Systematic Testing of Persistent Memory Programs

Henrique Fernandes, João Gonçalves, Miguel Matos
IST Lisbon & INESC-ID

Abstract

Persistent memory (PM) offers performance comparable to DRAM and non-volatility. However, writing correct and performant PM programs is difficult as witnessed by the large number of tools to find bugs in PM applications. These tools offer different coverage and performance trade-offs but the underlying challenge is the combinatorial explosion of possible states that can be observed in a post-crash execution. Notably, most existing tools explore only a small fraction of the possible state space, limiting their coverage.

We argue that it is possible to perform a broader, systematic exploration of the search space in useful time, and with that provide greater bug coverage. In this work we present a methodology to achieve this goal, based on the semantics of the hardware, deduplication of crash states, semantic equivalence of crash states, and parallelization.

1 Context and Challenges

Persistent Memory (PM) combines near DRAM speeds with the persistency of flash storage. For performance, the CPU can bypass the kernel and do accesses directly via load and store instructions. Moreover, stores can be buffered and persisted in an order that differs from the order they were issued by the program. This poses a challenge as the system can crash at any time, and the persisted state might not be consistent with the program's semantics.

To mitigate this problem, programmers can control the persistency order by issuing `flush` and `fence` instructions [12]. However, research shows that these instructions are often misused, resulting in a wide range of PM-related bugs [1, 3–11]. The use of higher-level libraries [2, 13] alleviates this tension but, as expected, does not fully solve the issue as the libraries can be misused or contain bugs themselves [4, 6, 14]. In practice, crashes that lead to inconsistent states are expected and, similarly to what is done in domains such as databases or filesystems, developers must provide a recovery procedure that is executed after a crash and brings the application to a consistent state.

To deal with the prevalence of PM bugs, researchers have proposed various tools for PM bug detection that offer different coverage, speed, and general applicability trade-offs [1, 3–11]. These tools are effective at catching PM bugs, and some are even efficient enough to promise integration with the development workflow to catch PM bugs early on [6]. However, most of these tools share a common limitation: they explore only a small fraction of the possible state space, limiting coverage. The reason behind this is that the number of possible store reorderings and respective post-crash states is combinatorial in nature, resulting in a search space that is unfeasible to explore exhaustively even for small programs [8]. Because of that, existing tools reduce the search space using heuristics [6, 7], skewing the exploration based on particular application semantics that compromise generality [1, 4, 5], or by heavily relying on developer input [3, 9–11]. The exception is Yat [8] which performs an exhaustive search of all states but, as expected, is too slow to be practical [8].

The current state of affairs is to either explore a small fraction of the state space, compromising coverage, or to perform an exhaustive search that is too slow to be practical. We argue that this does not need to be the case. Although a naive exploration of the state space is indeed unfeasible, we believe it is possible to perform a broader, systematic exploration of the search space in useful time, and with that provide greater bug coverage. Notably, we aim to do so automatically, without requiring developer input, and without compromising the generality of the approach.

2 Proposal

We propose a systematic approach to explore the state space of PM applications in a broad and yet efficient manner. In essence, the testing pipeline is comprised of two stages: (1) executing the target binary with a given workload, capturing PM accesses and generating possible crash states; and (2) validating the consistency of all crash states. As explained in §1, the main challenge is how to sift through all possible states, which range upwards of thousands of million for many tested applications, in an efficient manner. Our proposal is based on four main ideas: (1) hardware semantic-aware state creation, (2) state deduplication, (3) state semantic equivalence, and (4) parallelization.

Hardware semantic-aware state creation. At the base of our approach is the notion of a segment — a section of the program between any two fences. Since a fence ensures persistence ordering guarantees, at the end of each segment we can enumerate all possible orderings that can be observed in a post-crash execution. However, doing so naively not only results in a combinatorial explosion of states but also produces many invalid states, i.e. states that can never be observed due to the hardware characteristics. Intel x86 processors guarantee that stores to the same cache line reach the cache in program order, which in turn ensures they reach PM in the same order. Also, flush instructions have different semantics. Whereas `clwb` and `clflushopt` can be reordered between each other and with subsequent stores, `clflush` preserves program order. This means these instructions impose additional ordering constraints that reduce the search space.

In turn, this allows to prune out many invalid states but it is still not enough to make the search space manageable.

State deduplication. A naive generation of crash states will produce many redundant states. To tackle this, we use two forms of deduplication. First, we noticed that, although stores may reach PM in many different orders, we only care about the set of stores that reach the PM device for each crash-state. For example, if a program issues stores *A*, *B*, and *C*, all to different cache lines, this generates 16 different reoderings, ranging from the empty set {} to *ABC*. However, many of these reorderings produce equivalent states, such as *AB* and *BA*, or *ABC* and *CBA*. By considering store reorderings as sets instead of sequences, we considerably reduce the number of states generated. However, maintaining these sets for entire program is expensive. Although we apply this logic for each segment, equivalent states can be generated across segments. Since crash states typically differ from each other only by a small number of stores, our second deduplication method is to use merkle trees to efficiently prune equivalent states.

State semantic equivalence. State deduplication is a powerful tool to reduce the number of states generated, but it is still not enough to make the search space manageable. Based on this we made the observation that, although two states may differ in the order of stores, they may still be semantically-equivalent. For example, consider an PM hash table that resizes itself after N elements are inserted. Observe that under the simplifying assumption that all elements are unique and there are no collisions, after the first insertion, and until the resize, analyzing this program would generate N-1 different yet semantically-equivalent states from a crash-consistency perspective. In detail, the resize is only triggered after the Nth insertion which means that it is only at this point that the execution will explore a new code path and the corresponding crash state. Based on this observation we introduce the notion of unique segments. For each segment, we collect the sequence of addresses of all instruction executed (PM or otherwise). At the end of the segment, and before generating the crash states, we hash this sequence to determine if we executed it before, in which case it is ignored and no crash-state if generated.

Parallelization. Finally, we parallelize the process of generating crash-states and recovering from them. During the first stage, we intrument the target program to capture PM accesses and generate crash states for each segment by applying each reordering to the initial segment state and then copying the resulting state. The majority of the time is spent on I/O operations, we perform an initial run to count the number of segments and then parallelize the generation of crash states, evenly distributing the work across instrumentation processes. For the validation stage, we take inspiration from existing tools and use the recovery procedure as a correctness oracle [6]. This is a cheap and effective mechanism to assess the correctness of a state and allows our approach to remain agnostic of the target application semantics. Since each recovery procedure is completely independent, we have a pool of recovery threads that pick crash states to recover as soon as they become available. By decoupling the instrumentation and recovery stages and executing both in parallel, we are able to achieve substantial speedups in testing time and reduce the total disk space required for testing, as crash states are deleted after recovery is performed.

Preliminary results and challenges. Our approach reduces the search space by several orders of magnitude, enabling program analysis in hours rather than days or years [8], with further gains possible as we fine tune the strategies described above. However, challenges remain. First, some segments generate prohibitively many states due to a large number unordered store operations. While heuristics exist [8], we plan to explore a systematic solutions. Second, non-determinism impacts our parallelization strategies for state generation and semantic equivalence, such as in cases of thread interleavings. Disabling these components increases testing time significantly, thus we are exploring solutions.

Acknowledgments

This work was supported by Fundação para a Ciência e a Tecnologia under grants UIDB/50021/2020, 2021.07401.BD, and PTDC/CCI-COM/4485/2021 (Ainur); by the European Union's Horizon 2020 R&I programme under grant agreement No 952226, project BIG; and by Composable Foundation under project ScalableCosmosConsensus.

References

[1] Zhangyu Chen et al. 2022. Efficiently detecting concurrency bugs in persistent memory programs. In *ASPLOS '22*.

[2] Intel Corporation. [n.d.]. The libpmemobj library. https://pmem.io/pmdk/libpmemobj/.

[3] Bang Di et al. 2021. Fast, flexible, and comprehensive bug detection for persistent memory programs. In *ASPLOS '21*.

[4] Xinwei Fu et al. 2021. Witcher: Systematic Crash Consistency Testing for Non-Volatile Memory Key-Value Stores. In *SOSP '21*.

[5] Xinwei Fu et al. 2022. DURINN: Adversarial Memory and Thread Interleaving for Detecting Durable Linearizability Bugs. In *OSDI '22*.

[6] João Gonçalves et al. 2023. Mumak: Efficient and Black-Box Bug Detection for Persistent Memory. In *EuroSys '23*.

[7] Hamed Gorjiara et al. 2021. Jaaru: Efficiently Model Checking Persistent Memory Programs. In *ASPLOS '21*.

[8] Philip Lantz et al. 2014. Yat: A Validation Framework for Persistent Memory Software. In *USENIX ATC '14*.

[9] Sihang Liu et al. 2019. PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs. In *ASPLOS '19*.

[10] Sihang Liu et al. 2020. Cross-Failure Bug Detection in Persistent Memory Programs. In *ASPLOS '20*.

[11] Ian Neal and others". 2020. AGAMOTTO: How Persistent is your Persistent Memory Application?. In *OSDI '20*.

[12] Azalea Raad et al. 2019. Persistency semantics of the Intel-x86 architecture. *Proc. ACM Program. Lang.* (2019).

[13] Haosen Wen et al. 2021. A Fast, General System for Buffered Persistent Data Structures. In *ICPP '21*.

[14] Duo Zhang et al. 2021. A study of persistent memory bugs in the Linux kernel. In *SYSTOR '21*.