Jaaru: Efficiently Model Checking Persistent Memory Programs Extended Abstract

Hamed Gorjiara
hgorjiar@uci.edu
University of California, Irvine

Guoqing Harry Xu
harryxu@cs.ucla.edu
University of California, Los Angeles

Brian Demsky
bdemsky@uci.edu
University of California, Irvine

1. Motivation

Non-volatile memory technologies such as Intel Optane DC memory have recently become available. These technologies provide both near DRAM performance and the persistency of disks. Persistent memory presents an interface much like DRAM — programs perform normal load and store instructions to access persistent storage, bypassing the operating system to achieve higher performance — and provides a way for developers to redesign how programs manage data. For example, persistent memory makes it possible to use a single copy of a data structure for both a working in-memory copy of the data structure and as a persistent store for the data. This eliminates the overheads of serialization and deserialization and may simplify working with large data sets.

Ensuring crash consistency for persistent memory data structures is extremely challenging. Stores to persistent memory do not immediately become persistent — they are first written to the volatile cache and do not become persistent until their cache line is flushed to persistent memory. As this can take an arbitrary amount of time to happen due to cache capacity constraints, later stores can be made persistent before earlier stores. To enforce ordering properties as well as that stores are made persistent in a timely manner, it is necessary to use special instructions to explicitly flush cache lines, such as `clflush` and `clflushopt`.

Writing correct data structures for persistent memory in the presence of failures requires developers to carefully reason about the subtle ordering and persistency properties their code relies upon and to ensure that they enforce those properties with the appropriate use of flush and fence instructions. Our experience with persistent memory benchmarks suggests that it is very easy to make a mistake (*e.g.*, forget to add the necessary flush instructions).

Testing persistent memory programs is particularly challenging. Missing cache flush instructions would not become apparent unless the machine suffers from a failure at a very specific interval in the execution. Moreover, trying to test data structures by abruptly cutting power to a machine creates numerous practical challenges including the risk of corrupting other programs on the machine.

This paper presents a novel model checker, named Jaaru, to find bugs in persistent memory programs. Jaaru exhaustively explores the space of executions from non-determinism due to cache line persistency without needing any user annotations. Jaaru is also efficient due to (1) a constraint refinement based approach for partial order reduction that drastically reduces the state space to be explored and (2) an effective leverage of *commit stores* – a common programming pattern – that can reduce the number of steps taken from *exponential* in the length of a program execution to *quadratic*.

2. Limitations of the State of the Art

State of the art techniques for finding bugs in persistent memory programs fall into two primary categories:

Persistency Property Checkers: There is a line of work including PMTest [7], PMemcheck [1], and XFDetector [6], which checks various properties of stores such as persistency and ordering relative to other stores. These property checkers require annotating the code to specify the properties to be checked. Developing annotations is labor-intensive and error-prone itself; as such, the bug report depends very much on the quality of annotations — problems with annotations can lead to both false positives and false negatives. Moreover, these tools are not *exhaustive* — they focus on single executions and hence can miss bugs.

Naïve Model Checking: Model checking has been used extensively in the system community [5, 8, 9, 10] to find crash consistency bugs in file system implementations. These techniques are not scalable if applied directly to check persistent memory programs — persistent memory is byte-addressable and has many more states than disks that are accessed at the block granularity through the OS. In fact, there was an effort [3] from Intel Research that attempts to apply an eager model checking approach to exhaustively generate all possible persistent memory states following a failure. The number of states grows exponentially in the number of unflushed stores and thus Yat's approach does not scale.

3. Key Insights

Naïve model checking fails for persistent memory programs because the number of persistent memory states that must be explored is exponential in the number of unflushed stores. Jaaru is built on two major insights, as elaborated below.

First, instructions that explicitly flush a cache line into persistent memory such as `clflush` and `clflushopt` set a constraint on the *time at which a cache line was previously flushed*. Jaaru *builds* such constraints during a pre-failure execution and *refines* them (*i.e.*, make them tighter) during a post-failure execution using values that are actually read in the post-failure execution. Jaaru employs a partial-order reduction technique

that explicitly uses these refined constraints to significantly reduce search space.

Second, Jaaru follows from an important observation that persistent memory programs often perform a *commit write* to indicate that data has been fully persistent and is now in a consistent state. PM programs typically read from this commit write first to determine whether the data has been persisted, and then only read the data if it is known to be persisted. For example, for a tree data structure, the program may set the root pointer *after* it makes sure that all nodes in the tree have been well persisted (with flush instructions). The post-failure execution reads the root first and checks if the pointer is null; a null pointer indicates data inconsistency and hence, the program would not read the rest of the data any more.

While correct persistent memory programs may have a large number of unflushed stores when a failure occurs, *this observation indicates that their post-failure executions typically only read from a small number of such unflushed stores* because the read from this commit store (*e.g.*, the root pointer check) explicitly precludes accessing unflushed stores that are part of the data protected by the store.

By leveraging this observation, we develop a *lazy exploration technique* to explore only pre-failure stores that *are actually read by a post-failure execution*. Compared to an eager model checker that explores all possible states at a failure, our lazy approach waits until the post-failure code is executed to do the exploration, and hence *exponentially* reduces the executions that must be explored.

Note that leveraging commit stores leads to efficiency, but has nothing to do with the thoroughness of the search — Jaaru still exhaustively explores all possible states. For programs without such commit stores, Jaaru would just need to spend more time on the exploration.

Putting them all together, Jaaru is able to automatically explore only a handful of executions per injected crash location for persistent memory programs.

Relative to previous work on model checking persistent memory programs, Jaaru reduces the number of executions that must explored for the RECIPE benchmarks by many orders of magnitude. As a result, Jaaru can in seconds model check persistent memory programs that would require infeasible amounts of time for previous algorithms.

Relative to previous work on persistent memory bug finding tools, Jaaru is exhaustive while existing tools explore only a single execution or rely on carefully designed test cases. Jaaru can catch classes of bugs that they will miss. It does *not* require any manual effort of writing annotations, avoiding the issue that errors in annotations may cause bugs to be missed.

4. Main Artifacts

The main artifacts include the implementation of the Jaaru model checker with an LLVM frontend for instrumenting C/C++ code and a runtime library that implements the Jaaru model checker, as well as a set of bugs Jaaru found in well-studied persistent memory programs. We will make these artifacts available on Github.

5. Key Results and Contributions

Jaaru is the first practical model checker for persistent memory programs with *full automation* and *ultra efficiency*. Our contributions include:

- A *novel partial order reduction algorithm* that leverages the two ideas presented above – (1) constraint refinement and (2) commit stores – to scale to real persistent memory programs. These new ideas enable our model checker to be many orders of magnitude more efficient than previous exhaustive approaches;
- An implementation of Jaaru that fully supports the TSO memory model;
- An evaluation of the approach with PMDK [2] and RECIPE [4] that found bugs in every single program in our benchmark set with a total of 11 programs.

Usage Scenarios. Despite the aforementioned advantages, model checking is *not* a silver bullet for bug finding in PM programs. For example, even if Jaaru is orders of magnitude more efficient than existing model checkers such as Yat, Jaaru needs to execute a program many times to fully explore the state space, taking a large amount of time for checking. Compared to testing tools such as PMTest and XFDetector, Jaaru is able to find more bugs, in a completely automated fashion, but has difficulties checking programs such as Redis that interact with the outside world and whose non-determinism from the network would require deterministic replay for a model checker to work. As such, the best use case for Jaaru is to exhaustively check widely-used libraries such as PMDK, finding as many potential bugs as possible before their release, while non-exhaustive tools such as PMTest and XFDetector can scalably check large programs and find bugs only when they are triggered by tests.

6. Summary

Jaaru shows how to scale model checking of persistent memory programs to find errors in wide range of persistent memory data structures and programs. The partial order reduction techniques introduced in this paper improve the efficiency of model checking by *many orders of magnitude* to optimize a technique that was previously completely infeasible for persistent memory programs so that it can finish in seconds for code that accesses persistent memory.

References

[1] An introduction to pmemcheck (part 1) - basics. https://pmem.io/2015/07/17/pmemcheck-basic.html, July 2015.

[2] Intel Corporation. Persistent memory development kit. https://pmem.io/pmdk/, 2020.

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

[4] 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, page 462–477, New York, NY, USA, 2019. Association for Computing Machinery.

[5] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. Samc: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 399–414, USA, 2014. USENIX Association.

[6] 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*, ASPLOS '20, page 1187–1202, New York, NY, USA, 2020. Association for Computing Machinery.

[7] 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*, ASPLOS '19, page 411–425, New York, NY, USA, 2019. Association for Computing Machinery.

[8] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding crash-consistency bugs with bounded black-box crash testing. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 33–50, 2018.

[9] Junfeng Yang, Can Sar, and Dawson Engler. Explode: A lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, page 10, USA, 2006. USENIX Association.

[10] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst.*, 24(4):393–423, November 2006.