Yashme: Detecting Persistency Race

Hamed Gorjiara
University of California, Irvine

Guoqing Harry Xu
University of California, Los Angeles

Brian Demsky
University of California, Irvine

1. Introduction

Failures are a key challenge in redesigning systems to use persistent memory. We present a new class of persistent memory bugs that we call *persistency races*. Persistency races stem from the fact that most programming language specifications provide compilers with the freedom to assume that non-atomic stores do not participate in data races. This means that compilers can perform optimizations that assume that the stored locations are not accessed concurrently by other threads. Such optimizations include implementing a non-atomic store with multiple store instructions. This is often referred to as *store tearing*. For example, given an architecture having 16-bit store instructions with immediate fields, the compiler might be tempted to use two 16-bit store-immediate instructions to implement a 32-bit store. Although it is rare that compilers introduce these optimizations, it is enough of a concern that **both PMDK developers and the Linux Kernel developers take care to avoid it**. Indeed, the Linux kernel mailing list provides a number of examples [3] of modern compilers tearing non-atomic stores even when they are aligned, word-length stores. Compilers can also introduce store tearing via other optimizations. Mainstream compilers commonly rewrite code that copies or initializes several contiguous fields into calls to the libc functions `memcpy`, `memmove`, or `memset`. These optimizations are very common in practice. These functions do not guarantee 64-bit atomicity and can hence result in store tearing. Crashes in persistent memory systems can make the effects of these optimizations visible. For example, store tearing creates the possibility that a poorly timed crash can cause non-atomic stores to be made partially persistent. A post-crash execution can then potentially read values that mix bytes from multiple different store operations. This could, for example, cause a post-crash execution to read an invalid array index, leading to further corruption.

Persistency races are similar in spirit to data races because both persistency races and data races violate assumptions made by compilers and thus can break the abstraction of a language-level store writing the specified value to memory. However, there are important fundamental differences between the two as each persistency race involves three distinct events: (1) the racing store in the pre-crash execution, (2) the crash event against which the store races, and (3) a race-observing load in the post-crash execution that observes the effects of the race. Although persistency races may not manifest under a particular compiler/architecture, they can lead to bugs that are extremely difficult to detect during development and testing (*e.g.*, exposing persistency race due to compiler update).

Most existing PM bug finding tools use techniques that fundamentally cannot detect persistency races because they just validate that stores are flushed or performed in a specific order. Only model checking tools could conceptually be adapted to find persistency races by splitting the stores into single byte stores at the cost of an exponential increase in the number of executions that must be explored. However, if these frameworks explore the correct execution, they can potentially observe a crash caused by a persistency race.

2. Yashme

We implemented a tool [4], Yashme, to detect persistency races. Yashme's basic approach is to simulate the execution of a PM program, inject a crash, and then simulate the execution of the post-crash recovery program. During the post-crash execution, we compute *which stores may have persisted incorrect values due to the crash*. There are two ways that PM program executions can ensure that stores are fully persisted: (1) the execution explicitly flushes the cache line *after* the store writes to the cache line and *before* the crash or (2) the post-crash execution reads from a later atomic store to the same cache line and relies on *cache coherence* to ensure the persistency of the store.

Figure 1 presents an example where the pre-crash execution stores x=1, and then persists it by executing a `clflush` instruction. To ensure persistency, it is critical that the store *happens before* the clflush instruction. This execution does not expose a persistency race because x=1 has been flushed before the crash.

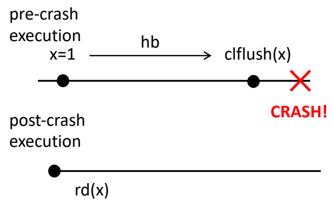
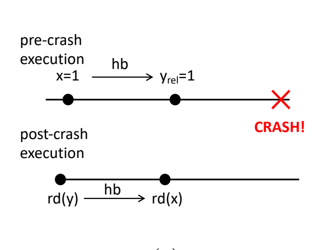
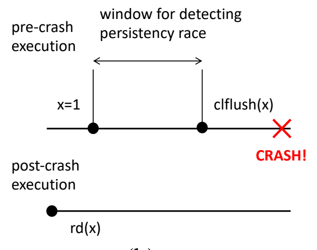


Figure 1: Example of using `clflush` to flush the store to x.

Cache coherence protocols ensure a total order in the persistence of stores to the same cache line. Figure 2(a) provides an example of an execution that uses cache coherence to avoid a persistency race. We use the notation $\texttt{y}_{\texttt{rel}}\texttt{=1}$ to indicate that the store of 1 to `y` is an atomic release store. Assume that the variables `x` and `y` reside on the same cache line. In the pre-crash execution, the store to `x` happens before the store to `y`. Cache coherence protocols guarantee `x=1` is completely written to the cache line before $\texttt{y}_{\texttt{rel}}\texttt{=1}$ even if store to `x` is torn into multiple store operations. Since the post-crash execution observes the store to `y`, the cache line is flushed sometime after persisting $\texttt{y}_{\texttt{rel}}\texttt{=1}$ and before the crash event. Consequently, the post-crash execution must also observe the fully completed store to x due to cache coherence. Thus, there is no persistency race in this execution.



(a)

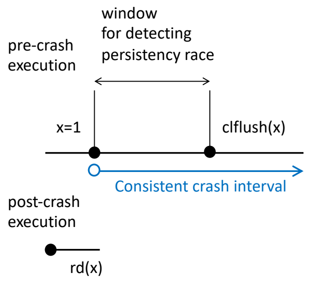


(b)

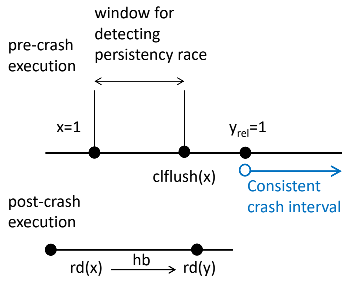
Figure 2: (a) Example of coherence preventing persistency races. Variables x and y reside on the same cache line and that the store to y is an atomic release store. (b) Crash misses window for detecting persistency race using core algorithm.

2.1. Key Idea: Expanding the Detection Window

The two approaches mentioned above can only detect persistency races involving stores in *a small window of the pre-crash execution*. Figure 2(b) shows an example crash scenario to illustrate this problem. In this example, the pre-crash execution writes to x, flushes the write, and then crashes. The post-crash execution then reads from x. Since the crash occurs after the write is flushed, the approach misses detecting the persistency race in this program. To detect this persistency race, the program must crash in the small window of time between the store to x and the corresponding flush. This implies that detecting races using the basic approaches would require injecting crashes in a large number of executions, which can be prohibitively expensive for large programs.



(a)



(b)

Figure 3: (a) Prefixes of pre-crash execution that are consistent with the post-crash execution (b) Prefixes of pre-crash execution that are consistent with the post-crash execution after reading from y residing on the same cache line as x.

Our key insight for effectively detecting persistency races is that we can check whether the post-crash execution E' has a persistency race with *any prefix* E^+ *of the pre-crash execution* E *that is consistent with* E'. Figure 3(a) illustrates this insight. While the pre-crash execution has flushed the store to x, the post-crash execution has not read from any store that happens after the cache line flush. Thus the post-crash execution at this point is *consistent with any prefix* of the pre-crash execution starting at the store that writes 1 to x. The blue arrow shows the range of consistent prefixes of the pre-crash execution. In Figure 3(b), as the post-crash execution reads from the atomic variable y, Yashme updates the constraint of pre-crash execution to be *consistent* with the post-crash execution and to include the clflush(x) instruction.

Consistent Prefixes. Intuitively, a consistent prefix of the pre-crash execution must contain any statement which happens before a pre-crash store that the post-crash execution reads from. Yashme tracks every pre-crash store that the post-crash executions reads from and computes the shortest consistent prefix by using clock-vector-based techniques that are commonly used by race detectors. Yashme uses the consistent prefix to determine whether there is a prefix of the pre-crash execution that did not execute a given `clflush`. If so, Yashme ignores the instruction when checking for races, because there is a pre-crash execution that does not execute the instruction and yields the same post-crash execution.

3. Evaluation

We have evaluated Yashme on a collection of persistent data structures including the RECIPE benchmarks [7], FAST_FAIR [5], and CCEH [8] in its model checking mode. We tested Yashme on real-world frameworks including PMDK [1], Memcached [2], and Redis [6] in random mode. For each of these benchmarks, we used example programs manipulate data store through standard insertion, deletion, and lookup operations. Yashme has found 24 persistency races in these benchmarks that are *all new and have not been discovered by prior tools*. Most of these bugs are confirmed by the developers of these tools. To fix these bugs, the developers need to replace racing non-atomic stores with atomic ones

To evaluate the importance of searching for persistency races in prefixes of available executions, we ran Yashme with and without this optimization to compare their bug finding capabilities. Yashme finds **5**× more persistency races with the prefix optimization.

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] Will Deacon. Re: [patch 1/1] fix: trace sched switch start/stop racy updates. https://lore.kernel.org/lkml/20190821103200.kpufwtviqhpbuv2n@willie-the-truck/, August 2019.

[4] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. Yashme: Detecting persistency races. https://doi.org/10.1145/3503222.3507766, ASPLOS 2022.

[5] 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.

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

[7] 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.

[8] 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.