Jaaru: Efficiently Model Checking Persistent Memory Programs

Hamed Gorjiara
University of California, Irvine

Guoqing Harry Xu
University of California, Los Angeles

Brian Demsky
University of California, Irvine

1. Introduction

Ensuring crash consistency for persistent memory data structures is extremely challenging. Stores to persistent memory (PM) 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.

Prior work for finding bugs in persistent memory programs fall into two primary categories. There is a line of work including PMTest [6] and XFDetector [5], 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. These tools are not *exhaustive* — they focus on single executions and hence can miss bugs. The model checker Yat [3] from Intel implements 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.

2. Jaaru

We present a novel model checker [2], 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. Our key insight is to enumerate post-failure executions and not post-failure states. Jaaru uses a constraint-refinement based approach for partial order reduction that drastically reduces the executions to be explored. This approach effectively leverages *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*.

2.1. Constraint-Refinement

Jaaru uses dynamic partial order reduction (DPOR) to determine that states produce the same execution, and instead explore the equivalent post-failure executions once. This novel DPOR technique considers the effect of cache line flushes and volatile memory. To reduce the search space, we use `clflush` instructions to infer constraints on the *last time each cache line was written back to persistent memory* in a pre-failure execution and refine these constraints in a post-failure execution to narrow down when a cache line became persistent. For example, when a `clflush` instruction leaves the store buffer, it forces the cache line to be written back to persistent memory. Hence, the `clflush` instruction essentially sets a constraint that the last time the corresponding cache line is written back to memory must be *after* the `clflush` instruction exits the store buffer.

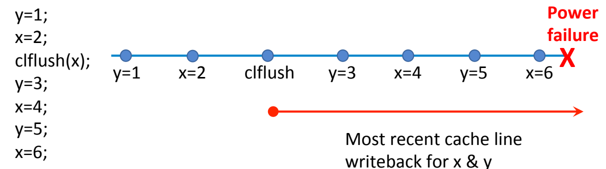

Figure 1: Pre-failure execution of a simple PM program. The blue line represents the total order in which stores are written to the cache. The red line shows the interval for the last time the cache line containing x and y may be written back to PM.

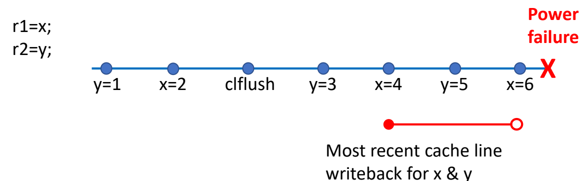

Figure 2: Post-failure execution of the program that reads the value 4 from x. This *refines* the interval for the most recent writeback of the cache line to be between the store $x = 4$ and the store $x = 6$.

Figure 1 shows a pre-failure execution where the program executes the instruction sequence on the left-hand side prior to the failure. The blue line shows the order that stores were written to the cache. Both variables x and y are located in the same cache line. After the program executes the stores $y = 1$

and $\texttt{x} = 2$, it performs a clflush instruction. This instruction flushes the cache line that holds x and y to persistent memory. At this point, Jaaru infers the cache line was most recently flushed during the interval $[\texttt{clflush}, \infty)$ as represented by the red line. Then, the program performs the stores $\texttt{y} = 3$, $\texttt{x} = 4$, $\texttt{y} = 5$, and $\texttt{x} = 6$. Finally, power is lost and the program fails.

There are constraints between the values for variables sharing a cache line. To ensure these variables have consistent values, Jaaru *refines* these intervals using the values observed by loads during the recovery execution. Figure 2 shows a post-failure execution. This execution reads the value 4 for x. Jaaru infers the cache line is flushed after $\texttt{x} = 4$ and before the next store. Thus it refines the interval for the most recent flush to be $[\texttt{x} = 4, \texttt{x} = 6)$. Since x and y resides on the same cache line, Jaaru also can infer cache line was flushed some time after $y = 3$ and after $\texttt{y} = 5$. Therefore, the post-failure execution reads only value 3 or 5 but not $y = 1$ from y. Jaaru uses this refinement-based approach to simulate cache line flushes and lazily construct the state of persistent memory after the failure, eliminating the need to eagerly explore all states.

2.2. Leveraging Commit Stores for Additional Efficiency

Our constraint-refinement approach works well for PM programs because it effectively leverages commit stores to achieve efficiency. Commit stores are a common programming practice. Jaaru does *not* eagerly enumerate all pre-failure stores; instead, Jaaru lazily enumerates a small subset of them that are *actually read* by a post-failure execution (*i.e.*, commit stores).

```
1 void addChild(node *ptr,
2   char * data) {
3   node * tmp = alloc_child();
4   tmp->data = data;
5   clflush(tmp, sizeof(node));
6   ptr->child = tmp;
7   clflush(&ptr->child,
8     sizeof(node *));
9 }
```

```
1 char * readChild(node *ptr) {
2   if (ptr->child != NULL) {
3     return ptr->child->data;
4   }
5   return NULL;
6 }
```

Figure 3: An example program with a commit store.

Figure 3 presents a simple program that uses a commit store where it executes method addChild, fails, and then calls the readChild method during recovery. In the addChild method, the store to the child field is a commit store and indicates the prior store to data field successfully persisted.

Jaaru injects failures in the execution of method addChild at three points: (1) before the clflush at Line 5, (2) before the clflush at Line 8, and (3) at the end of addChild method. Injecting failures at these three points is sufficient to explore all distinct program behaviors. In this example, child field is a commit store. For each commit store, Jaaru explores only two executions at each failure point: (1) an execution that reads from the commit store which would be the case if the child field contains a non-null value and (2) an execution that reads the value of the memory location before the commit store which would be the case if child field contains a null value. Jaaru does not enumerate all possible states at the failure point. It executes the post-failure code and *lazily explores pre-failure stores that are actually read by the post-failure execution*.

3. Evaluation

We evaluated Jaaru on PMDK [1] and RECIPE [4]. Jaaru found 7 bugs in PMDK of which 6 were new. For RECIPE, Jaaru found 18 bugs of which 12 were new. Bugs that Jaaru can identify must have some visible manifestation — either a crash, *e.g.*, segmentation fault, or an assertion failure in the program. Many programs contain multiple bugs. When Jaaru found an execution that causes the program to crash (or loop) we have examined Jaaru's outputted trace and debugging information to understand the bug. For each RECIPE benchmark, we have fixed the bug and used Jaaru to look for additional bugs. In general, Jaaru was able to identify bugs caused by missing flushes, flushing the wrong memory location, and storing data structures in non-volatile memory that program need to be persistent across failures. At the time of writing, 6 out of 12 new bugs found in RECIPE were fixed by the developers. The other 6 bugs were related to memory allocators and garbage collectors. The RECIPE developers did not fix the persistency bugs related to memory allocators because they believe these bugs need to be addressed by the memory allocators, which is not their focus.

Compared to Yat [3], the naïve model checker, Jaaru drastically reduces the number of executions that must be explored. In particular, the space state reduction algorithm enables Jaaru to model check each RECIPE program in less than 15 seconds with an average of 230 executions per benchmark. However, with Yat, exhaustively model checking these programs is infeasible since, for example, Yat requires exploring around 10^{605} number of executions for one of the RECIPE benchmarks.

References

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

[2] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. Jaaru: Efficiently model checking persistent memory programs. https://doi.acm.org/10.1145/3445814.3446735, ASPLOS 2021.

[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, pages 462–477, New York, NY, USA, 2019. Association for Computing Machinery.

[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 25th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 1187–1202, New York, NY, USA, 2020. Association for Computing Machinery.

[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 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 411–425, New York, NY, USA, 2019. Association for Computing Machinery.