HIPPOCRATES: Healing Persistent Memory Bugs Without Doing Any Harm
Extended Abstract

Ian Neal
University of Michigan

Andrew Quinn
University of Michigan

Baris Kasikci
University of Michigan

1. Motivation

Persistent memory (PM) technologies aim to revolutionize the storage-memory hierarchy [34, 37]. PM technologies, such as Intel Optane DC [12, 20], 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 [21,26,41,45]. PM can be accessed using the conventional load and store instructions and thus offers persistence without needing heavyweight file-system operations. Popular applications (memcached [13] and Redis [11]) and companies (e.g., VMware and Oracle [19]) have already begun employing PM.

Alas, programming PM systems is error-prone [5, 7, 17, 30, 31, 33, 34, 40, 42–44]. 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 necessary for crash consistency. The misuse or omission of these mechanisms results in *durability bugs* which compromise program correctness. The challenges of finding these durability bugs have spurred many works in PM-specific bug finding tools [28, 29, 36, 38].

However, even with these effective PM-specific bug finding tools, *fixing* durability bugs in PM systems is still challenging. In this work, we first analyze 26 bugs reported by Intel's own bug finding tool, `pmemcheck`, and 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`. Ultimately, the PM bugs in our study take 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.

We find that fixing PM durability bugs is complicated due to a key tradeoff between performance and simplicity. Simple intraprocedural fixes insert a flush or fence in-line with the store that is missing one, making it easy to reason about the durability of the application. However, if the intraprocedural fix is often accessed with volatile data (e.g., adding a flush in `memcpy`), the performance of the application suffers. Instead, a developer will employ a more complicated interprocedural fix, in which they add flush or fence operations to other functions in the call stack that resulted in the missing flush. Such a fix can be more efficient, but are trickier to place correctly in the program to ensure crash consistency. These tradeoffs and technical challenges explain why fixing durability bugs is difficult, even though the fixes are seemingly simple.

It is time to explore methods of fixing PM durability bugs, as finding these bugs is only part of the challenge.

2. Limitations of the State of the Art

Recent work has developed PM-specific debugging tools, such as AGAMOTTO [35], PMTest [29], XFDetector [28], `pmemcheck` [38, 40], and Persistency Inspector [36, 40]. However, even with such useful tools, fixing durability bugs in PM systems is challenging and time-consuming. A plethora of work focuses on building reliable APIs, libraries, and language extensions to make PM programming easier for developers [2, 4, 6, 10, 14, 15, 18, 44], but durability bugs can still occur if these mechanisms are misused or contain internal durability bugs.

There is a long line of research on automated bug fixing [16, 24, 25, 32, 39], some of which is deployed in production (e.g., at Janus Rehabilitation [16] and Facebook [32]). However, these tools are best-effort, meaning that their fixes cannot always be proven to be *safe* (i.e., not introduce incorrect program behavior), making them a poor fit for crash-consistent PM systems, as buggy patches could lead to data corruption. Automated bug-fixing systems for specific domains, like AFix [22] and CFix [23] for concurrency bugs, give stronger guarantees and serve as inspiration for this work.

3. Key Insights

Our main insight is that most PM durability bugs can be *safely* fixed with fixes that *guarantee* correctness (i.e., they do not hurt program correctness). We define a *bug* as the **possibility** of incorrect program behavior. Then, we show that the mechanisms used 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 instruction).

4. Main Artifacts

Based on the bug fixes that we analyze in our study, we develop three kinds of fixes that can be applied automatically in PM systems: (1) intraprocedural fence instruction insertion (i.e., inserting a fence in-line with an update and flush to PM); (2) intraprocedural flush instruction insertion (i.e., inserting a flush in-line with an update to PM); and (3) 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). We provide a proof sketch for each class of fix arguing why these

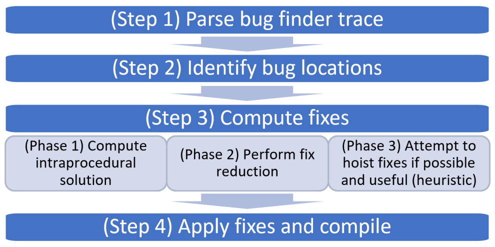
(Step 1) Parse bug finder trace
(Step 2) Identify bug locations
(Step 3) Compute fixes
(Phase 1) Compute intraprocedural solution
(Phase 2) Perform fix reduction
(Phase 3) Attempt to hoist fixes if possible and useful (heuristic)
(Step 4) Apply fixes and compile

Figure 1: An overview of HIPPOCRATES.

fixes are guaranteed to *safely* fix the original durability bug (i.e., fix the bug without creating new bugs).

We then develop HIPPOCRATES, an automated PM bug fixing tool which applies these fixes. Fig. 1 shows the high level design of HIPPOCRATES. HIPPOCRATES accepts a PM-specific execution trace containing data commonly provided by existing PM bug finding tools (stack trace, etc.); many tools generate this data by default (e.g., `pmemcheck`), others can easily be modified to produce it. HIPPOCRATES first parses this trace into a tool-independent format (Step 1). Then, HIPPOCRATES uses the trace to locate the original operation that causes each bug detected by the bug finder (e.g., the unflushed store causing 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 computes fixes using a three-phase process (Step 3): first, for all bugs, it computes the simplest possible fix using intraprocedural fixes; second, HIPPOCRATES performs “fix reduction,” where fixes that are redundant (e.g., flushes to the same cache line) are merged together; 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) should be converted into an interprocedural fix (i.e., in a caller function).

We test HIPPOCRATES using both production and research systems; PMDK (Intel’s Persistent Memory Development Kit [10]), P-CLHT (a persistent index from RECIPE [27]), memcached-pm (a PM-port of memcached [31] [13]), and Redis-pmem (a PM-port of Redis [3] maintained by Intel [11]). We test PMDK using developer created unit tests, P-CLHT using an example application provided by the RECIPE authors, and memcached-pm and Redis-pmem using standard YCSB workloads [9] using a popular YCSB driver [8].

5. Key Results and Contributions

We use HIPPOCRATES to automatically fix all 23 of the 23 durability bugs we find and reproduce in PMDK [10], P-CLHT (from RECIPE [27]), and memcached-pm [13]. We manually verify that HIPPOCRATES is able to correctly fix all the bugs using the bug finding tool that originally found the bugs (`pmemcheck`). We compare developer’s fixes and HIPPOCRATES automated fixes for all bugs where patches are available (11 PMDK bugs), and find that in most cases (9/11), HIPPOCRATES fixes are functionally identical to developer fixes. In the remaining cases (2/11), 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 runtime).

We show the effectiveness of HIPPOCRATES’s interprocedural fix heuristic with a case study of Redis-pmem [11]. We test Redis-pmem against Redis$_{\text{H-full}}$, a version where all flushes have been automatically inserted by HIPPOCRATES instead of by a developer, and show that Redis$_{\text{H-full}}$ matches or exceeds the performance of Redis-pmem (up to 7% increase in throughput on YCSB workloads). In this experiment we also demonstrate that HIPPOCRATES’s interprocedural fixes are required to provide good overall fix performance, as Redis$_{\text{H-full}}$ is 2.4–11.7$\times$ faster than Redis$_{\text{H-intra}}$, a HIPPOCRATES-fixed version of Redis which only inserts simple intraprocedural fixes. We also show that HIPPOCRATES produces fixes with latency comparable to deployed automated bug fixing tools [16], taking 30 minutes to fix all bugs in Redis and less than 15 minutes on all other systems we test.

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.
- We develop HIPPOCRATES, a novel automated PM bug fixing tool. HIPPOCRATES uses safe fixes coupled 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.

6. Why ASPLOS

This paper tackles the new problem of automatically fixing PM durability bugs, which is at the intersection of architecture (the use of emerging PM technologies) and programming languages (automated program repair using compiler techniques and static analysis) research. Therefore, we believe that ASPLOS is a good venue for this work.

7. Citation for Most Influential Paper Award

This paper represents a innovative work in automatically repairing durability bugs in persistent memory systems and is the first work to tackle the problem of how to fix the bugs found by automatic persistent memory bug-finding tools. The authors develop several strategies for *safely* fixing durability bugs, thus fixing the original durability bug without introducing any new correctness problems. The paper demonstrated that HIPPOCRATES, the tool built to automatically fix these bugs, is both effective and efficient, thus proving the value of a targeted and “no-harm” approach to automated bug fixing.

References

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

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

[3] Josiah L Carlson. *Redis in action*. Manning Publications Co., 2013.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[19] Intel. Intel optane persistent memory workload solutions. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-persistent-memory-solutions.htmll.

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

[21] 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.

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

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

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

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

[26] E. Lee, H. Bahn, S. Yoo, and S. H. Noh. Empirical study of nvm storage: An operating system's perspective and implications. In *2014 IEEE 22nd International Symposium on Modelling, Analysis Simulation of Computer and Telecommunication Systems*, pages 405–410, Sep. 2014.

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

[28] 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.

[29] 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.

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

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

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

[33] 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.

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

[35] 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.

[36] 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.

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

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

[39] Seemanta Saha, Ripon K. Saha, and Mukul R. Prasad. Harnessing evolution for multi-hunk program repair. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 13–24. IEEE Press, 2019.

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

[41] Steven Swanson. Early measurements of intel's 3dxpoint persistent memory dimms, Apr 2019.

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

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

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

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