Snapshot: Fast, Userspace Crash Consistency Using msync

Suyash Mahar*, Mingyao Shen*, Terence Kelly†, Steven Swanson*
*UC San Diego †No affiliation

1. Introduction

Crash consistency using persistent memory programming libraries requires programmers to use complex transactions and manual annotations. In contrast, the failure-atomic `msync()` [10] (FAMS) interface is much simpler as it transparently tracks updates and guarantees that modified data is atomically durable on call to the failure-atomic variant of `msync()`. However, FAMS suffers from several drawbacks, like the overhead of the `msync()` system call and the write amplification from page-level dirty data tracking.

To address these drawbacks while preserving the advantages of FAMS, we propose Snapshot, an efficient userspace implementation of FAMS. Snapshot uses novel, compiler-based instrumentation to transparently track updates in userspace and syncs them with the backing persistent memory copy on a call to `msync()`. By keeping a copy in DRAM, Snapshot improves access latency. Moreover, with automatic tracking and syncing changes only on a call to `msync()`, Snapshot provides crash-consistency guarantees, unlike the POSIX `msync()` system call.

For a KV-Store backed by Intel Optane running the YCSB benchmark, Snapshot achieves at least 1.2× speedup over PMDK while significantly outperforming non-crash-consistent `msync()`. On an emulated CXL memory semantic SSD, Snapshot significantly outperforms PMDK by up to 10.9× on all but one YCSB workload, where PMDK is 1.2× faster than Snapshot. Further, Kyoto Cabinet commits perform up to 8.0× faster with Snapshot than its built-in, `msync()`-based crash-consistency mechanism on Intel Optane DC-PMM.

2. Background and Motivation

Recent memory technologies like CXL-based memory semantic SSDs [3], NV-DIMMs [11], Intel Optane DC-PMM [6], and embedded non-volatile memories [5] have enabled byte-level, non-volatile storage devices. However, achieving crash consistency on these memory technologies often requires complex programming interfaces. Programmers must atomically update persistent data using failure-atomic transactions and carefully annotated LOAD and STORE operations, significantly increasing programming complexity [8, 9, 7].

The `msync()` system call offers a simpler interface for durability. The programmer maps a file from persistent media into virtual memory and calls `msync()` to make any changes durable.

The `msync()` interface, however, makes no crash-consistency guarantees. The OS is free to evict dirty pages from the page cache before the application calls `msync()`. A common workaround to this problem is to implement a write-ahead-log [1, 4] (WAL) which allows recovery from an inconsistent state after a failure. However, crash consistency with WAL requires an application to call multiple `msync()`s to ensure the data is always recoverable after a crash.

Park et al. [10] overcome this limitation by enabling failure-atomicity for the `msync()` system call. Their implementation, FAMS (failure atomic `msync()`), holds off updates to the backing media until the application calls `msync()` and then leverages filesystem journaling to apply them atomically. FAMS is implemented within the kernel and relies on the OS to track dirty data in the page cache.

OS-based implementation, however, suffers from several limitations:

(a) Write-amplification on `msync()`: The OS tracks dirty data at the page granularity, requiring a full page write-back even for a single-byte update, wasting memory bandwidth on byte-addressable persistent devices. Using 2 MiB huge pages to reduce TLB pressure exacerbates this problem.

(b) Dirty page tracking overhead: FAMS relies on the page table to track dirty pages, thus every `msync()` requires an expensive page table scan to find dirty pages to write to the backing media. Moreover, since the OS is responsible for maintaining TLB coherency, the kernel must perform a TLB flush after clearing the access and dirty bits [2], adding significant overhead to every `msync()` call.

(c) Context switch overheads: Implementing crash consistency in the kernel (e.g., FAMS) adds context switch overhead to every `msync()` call, compounding the already high overhead of tracking dirty pages in current implementations.

3. Snapshot

We address the shortcomings of FAMS with Snapshot, a drop-in, userspace implementation of failure atomic `msync()`. Snapshot transparently logs updates to memory-mapped files using compiler-generated instrumentation, implementing fast, fine-grained crash consistency. Snapshot tracks all updates in userspace and does not require switching to the kernel to update the backing media.

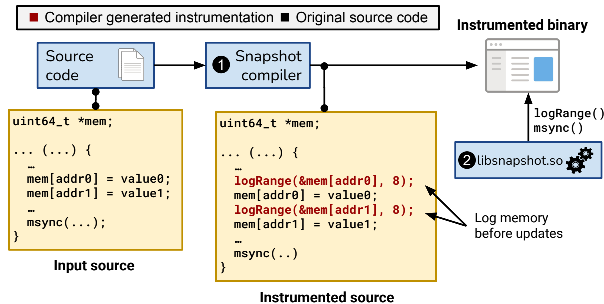

Figure 1: Snapshot compilation process.

Figure 1 shows the overview of Snapshot's compilation and runtime. Snapshot works by logging STOREs transparently (➊) and makes updates durable on the next call to `msync()`. During runtime (➋), the function checks whether the store is to a persistent file and logs the data in an undo log.

Snapshot's ability to automatically track modified data allows applications to be crash-consistent using `msync()` without significant programmer effort. For example, Snapshot's automatic logging enables crash consistency for volatile data structures, like shared-memory allocators, with low-performance overhead.

Snapshot makes the following key contributions:

(a) Low overhead dirty data tracking for `msync()`. Snapshot provides fast, userspace-based dirty data tracking and avoids write-amplification of the traditional `msync()`.

(b) Accelerating applications on byte-addressable storage devices. Snapshot allows porting existing `msync()`-based crash-consistent applications to persistent, byte-addressable storage devices with little effort (e.g., disabling WAL-based logging) and achieves significant speedup.

(c) Implementation space exploration for fast write-back. We perform a detailed study on the performance of NT-stores, `clwb`, and `sfence` and their interactions with each other. We use the results to tune Snapshot's implementation and achieve better performance. These results are general and can help accelerate other crash-consistent applications.

4. Results

We compared Snapshot's performance against PMDK and conventional `msync()` (as FAMS is not open-sourced) using Intel Optane DC-PMM and emulated memory semantic SSDs. We emulate memory semantic SSDs with a large DRAM cache backed by a block device using a two-socket server with one socket running the workload and the other socket running the cache using shared memory and an SSD as a block device.

On Intel's Optane DC-PMM, for b-tree insert and delete workloads, Snapshot performs as well as PMDK

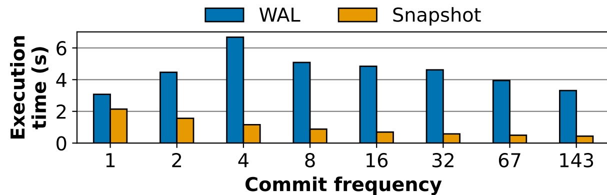

Figure 2: Performance comparison of commit frequency for writes in Kyoto Cabinet. Lower is better.

and outperforms it on the read workload by 4.1×. Moreover, Snapshot outperforms non-crash-consistent `msync()` based implementation by 2.8× with 4 KiB page size and 463.8× with 2 MiB page size for inserts. For KV-Store, Snapshot outperforms PMDK by up to 2.2× on Intel Optane and up to 10.9× on emulated memory semantic SSD.

Finally, Snapshot performs as fast as and up to 8.0× faster than Kyoto Cabinet's custom crash-consistency implementation (Figure 2).

5. Conclusion

Snapshot provides a userspace implementation of failure atomic `msync()` (FAMS) that overcomes its performance limitation. Snapshot's sub-page granularity dirty data tracking based crash-consistency out-performs both per-page tracking of `msync()` and manual annotation-based transactions of PMDK across several workloads.

References

[1] PostgreSQL, 2022. https://www.postgresql.org/.

[2] Nadav Amit. Optimizing the TLB shootdown algorithm with page access tracking. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 27–39, 2017.

[3] Samsung Electronics. Samsung electronics unveils far-reaching, next-generation memory solutions at flash memory summit 2022. https://news.samsung.com/global/samsung-electronics-unveils-far-reaching-next-generation-memory-solutions-at-flash-memory-summit-2022.

[4] FAL Labs. Kyoto Cabinet: a straightforward implementation of DBM, 2010. http://fallabs.com/kyotocabinet/.

[5] Crossbar Inc. Rethink embedded memory with ReRAM. https://www.crossbar-inc.com/products/high-performance-memory/.

[6] Intel. Intel Optane Memory, 2017. http://www.intel.com/content/www/us/en/architecture-and-technology/optane-memory.html.

[7] Sihang Liu, Suyash Mahar, Baishakhi Ray, and Samira Khan. Pmfuzz: test case generation for persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 487–502, 2021.

[8] 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*, pages 1187–1202, 2020.

[9] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. AGAMOTTO: How persistent is your persistent memory application? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1047–1064, 2020.

[10] Stan Park, Terence Kelly, and Kai Shen. Failure-atomic msync(): A simple and efficient mechanism for preserving the integrity of durable data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, page 225–238. Association for Computing Machinery, 2013.

[11] Viking Technologies. DDR4 NVDIMM.