PSan: Checking Robustness to Weak Persistency Models

Hamed Gorjiara Weiyu Luo Alex Lee
University of California, Irvine

Guoqing Harry Xu
University of California, Los Angeles

Brian Demsky
University of California, Irvine

1. Introduction

Bugs in the uses of flush and drain operations can be trivially eliminated by persisting stores in the same order that they become visible to other threads. Strict persistency [8] is such a persistency model that ensures that the "persistency memory order is identical to volatile memory order". Most hardware persistent memory specifications do not provide strict persistency. As a result, PM developers must explicitly use *flush instructions* to ensure that program executions under weak persistency semantics are correct. *Our key observation is that the typical correct usage of flush instructions in PM programs ensure that program executions under weak persistency semantics are equivalent to those under strict persistency semantics.* Building on this observation, we define a new notion of correctness, *robustness*, for programs under weak persistency in terms of their equivalence to post-crash executions under strict persistency. A program is robust to a weak persistency model if, for any crash events, each post-crash execution of the program under that weak persistency model is equivalent to some post-crash execution after some crash event under the strict persistency model. Robustness is a *sufficient criterion* to assure correct usage of flush and drain operations—adding more flush and drain operations to a robust program will not alter the set of possible post-crash executions.

In general, robustness does *not* require a developer to insert flush operations immediately after every store. Figure 1 shows an example on the x86 persistency model. Suppose that execution of the `addChild` method crashes immediately before line 6 and that after the crash the program executes the `readChild` method on the same node. There are two possible post-crash executions: (1) the post-crash execution that results from the pre-crash execution where the store of the reference to the `child` field was flushed, and (2) the post-crash execution that results from the pre-crash execution where the store of the reference was *not* flushed. The first post-crash execution is equivalent to the post-crash execution under strict persistency where the pre-crash execution crashes before the store in line 5. The second post-crash execution is equivalent to the post-crash execution under strict persistency where the pre-crash execution crashes after the store in line 5. Since all post-crash executions of this program under the weak persistency model are equivalent to some post-crash execution under strict persistency, this example program execution is robust.

2. PSan

We develop PSan [3], a tool that dynamically checks robustness for programs under the x86 persistency model and reports

```
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 1: An example of execution being robust to the x86 persistency model.

violations in a fully automated fashion. For a given execution, PSan can detect *all persistency bugs due to ordering issues* in that execution; finding other types of bugs (such as concurrency bugs) is not our focus. Given a crash event and a post-crash execution, PSan computes a set of strictly persistent executions whose pre-crash executions are consistent with the given post-crash execution. If such strictly persistent executions do not exist, PSan finds a robustness violation.

Our key insight is that we can efficiently compute this set of consistent pre-crash executions under strict persistency by *reasoning about the interval in which an equivalent strictly persistent pre-crash execution must have crashed using constraints*. In particular, each load in the post-crash execution that reads from a store s in the pre-crash execution under the x86 persistency model constrains where an equivalent strictly persistent execution may crash—the crash point must be somewhere between the store s and the next store to the same memory location. If this set of constraints is unsatisfiable, there is no equivalent strictly persistent execution.

```
1   x = 1;
2   y = 1;
3   x = 2;
4   y = 2;
```

Pre-crash execution

```
1   r1 = x;
2   r2 = y;
```

Post-crash execution.

Figure 2: A weakly-persistent single-threaded execution that reads `r1 = 1` and `r2 = 2` is not robust.

To illustrate, consider the executions in Figure 2, which shows a single-threaded program executed under a weak persistency model. If `r1 = 1`, we know that an equivalent strictly persistent execution must have crashed after the assignment `x = 1` but before the assignment `x = 2`. If `r2 = 2`, then an equivalent strictly persistent execution must have crashed after the assignment `y = 2`. These two constraints are not simultaneously satisfiable, and therefore this execution is *not* robust.

To support multi-threaded programs, the key idea is that PSan determines whether there is an equivalent trace that can be produced by selecting different (but compatible) crash points for different threads. Our idea for implementing this is to have the robustness analysis compute per-thread crash

intervals and ensure that these intervals describe a prefix of the pre-crash execution that is closed under happens-before.

2.1. Suggesting Fixes for Robustness Violations

Robustness enables PSan to infer the location of a missing flush or drain operation. Each robustness violation involves an earlier store that was not made persistent and a later store that was made persistent—the earlier store is missing a flush operation. For instance, for the execution in Figure 2, PSan determines a flush instruction must be inserted after `x = 2` to fix the robustness violation.

In general, there are two ways to fix a robustness violation. The first is to use flush and/or drain operations to force the cache to write a cache line to persistent memory. The second is to leverage the existing cache coherence mechanism to enforce the desired ordering by locating a pair of stores for which an ordering violation is observed on the same cache line.

There are two cases in which a robustness violation may be reported: (1) the most recent load reads from a store that is too old to be consistent with the strict persistency model and (2) the most recent load reads from a store that is too new.

Reading from Too Old of Store. Figure 3-a presents a robustness violation that occurs when the most recent load $ld\langle y\rangle$ reads from a store $st_1\langle y\rangle$ that is too old. This occurs because the program is missing a flush on some newer store $st_2\langle y\rangle$ to the same memory location. Our algorithm detects this when the presence of the later store $st_2\langle y\rangle$ causes the algorithm to move the end of the crash interval backward past the beginning of the interval.

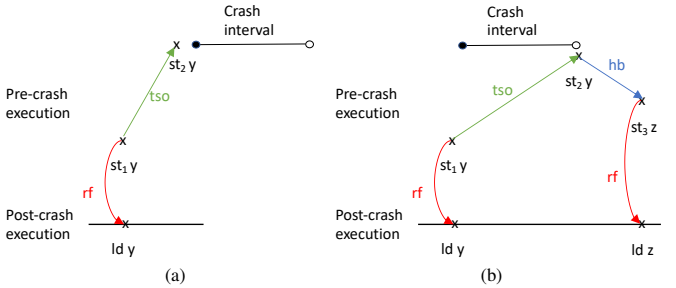

Figure 3: (a) Reading from a store that is too old. (b) Reading from a store that is too new.

The fix for this bug is to insert a flush and a drain that happen after the store $st_2\langle y\rangle$ and happen before the beginning of some crash interval.

Reading from Too New of Store. Figure 3-b presents an execution in which the most recent load $ld\langle z\rangle$ reads from a store $st_3\langle z\rangle$ that is too new to be consistent with the strict persistency model. This occurs because a previous load $ld\langle y\rangle$ reads from a store that was too old since some store $st_2\langle y\rangle$ was missing an appropriate flush operation. Our algorithm detects this violation when the store $st_3\langle z\rangle$ causes the beginning of the crash interval to be move forward past the end of the crash interval.

The fix for this bug is to insert a flush and a drain operation such that $st_2\langle y\rangle$ happens before the flush and drain operation and the flush and drain operation happens before $st_3\langle z\rangle$.

3. Evaluation

We evaluated PSan on a set of data structures including RECIPE [6], CCEH [7], and FAST_FAIR [4]. We also evaluated PSan on three popular real-world frameworks: PMDK [1], Memcached [2], and Redis [5]. Each program has a test driver that performs operations on the data structure. PSan supports two exploration strategies: (1) a random search mode where explores random executions with random crash points and (2) a model checking mode where it systematically inserts crashes before each fence-like operation and explores all potential load values. We evaluated all benchmarks with both strategies except Memcached and Redis that we used random strategy since they require an outside client.

During our experiment, PSan found a total of 48 bugs in benchmarks of which 17 of them were not reported by any prior testing tools. 13 bugs were related to robustness violations in the memory management code of the benchmarks. We reported these violations to the developer of these tools and so far, developers of CCEH and FAST_FAIR have confirmed these violations are real bugs. For each of these violations, PSan reports the variable that needs a flush instruction and the precise range where the flush needs to be inserted. These violations are cases of missing flushes/fences, cache line non-alignment, and memory management bugs which could cause data corruption, data loss, or memory leak. Our evaluation shows PSan introduces minimal overhead of on average 0.001s per execution for each benchmark compared to Jaaru the underlying model checker.

References

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

[2] Inc. Danga Interactive. Memcached. https://github.com/lenovo/memcached-pmem, November 2018.

[3] Hamed Gorjiara, Weiyu Luo, Alex Lee, Guoqing Harry Xu, and Brian Demsky. Checking robustness to weak persistency models. http://plrg.ics.uci.edu/wordpress/wp-content/uploads/2022/03/psan-pldi22.pdf, Conditionally Accepted to PLDI 2022.

[4] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable transient inconsistency in byte-addressable persistent B+-Tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST '18, pages 187–200, USA, 2018. USENIX Association.

[5] Redis Labs. Redis. https://github.com/pmem/redis, August 2020.

[6] 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.

[7] Moohyeon Nam, Hokeun Cha, Young-Ri Choi, Sam H. Noh, and Beomseok Nam. Write-optimized dynamic hashing for persistent memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, FAST '19, pages 31–44, USA, 2019. USENIX Association.

[8] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 265–276, Minneapolis, MN, USA, 2014. Institute of Electrical and Electronics Engineers.