Towards Bug-free Persistent Memory Applications

Ian Neal
University of Michigan

Andrew Quinn
University of Michigan

Baris Kasikci
University of Michigan

Persistent Memory (PM) aims to revolutionize the storage-memory hierarchy, but programming these systems is error-prone. Our work investigates how to to help developers write better, bug-free PM applications by automatically debugging them. We first perform a study of bugs in persistent memory applications to identify the opportunities and pain-points of debugging these systems. Then, we discuss our work on AGAMOTTO*, a generic and extensible system for automatically detecting PM bugs. Unlike existing tools that rely on extensive test cases or annotations,* AGAMOTTO *automatically detects bugs in PM systems by extending symbolic execution to model persistent memory.* AGAMOTTO *has so far identified 84 new bugs in 5 different PM applications and frameworks while incurring no false positives. We then discuss* HIPPOCRATES*, a system that automatically fixes bugs in PM systems.* HIPPOCRATES *"does no harm": its fixes are guaranteed to fix an PM bug without introducing new bugs. We show that* HIPPOCRATES *produces fixes that are functionally equivalent to developer fixes and that* HIPPOCRATES *fixes have performance that rivals manually-developed code.*

Persistent memory (PM) technologies aim to revolutionize the storage-memory hierarchy. PM technologies, such as Intel Optane DC [3], are roughly $8\times$ less expensive than DRAM [1] and offer disk-like durability with access latencies that are only 2–$3\times$ higher than DRAM latencies [4]. PM can be accessed using the conventional load and store instructions and thus offers persistence without needing heavyweight file-system operations. Popular applications (memcached, Redis) and companies (VMware, Oracle) are employing PM.

Alas, programming PM systems is error-prone [7, 11]. Updates to PM are cached in volatile CPU caches, requiring developers to explicitly flush cache lines to guarantee that updates are written to PM. Moreover, cache flushes are weakly ordered on most architectures (i.e., flushes do not follow store order), so developers must insert memory fences to order updates as required for crash consistency. The misuse or omission of these mechanisms results in *durability bugs* which compromise program correctness. Existing work on PM debugging targets specific systems [9, 10] or requires significant developer effort [5, 6]. In contrast, we're investigating systems that help developers write bug-free PM applications by fully automating the process of debugging them.

To begin our work on automating PM debugging, we performed a study of 63 bugs in PM applications and APIs. We first investigated the automation of bug finding. We identified two application-independent patterns of PM misuse (*missing flush/fence* and *extra flush/fence*) which cover the majority (89%, or 56/63) of bugs in our study and can be detected automatically. The remaining bugs are application-specific; for example, many involve misusing transactions when updating data structures. Existing PM testing approaches conflate these classes and require annotations or expensive model-checking methods to detect even application-independent PM bugs.

We then examine the difficulty of fixing PM bugs, focusing on *durability bugs*, the bugs caused by missing flushes/fences, because they are application-independent bugs that lead to data loss. We analyzed 26 reported and fixed bugs which were discovered by Intel's own bug finding tool, `pmemcheck`. These bugs were arduous to manually debug and fix, taking on average weeks (23 days) and up to months (66 days) to fix, and require numerous attempts (13 commits on average) to produce a complete fix. The main challenge with fixing these bugs arises because of a key tradeoff between performance and simplicity; each bug can be fixed in a myriad of different ways which developers have to carefully consider before implementing a patch. As a result, even though developers have effective PM-specific bug finding tools, actually *fixing* a durability bug in a PM system remains challenging.

Our work is the first to automate both PM bug finding (AGAMOTTO) and bug fixing (HIPPOCRATES). Critically, our work can be *automatically* applied to *arbitrary* PM applications—it requires no unit tests, source-code modifications/annotations, nor does it place restrictions on application behavior (e.g., applications can use arbitrary PM allocators).

First, we present AGAMOTTO [8][1], a framework that uses the insights from our study to automatically detect bugs in PM applications. Rather than rely on unit tests, AGAMOTTO uses symbolic execution [2] to thoroughly explore the state space of a program. AGAMOTTO applies symbolic execution to PM by introducing a memory model to track updates made to PM by the explored program paths. To mitigate the infamous "path-explosion" problem in traditional symbolic analysis, which arises because the state space of possible executions grows exponentially, AGAMOTTO introduces a novel search algorithm that targets execution states that are most likely to lead to PM bugs. AGAMOTTO uses alias analysis together with a back propogation algorithm to assign a priority to execution states based upon the number of PM-modifying operations (stores, flushes, etc.) that each state can reach; the system steers symbolic execution towards program states that can reach the most PM modifying operations.

To detect bugs, AGAMOTTO supports *bug oracles*, which use the PM state that the system gathers along each execution path to identify persistency bugs. AGAMOTTO automatically

[1] See: **https://www.usenix.org/conference/osdi20/presentation/neal**

detects PM bugs using two *universal persistency bug oracles* based on the common application-independent patterns of PM misuse identified by our study. To identify application-specific persistency bugs, AGAMOTTO enables *custom persistency bug oracles*—we implement two such oracles in AGAMOTTO to detect bugs related to the misuse of PMDK's transactional API [5, 6]. We used AGAMOTTO to find 84 new persistency bugs in 5 real-world PM storage systems and research prototypes. In particular, we found 13 new correctness and 70 new performance bugs using the universal persistency bug oracles, and 1 new correctness bug using a custom persistency bug oracle. We report all bugs to their authors; so far 40 of them have been confirmed and none denied.

We next present HIPPOCRATES[2], an automated PM bug fixing tool guaranteed to "do no harm". HIPPOCRATES automatically reasons through the performance/simplicity tradeoff inherent in fixing PM bugs by mimicking the reasoning of PM developers. The system considers two types of fixes. Simple *intraprocedural fixes* insert a flush or fence in-line with the store that is missing one and make it easy to reason about the durability of the application. Unfortunatley, these simple fixes lead to poor performance when they are often accessed with volatile data (e.g., adding a flush in `memcpy`). So, HIPPOCRATES considers more complex *interprocedural fixes*, which add flush or fence operations to other functions in the call stack that resulted in the missing flush. To perform an interprocedural fix, HIPPOCRATES performs persistent subprogram creation (i.e., duplicating a function and inserting flushes and a single memory fence at each exit point to preserve program semantics while adding durability mechanisms).

Critically, HIPPOCRATES "does no harm" by provably guaranteeing that the fixes that it inserts fix the bug without introducing any new bugs. We define a *bug* as the **possibility** of incorrect program behavior. We show that HIPPOCRATES can safely and automatically apply the three kinds of fixes that it performs, namely intraprocedural fence, intraprocedural flush, and persistent subprogram creation. Intuitively, this is because the mechanisms that HIPPOCRATES uses to fix PM durability bugs (cache-line flush and/or memory fence instructions) do not introduce the possibility of *any* new program behaviors, and can therefore not cause any new bugs. Intuitively, this is because a missing durability instruction does not preclude the effects of that instruction (e.g., memory pressure can evict arbitrary cache lines without using an explicit cache-line flush).

HIPPOCRATES automates PM bug fixing by reusing the output of PM bug finding tools to create safe fixes. HIPPOCRATES computes a fix for each bug using a three-phase process: first, it computes the simplest possible intraprocedural fix; second, HIPPOCRATES performs "fix reduction," where redundant fixes (e.g., flushes to the same cache line) are merged together; and third, it performs a heuristic transformation to determine if a fix should be "hoisted," i.e., if an intraprocedural fix should be converted into an interprocedural fix using persistent subprogram creation. The heuristic chooses a level in the call stack which is most likely to only operate on PM (e.g., avoid performing persistence operations on helpful functions which operate on PM and DRAM). Regardless of where HIPPOCRATES inserts a fix, the fix is provably safe.

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. Then, we show that HIPPOCRATES produces fixes with strong performance. Notably, we show that a PM port of Redis using HIPPOCRATES fixes can attain even better performance than a manually-developed PM-port of Redis.

[2]See: https://asplos-conference.org/papers/

References

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

[2] 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.

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

[4] 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.

[5] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. Cross-failure bug detection in persistent memory programs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1187–1202, 2020.

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

[7] 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.

[8] 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. USENIX Association, November 2020.

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

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

[11] Steve Scargall. Debugging persistent memory applications. In *Programming Persistent Memory*, pages 207–260. Springer, 2020.