 1198,1787.

[28] Intel. 2020. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Vol. 2. Intel Corporation, Chapter 3, 739–742,748–749.

[29] Intel. 2020. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Vol. 1. Intel Corporation, Chapter 16, 391–398.

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

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

[32] 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. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. arXiv:1903.05714 [cs.DC]

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

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

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

[36] Leslie Lamport. 2019. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*. Association for Computing Machinery, 179–196.

[37] Philip Lantz, Subramanya Dulloor, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. 2014. Yat: A Validation Framework for Persistent Memory Software. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 433–438. https://www.usenix.org/conference/atc14/technical-sessions/presentation/lantz

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

[39] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. 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)*. IEEE, 3–13.

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

[41] E. Lee, H. Bahn, S. Yoo, and S. H. Noh. 2014. 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*. 405–410. https://doi.org/10.1109/MASCOTS.2014.56

[42] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. RECIPE: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 462–477.

[43] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. 2020. Cross-Failure Bug Detection in Persistent Memory Programs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1187–1202.

[44] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. 2019. PMTest: A fast and flexible testing framework for persistent memory programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 411–425.

[45] Pratyush Mahapatra, Mark D. Hill, and Michael M. Swift. 2019. Don't Persist All: Efficient Persistent Data Structures. arXiv:1905.13011 [cs.DB]

[46] Virendra J Marathe, Margo Seltzer, Steve Byan, and Tim Harris. 2017. Persistent memcached: Bringing legacy code to byte-addressable persistent memory. In

9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17). 7 pages.

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

[48] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. 2017. 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* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 135–148. https://doi.org/10.1145/3037697.3037730

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

[50] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. 2020. Agamotto: How Persistent is your Persistent Memory Application?. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 1047–1064. https://www.usenix.org/conference/osdi20/presentation/neal

[51] Kevin Oleary. 2018. 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.

[52] Soyeon Park, Shan Lu, and Yuanyuan Zhou. 2009. CTrigger: exposing atomicity violation bugs from their hiding places. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*. 25–36.

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

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

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

[56] Azalea Raad, John Wickerson, and Viktor Vafeiadis. 2019. 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 (2019), 1–27.

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

[58] Seemanta Saha, Ripon K. Saha, and Mukul R. Prasad. 2019. Harnessing Evolution for Multi-Hunk Program Repair. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 13–24. https://doi.org/10.1109/ICSE.2019.00020

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

[60] Steven Swanson. 2019. Early Measurements of Intel's 3DXPoint Persistent Memory DIMMs. https://www.sigarch.org/early-measurements-of-intels-3dxpoint-persistent-memory-dimms/

[61] The LLVM Project. 2019. Overview — LLVM 8 documentation. https://releases.llvm.org/8.0.0/docs/index.html.

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

[63] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*. USENIX Association, 5–5. http://dl.acm.org/citation.cfm?id=1960475.1960480

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

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

[66] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-Based Single Level Systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. 167–181. https://www.usenix.org/conference/fast15/technical-sessions/presentation/yang

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

[68] Yiying Zhang and Steven Swanson. 2015. A study of application performance with non-volatile main memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. 1–10. https://doi.org/10.1109/MSST.2015.7208275