### Title

Snapshot: Fast, Userspace Crash Consistency for CXL and PM Using msync




### Authors

Mahar, Suyash

Shen, Mingyao

Kelly, Terence

et al.



### Publication Date

2023-11-08

### DOI

10.1109/iccd58817.2023.00082

# Snapshot: Fast, Userspace Crash Consistency Using msync

Suyash Mahar
*UC San Diego*
San Diego, USA

Mingyao Shen
*UC San Diego*
San Diego, USA

Terence Kelly
*Independent*
USA

Steven Swanson
*UC San Diego*
San Diego, USA

***Abstract*—Crash consistency using persistent memory programming libraries requires programmers to use complex transactions and manual annotations. In contrast, the failure-atomic `msync()` (FAMS) interface is much simpler as it transparently tracks updates and guarantees that modified data is atomically durable on a call to the failure-atomic variant of `msync()`. However, FAMS suffers from several drawbacks, like the overhead of `msync()` 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 compiler-based annotation to transparently track updates in userspace and syncs them with the backing persistent memory copy on a call to `msync()`. By keeping a copy of application data 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 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.**

## I. Introduction

Recent memory technologies like CXL-based memory semantic SSDs [1], NV-DIMMs [2], Intel Optane DC-PMM, and embedded non-volatile memories [3] 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 [4].

The `msync()` system call offers a simpler interface for durability. The programmer maps a file from the persistent media into the 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 [5]–[7] (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. [8] 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. These limitations are:

*(a) Write-amplification on `msync()`:* The OS tracks dirty data at page granularity, requiring a full page writeback 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 [9], 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.

In this paper, 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.

Snapshot works by logging STOREs transparently and makes updates durable on the next call to `msync()`. During runtime, the instrumentation 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 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 enables porting of existing `msync()`-based crash consistent applications to persistent, byte-

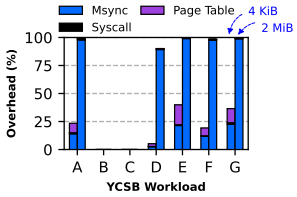



Fig. 1: `msync()`-based KV-Store perf. breakdown (4 KiB and 2 MiB pages).

TABLE I: Byte-addressable, persistent storage devices.

| Device | Interface | Hierarchy |
|---|---|---|
| Optane PM | Mem. Bus | PM & Internal caches [10] |
| Mem. Semantic SSDs [1] | CXL | Flash + Large DRAM cache |
| NV-DIMMs [2] | Mem. bus | DRAM |
| Embedded NVM [3] | Internal Bus | ReRAM |

addressable storage devices with little effort (e.g., disabling WAL-based logging) and achieves significant speedup.

**(c) Implementation space exploration for fast writeback.** We study the latency characteristics of NT-stores and `clwb`s and use the results to tune Snapshot's implementation and achieve better performance. These results are general and can help accelerate other crash-consistent applications.

We compared Snapshot against PMDK and conventional `msync()` (as FAMS is not open-sourced) using Optane DC-PMM and emulated memory semantic SSDs (DRAM backed by flash media over CXL [1]). For b-tree insert and delete workloads running on Intel Optane DC-PMM, Snapshot performs as well as PMDK 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.

## II. Background and Motivation

To understand how the `msync()`-based programming interface can work on persistent devices, the following section presents a brief survey of byte-addressable storage devices. This is followed by a discussion on the `msync()`-based crash-consistency interface. Finally, we discuss an existing implementation of crash-consistent `msync()`, FAMS, and its limitations.

### A. Byte-addressable Storage Devices

Recent advances in memory technology and device architecture have enabled a host of storage devices that support byte-level persistence. These devices communicate with the host using interfaces like CXL.mem [11], DDR-T [12], or DDR-4 [2], rely on flash, 3D-XPoint, or DRAM as their backing media, and have varying access characteristics, as shown in Table I.

These devices share a few common characteristics: (1) they offer byte-level access to persistent data, (2) they require special instructions (e.g., cache-line flush) to ensure persistence, and (3) they are generally slower than DRAM. Later, in Section III, we will explain how Snapshot takes advantage of these properties of emerging memories to implement a fast, userspace-based `msync()`.

```
pool = pmemobj_create(..)

void append(u64 *arr,
     u64 val) {
TX_BEGIN {
  TX_ADD(arr[arr_sz]);
  TX_ADD(arr_sz);

  arr[arr_sz++] = val;
} TX_END;
}
```

(a) PMDK

```
mem = mmap(...)

void append(u64 *arr,
     u64 val) {
  wal = write_ahead_log();
  wal->add(&arr[arr_sz]);
  wal->add(&arr_sz);

  wal->msync();

  arr[arr->size++] = val;

  msync(mem, pool_size,
        MS_SYNC);
}
```

(b) msync

```
mem = mmap(..)

void append(u64 *arr,
     u64 val) {
  arr[arr_sz++] = val;

  msync(mem, pool_size,
        MS_SYNC);
}
```

(c) Failure atomic msync

Logging
Update
Persist operation

Fig. 2: Comparison of PM programming techniques using an `append()` function that appends a new entry at the end of a persistent array.

### B. Filesystem-based Durability and `msync`

The POSIX `msync()` system call guarantees persistency but provides no atomicity. Part of the dirty data can reach the storage before the application calls `msync()`.

To achieve atomicity, applications often use a write-ahead-log (WAL) [7], [13] to write the modified data to a log and then change the memory location in place. Applications (e.g., Kyoto Cabinet [7]) issue two `msync()` every time they need to atomically update a mapped file, one to persist the write-ahead-log and the second to persist the application updates. Once the data is updated and durable with `msync()`, the application can drop the log.

Although applications using `msync()` based crash-consistency model directly run off of DRAM (when memory-mapped), they suffer from the overhead of context switches, page table scanning (for finding dirty pages), and TLB shootdowns (to clear access/dirty bit for the page table). This overhead is negligible compared to the access latency of disks and SSDs, but when running on NVM or memory semantic SSDs, the overhead of performing an `msync()` dominates the application's runtime. Figure 1 shows the % of runtime spent on the `msync()` call, the context switch overhead for the `msync()` call, and the TLB shootdown overhead across the YCSB workloads for PMDK's KV-Store, modified to use `msync()`. For 2 MiB pages, `msync()`'s overhead is up to 100% of the execution runtime.

With DAX-mapped files, although application LOADs and STOREs are directly to the storage media (e.g., Optane) and filtered through caches, the `msync()` system call still provides no atomic durability guarantee. On `msync()` on a DAX-mapped file, the kernel simply flushes the cachelines of all dirty pages to the persistent device.

Programmers can use a userspace transactional interface (e.g., PMDK) to avoid performance bottlenecks of the `msync()` system call. While this helps with the software overhead, PMDK requires programmers to carefully annotate variables, wrap operations in transactions, and use non-standard pointers. These additional requirements make crash-consistent programming with PMDK hard [14] and error-prone [4].

### C. Programming with FAMS

FAMS fixes shortcomings of the POSIX `msync()` and simplifies crash consistency with a failure-atomic `msync()` interface. FAMS lets the programmer call `msync()` to guarantee that the updates since the last call to `msync()` are made atomically durable. Despite FAMS's simpler programming

model, its kernel-based implementation suffers from performance overheads.

Figure 2 compares PMDK, traditional `msync()`-based WAL [7], [13], and FAMS using an `append()` method for an array. Unlike PMDK or `msync()`, failure atomic `msync()` does not require the programmer to manually log updates either using an undo-log or a write-ahead log.

In the FAMS variant (Figure 2c), the application maps a file into its address space (Line 1). Next, the application updates the mapped data using LOADs and STOREs (Line 5) and, finally, calls `msync()` when the data is in a consistent state (Lines 7-8). FAMS ensures that the backing file always reflects the most recent successful `msync()` call, which contains a consistent state of application data from which the application may recover.

FAMS implements failure-atomicity for a file by disabling writebacks from the page cache. When an application calls `msync()` on a memory-mapped file, FAMS uses the JBD2 layer of `Ext4` to journal both metadata and data for the file. In contrast, PMDK and WAL-based crash consistency require programmers to annotate updated memory locations manually. Lines 6-7, Figure 2a for PMDK, and Lines 5-9, Figure 2b for WAL-based crash consistency.

Despite FAMS's simpler programming interface, applications still suffer from kernel-based durability's performance overhead (e.g., context switch overhead, page table scanning, etc.).

## III. Overview

Snapshot overcomes FAMS's limitations by providing a drop-in, userspace implementation of failure atomic `msync()`, resulting in a significant performance improvement for crash-consistent applications. To provide low overhead durability, Snapshot introduces a compiler-based mechanism to track dirty data in userspace. Snapshot records these updates in an undo log that is transparent to the programmer. On `msync()`, Snapshot updates the persistent storage locations recorded in the log. In case of a failure, Snapshot can use the log to roll back any partially durable data.

Since Snapshot is implemented in userspace, it avoids the overhead of switching to the kernel and managing TLB coherency. Using Snapshot, the application synchronously modifies only data on the DRAM, speeding up the execution. At the same time, Snapshot maintains a persistent copy on the backing media and automatically propagates all changes to the persistent copy on an `msync()`.

As Snapshot is built on the `msync()` interface, programmers can port any conventional application written for `msync()`-based crash consistency with minimal effort to benefit from automatic dirty-data tracking while significantly improving runtime performance. Snapshot's userspace implementation enables legacy disk-based applications to take advantage of faster access times and direct-access (DAX) storage without requiring extensive application rewrites.

## IV. Implementation

Snapshot is implemented as a combination of its compiler pass and a runtime library, `libsnapshot`. Snapshot's compiler instruments every store instruction that can write to the heap using a call to an undo-log function. The runtime library, `libsnapshot`, provides runtime support for Snapshot: implementing the logging function and Snapshot's `msync()`.

Next, we will discuss how the programming interface and logging for Snapshot are implemented, followed by the various optimizations possible in Snapshot to improve its performance.

### A. Logging, Instrumentation, and `msync()`

Snapshot tracks updates to persistent data by instrumenting each STORE in the target application with a call to the logging function (instrumentation). During runtime, the instrumentation takes the STORE's address as its argument, checks if the STORE is to a persistent memory file, and logs the destination memory location to an undo log.

**Logging and Recovery.** Snapshot's undo log lives on persistent media to enable recovery from crashes during an `msync()` call. When an application calls `msync()`, Snapshot reads the addresses from the undo log entries and copies all modified locations from DRAM to the backing media. The call to `msync()` only returns when all the modified locations are durable. If the system crashes while copying the persistent data, on restart, Snapshot uses its undo-log to undo any changes that might have partially persisted.

While Snapshot maintains per-thread logs, calls to `msync()` persist data from all threads that have modified data in the memory range. Snapshot maintains a thread-local log to keep track of modified locations and their original values. Snapshot provides limited crash-consistency guarantees for multithreading, similar to PM transactional libraries like PMDK. E.g., PMDK prohibits programmers from modifying shared data from two threads in a single transaction. Similarly, in Snapshot, the program should not modify a shared location from two threads between two consecutive `msync()`s.

**Log Format.** Logs in Snapshot are per-thread and store only the minimal amount of information needed to undo an interrupted `msync()`. Logs hold their current state, that is, whether it holds a valid value. The log also maintains a tail that points to the next free log entry and the size of the log for use during recovery. Each log entry in the log is of variable length. The log entry consists of the address, its size in bytes, and the original value at the address.

While Snapshot tracks all store operations to the memory-mapped region, POSIX calls such as `memcpy()`, and `memmove()` are not instrumented as they are part of the OS-provided libraries. To solve this, `libsnapshot` wraps the calls to `memcpy()`, `memmove()`, and `memset()` to log them directly and then calls the corresponding OS-provided function. While Snapshot catches some of these functions, applications relying on other functions, e.g., `strtok()`, would need to recompile standard libraries (glibc, muslc, etc.) with Snapshot to be crash-consistent.

**Logging Design Choices.** Despite implementing undo-logging, Snapshot only needs two fences per `msync()` to be crash-consistent, as it does not need to wait for the undo-logs to persist before modifying the DRAM copy. This contrasts with

PMDK (which also implements undo-logging), where every log operation needs a corresponding fence to ensure the location is logged before modifying it in place. Redo logging persistent memory libraries eliminate this limitation and only need two fences per transaction. Redo logging, however, requires the programmer to interpose both the loads and stores to redirect them to the log during a transaction, resulting in higher runtime overhead. Snapshot, on the other hand, only interposes store instructions which always write only to the DRAM and avoids any redirection.

### B. Optimizing Snapshot

Snapshot includes a range of optimizations to maximize its performance. In particular, it must address challenges related to the cost of range tracking and reducing the required instrumentation.

*1) Low-cost Range Tracking:* Since Snapshot's compiler has limited information about the destination of a STORE, on every call, the instrumentation checks if the logging request is to a memory-mapped persistent file. Snapshot simplifies this check by reserving virtual address ranges for DRAM (*DRAM range*) and the backing memory (*persistent range*) when the application starts. Reserving ranges on application start makes checks for write operations a simple range check. In our implementation, we reserve 1 TiB of virtual address space for both ranges to map all persistent-device-backed files. This range is configurable and is limited only by the kernel's memory layout. Further, copying a location from DRAM to the backing media now only needs a simple arithmetic, i.e., copy from offset in the DRAM range to the same offset in the persistent range.

*2) Fewer Instrumentations:* Instrumenting stores indiscriminately results in useless calls for `stores` that cannot write to persistent locations (e.g., stack addresses). Snapshot reduces this overhead by tracking all stack allocations in a function at the LLVM IR level during compilation. Next, Snapshot instruments only those `stores` that may not alias with any stack-allocated addresses, resulting in a limited number of useless instrumentation.

### C. Optimizing Backing Memory Accesses

Flush and fence instructions needed to ensure crash consistency add significant runtime overhead. To understand and reduce this overhead, we study the relative latency of write+`clwb` vs. NT-Store instructions and find that non-temporal stores, particularly those which align with the bus's transfer size, result in considerable performance improvement over the `clwb` instruction.

While we perform the experiments on Intel Optane DC-PMM, we expect the results and methodology to be similar on other storage devices as they have similar memory hierarchies (volatile caches backed by byte-addressable storage devices).

Figure 3 shows the latency improvement from using NT-Stores to update PM data vs. using writes followed by `clwbs`. The heatmap measures the latency of the operation while

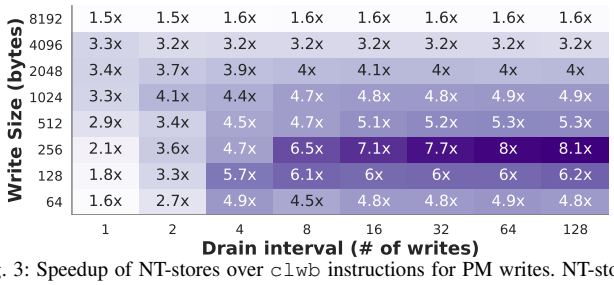



Fig. 3: Speedup of NT-stores over `clwb` instructions for PM writes. NT-stores always outperform write+`clwb`.

varying both the write size, that is, the amount of data written, and the frequency of the fence operation (`sfence`).

From Figure 3 we observe that NT-Stores consistently outperform `store+clwb`. Moreover, the performance gap between `clwbs` and NT-Stores increases as the fence interval increases. In contrast, when the write size is increased, the performance only increases until the write size matches the DDR-T transaction size (256 B). Since the CXL protocol uses 64B packet size for v1-2 and 256B for CXL v3, we expect to see maximas for those sizes for CXL-based persistent devices.

Based on this observation, Snapshot always uses NT-Store instructions to copy modified cachelines from the DRAM copy to the persistent copy on a call to `msync()`.

**Reducing Backing Memory Accesses** To find modified locations to write to the persistent copy from the DRAM copy, Snapshot iterates over its undo log. As this log is stored on the backing memory, it can be slow to access. This is a result of the log design where each log entry is of variable length, thus, accesses to sequential log entries result in variable strides and poor cache performance. As a result, Snapshot has to spend time traversing the entries. To reduce read traffic to the backing memory and mitigate this additional overhead, Snapshot keeps an additional, in-DRAM list of the updated addresses and their sizes, thus avoiding accessing the backing memory.

While it is possible to split the log to separate log entry's data into a different list, log entries would then require more instructions to flush them, adding overhead to the critical path.

### D. Memory Allocator

While Snapshot provides a failure atomic `msync()`, applications need to allocate and manage memory in a memory-mapped file. Shared memory allocators, like `boost.interprocess` provide an easy way to manage memory in a memory mapped file by providing `malloc()` and `free()`-like API. These operations, while enable memory management, are not crash consistent.

However, Snapshot's ability to automatically log all updates to the persistent memory, enables applications to use volatile shared memory allocators for allocating memory in a crash-consistent manner.

To demonstrate Snapshot's utility, we use Snapshot to enable `boost.interprocess` [15] to function as a persistent memory allocator. `boost.interprocess` allocates objects from a memory-mapped file, provides API to access the root object as a pointer, and allocate/free objects while Snapshot

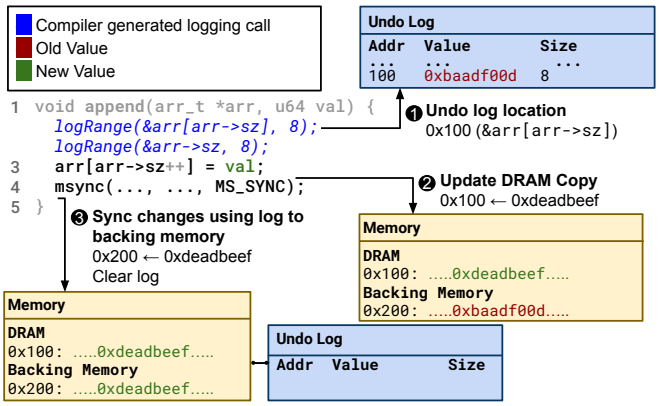



Fig. 4: Snapshot's working. Instrumented binary calls `libsnapshot.so`'s logging function for every store. Changes are atomically durable on `msync()`.

tracks updates and makes changes atomically durable on an `msync()`.

**Decoupling Memory Allocator and Logging** Unlike traditional PM programming libraries that couple memory allocators and logging techniques, Snapshot permits any combination of a logging technique and a memory allocator. For example, PMDK's allocator only supports redo logging. Restricting the programmer to the specific characteristics of their implementation. On the other hand, Snapshot provides programmer the ability to independently choose the memory allocator and logging technique, suiting the specific needs of the workload.

### *E. Putting it all Together*

To see how snapshot works in practice, consider an array `append()` function that takes a value and inserts it into the next available slot in the array. Figure 4 shows the implementation of this function along with the memory and log states as the program executes. In the example, the instrumented program automatically undo-logs the updated location (i.e., call to `logRange`). For illustration purposes, we only show updates to the array element, not the array size.

When the program starts executing ❶, the instrumentation calls the logging function with the address of updated locations (`&arr[arr->sz]` and `&arr->sz`), and update sizes. The function logs the address by creating a new entry in the thread-local undo log.

Next, ❷ the program continues and updates memory locations. Since the program only directly interacts with the DRAM, the value in the DRAM is updated, but the value in the backing memory (e.g., PM) is unchanged. Finally, ❸ the application calls `msync()` to update the backing memory. On this call, Snapshot iterates over the log to find all locations that have been updated and uses them to copy updates from DRAM to PM. After updating PM with the values from DRAM, Snapshot drops the log by marking it as invalid. Once `msync()` returns, any failure would reflect the persistent state of the most recent `msync()`.

### *F. Correctness Check*

We test our compiler pass and resulting binary to ensure correctness and crash consistency. We test for crash consistency bugs by injecting a crash into the program before it commits a transaction when Snapshot has copied all the changes to the backing store but has not invalidated the log. On a restart, Snapshot should recover and let the application continue its normal execution. This is only possible if the compiler pass correctly annotates all the store instructions and the logging function logs them. We ran these tests for multiple configurations and inputs and found that the compiler pass and the runtime recovered the application each time.

## V. Results

To understand Snapshot's performance, instrumentation overhead, and the impact of various optimizations, we evaluate several microbenchmarks, persistent memory applications, including Kyoto Cabinet, and crash-consistency solutions. Further, to get an estimate of Snapshot's performance on future hardware, we evaluate Snapshot against PMDK on a CXL-based emulated memory-semantic SSD.

### *A. Configuration*

Table II lists the six different configurations we use to compare the performance of Snapshot. With PMDK, the workloads are implemented using its software transactional memory implementation. The Snapshot-NV and Snapshot implementations are similar, with the difference in how they track dirty data in DRAM. The Snapshot-NV implementation uses the undo log to flush the dirty data on a call to `msync()`. In comparison, the Snapshot implementation uses a separate, volatile list to flush the dirty data (Section IV-C). All implementations of Snapshot otherwise have this optimization enabled. Table III lists the configuration used for all experiments in the results section.

### *B. Failure Atomic `msync()` Implementations*

Three implementations of failure atomic `msync()` are possible candidates for comparison with Snapshot. The original implementation, FAMS, is by Park et al. [8]. The other implementations, `famus_snap` [16] and AdvFS's implementation [17] use reflinks and file cloning, respectively to create shallow copies of the backing file on `msync()`.

FAMS by Park et al. [8] is not open-sourced. We use POSIX `msync()` with data journalling enabled to approximate its performance. FAMS works by reconfiguring the `Ext4`'s data journal to not write back to the backing media until the application calls `msync()`. Since FAMS uses data journalling to implement failure atomicity, their implementation performs similarly to `msync()` on `Ext4` mounted with the option `data=journal`, as shown by Park et al. [8].

Implementation of failure atomic `msync()` by Verma et al. on AdvFS [17] is not open-sourced, however, Kelly's `famus_snap` [16] is open-sourced and can be evaluated. `famus_snap` uses reflinks to create a snapshot of the memory-mapped file on a call to `msync()`.

`famus_snap`, however, is much slower than the POSIX `msync()` due to slow underlying `ioctl(FICLONE)` calls. In our evaluation, we found that `famus_snap` is between 4.57× to 338.57× slower than `msync()` for the first and 500th calls, respectively.

TABLE II: Evaluated configurations.

| Config | Description | Dirty data tracking | Crash consistent | Working memory |
|---|---|---|---|---|
| PMDK | Intel's PMDK-based implementation. | Programmer (byte) | ✔ | PM |
| Snapshot-NV | Snapshot, tracking using undo-log. | Auto., (byte) | ✔ | DRAM |
| Snapshot | Snapshot, tracking using a volatile list (Section IV-C). | Auto., (byte) | ✔ | DRAM |
| `msync()` 4 KiB | Page cache mapped, 4 KiB pages. | Auto., OS (4KiB) | ✘ | DRAM |
| `msync()` 2 MiB | Page cache mapped, 2 MiB pages. | Auto., OS (2MiB) | ✘ | DRAM |
| `msync()` data journal | Page cache, ext4 (`data=journal`), 4 KiB Pages | Auto., OS (4 KiB) | ✘ | DRAM |

TABLE III: System configuration

| | |
|---|---|
| CPU | 2 × Intel 6230, 40 HW threads, |
| DRAM | 192 GiB (DDR4) |
| Optane | 100 series, 128×12 = 1.5 TiB, AppDirect Mode |
| OS & Kernel | Ubuntu 20.04.3 & Linux 6.0.0 |
| Build system | LLVM/Clang 13.0.1 |
| Block Device (for emulation) | Intel Optane SSD DC P4800X |

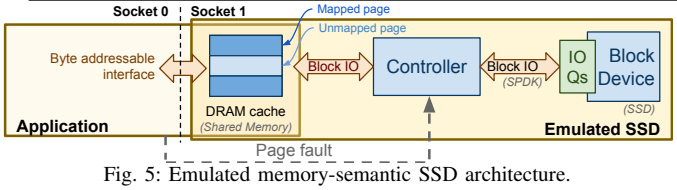



Fig. 5: Emulated memory-semantic SSD architecture.

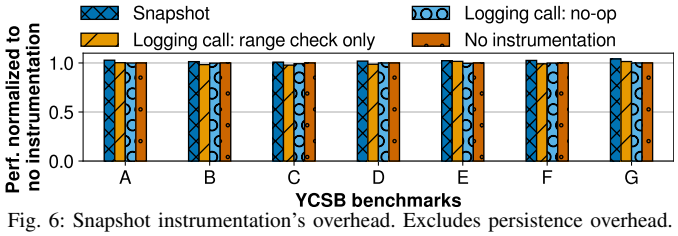



Fig. 6: Snapshot instrumentation's overhead. Excludes persistence overhead.

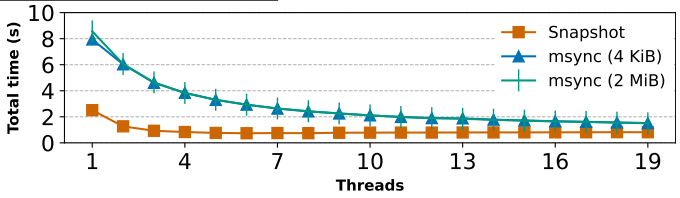



Fig. 7: Scaling of Snapshot and `msync()` with increasing thread count.

Since `famus_snap` is orders of magnitude slower than `msync()`, we do not evaluate it further.

### C. Evaluating Snapshot on CXL-based Memory Semantic SSDs

CXL-attached memory semantic SSDs [1] are CXL-based devices with a large DRAM cache backed by a block device. These devices appear as memory devices to the host processor and support byte-addressable accesses.

To understand how Snapshot would perform on CXL-attached memory semantic SSDs, we created a NUMA-based evaluation platform. In our emulation, we implement the DRAM cache using shared memory and service cache miss from a real SSD in userspace using SPDK [18]. The target application and the emulated SSD are pinned to different sockets to emulate a CXL link (similar to Maruf et al. [19]). Figure 5 shows the architecture of our emulated memory semantic SSD.

In our implementation, the shared memory used to emulate the DRAM cache has only a limited number of pages (128 MiB) mapped. On access to an unmapped page by the application, the emulated SSD finds a cold page to evict using Intel PEBS and reads and maps the data from the SSD.

Our implementation has a 14.3µs random access latency at a 91.8% DRAM cache miss rate and a 2.4µs latency at a 16.3% DRAM cache miss rate. While these latencies might be high compared to Intel Optane DC-PMM, they represent a slower, byte-addressable media and are close to low-latency flash, e.g., Samsung Z-NAND [20].

### D. Microbenchmarks

Next, we use microbenchmarks to study how Snapshot's store instrumentation affects performance and if Snapshot's implementation limits its scalability in multithreading.

**Instrumentation Overhead.** To understand the instrumentation overhead of Snapshot, we run it with and without instrumentation and logging enabled. Figure 6 shows the runtime performance of different variants of Snapshot running the YCSB workload. The "Logging call: no-op" variant returns from the logging call without performing any checks or logging. The "Logging call: range check only" measures the execution where the logging call only performs the range check but does not log any data. Finally, the "No instrumentation" variant is compiled without Snapshot's compiler pass and thus has no function call overhead. Among these, only Snapshot logs the modifications and is crash-consistent. In all other variants, a call to `msync()` is a no-op.

The results show that even with the compiler's limited information about a store instruction, the overhead from the instrumentation is negligible since stores are relatively few compared to other instructions.

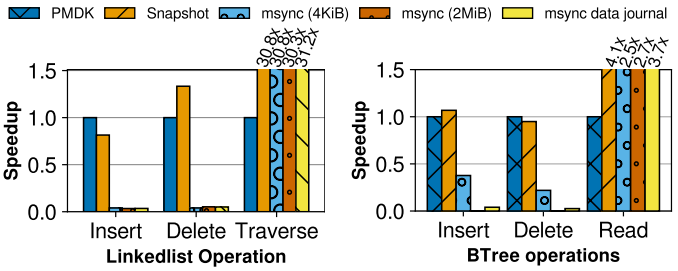
**Store Instrumentation Statistics.** Across the workloads, Snapshot instrumented 10.8% store instructions on average. Out of all the store instructions, Snapshot skipped 84.6% because they were stack locations, and 4.54% because they were aliased to stack locations. In case a location is not a stack location, or aliased to one, Snapshot errs on the side of caution and instruments it. During runtime, the instrumentation checks the store's destination to ensure it is to a persistent location.

**Multithreaded Scaling.** To understand the impact of multithreading on performance, we scale the number of threads and measure the total runtime for a microbenchmark. Figure 7 shows that Snapshot scales similar to `msync()` with an increasing number of threads while maintaining a lower overhead overall. Each thread in this microbenchmark operates on an independent memory region and mimics a small transactional update by writing to random memory locations and calling `msync()`. The threads call 500k `msync()`s in total, with each `msync()` flushing modification from two random writes to their memory region.

### E. Persistent Memory Applications and Data-Structures

We evaluate several applications to show that Snapshot consistently outperforms PMDK across various workloads and POSIX `msync()` on write-heavy workloads when running on Intel Optane DC-PMM. Workloads include a linked list and a b-tree implementation from Intel's PMDK, PMDK's KV-Store using the YCSB workload, and Kyoto Cabinet.

**Linked List** Figure 8a shows the performance of a linked list




(a) Linked list implementation on Intel Optane DC-PMM. Higher is better.

(b) B-tree map implementation on Intel Optane DC-PMM. Higher is better.

Fig. 8: Performance comparison of PMDK, Snapshot, and `msync()`.

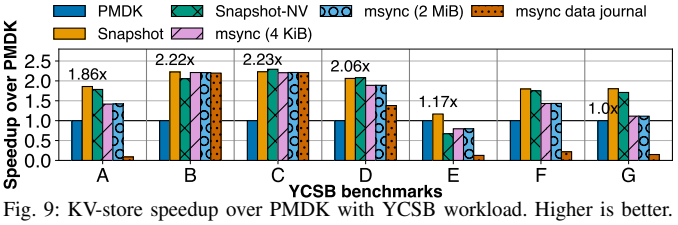



Fig. 9: KV-store speedup over PMDK with YCSB workload. Higher is better.

data structure implemented using PMDK, traditional `msync()` with 4 KiB and 2 MiB page size, `msync()` with data journal, and Snapshot. *Insert* inserts a new node to the tail of the list, *Delete* removes a node from the head, and *Traverse* visits every node and sums up the values. PMDK and Snapshot are crash-consistent, while the `msync()` implementations are not. Each operation is repeated 1 million times.

Since Snapshot runs the application entirely on DRAM and performs userspace synchronization with the backing media on a call to `msync()`, it significantly outperforms PMDK on Traverse workload while being competitive in Insert and Delete. On every call to `msync()`, the traditional filesystem implementation needs to perform an expensive context switch and TLB shootdown, slowing it considerably compared to PMDK and Snapshot.

**B-Tree** We compare Snapshot against PMDK using a b-tree data structure of order 8 with 8-byte keys and values on Intel Optane DC-PMM. We use three workloads: (1) *Insert workload* generates 1 million random 8-byte keys and values. (2) *Read workload* traverses the tree in depth-first order. Finally, (3) the *delete workload* deletes all the keys in the insertion order.

Figure 8b shows that Snapshot performs similarly to PMDK for the insert and delete workloads while outperforming all `msync()` implementations by at least 2.8×. For the read workload, Snapshot and `msync()` achieve significant speedup (4.1×) over PMDK.

**KV-Store** Next, we compare the performance of Snapshot on Optane DC-PMM using a key-value store implemented using a hash table where each bucket is a vector. For evaluation, we use the YCSB workloads A-F and an additional write-only workload, G. Each workload performs 5 million operations on a database with 5 million key-value pairs each.

Figure 9 shows the performance of the KV-store against PMDK using different Snapshot configurations described in Table II. For Snapshot, we present the results using volatile and non-volatile lists for finding modified cachelines on `msync()`. Overall, Snapshot shows between 1.2× and 2.2× performance improvement over PMDK.

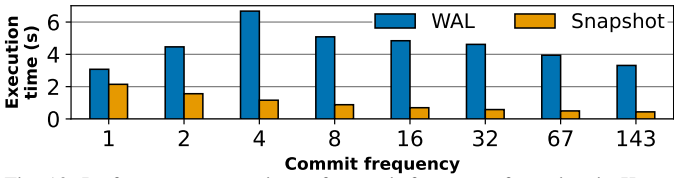



Fig. 10: Performance comparison of commit frequency for writes in Kyoto Cabinet on Intel Optane DC-PMM. Lower is better.

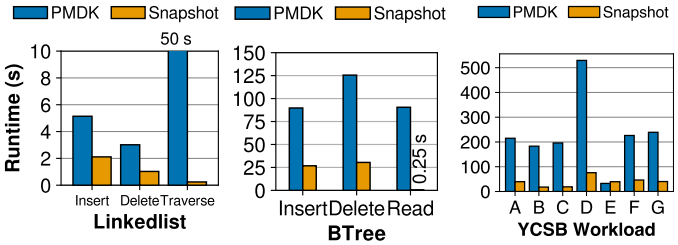



Fig. 11: Linked list, b-tree and KV-store on Emulated Memory Semantic SSD (Section V-C). Lower is better.

Against `msync()`, Snapshot shows a significant performance improvement, especially when compared to `msync()` with data journal enabled. Snapshot does this while providing an automatic crash-consistency guarantee. Moreover, several Snapshot optimizations, including using a separate volatile list for tracking dirty data in the memory (Section IV-C) provide considerable improvement to Snapshot's performance.

**Kyoto Cabinet** Figure 10 shows the performance comparison between Kyoto Cabinet's built-in WAL+`msync()` based crash-consistency mechanism to Snapshot with a varying number of updates per transaction. Overall, Snapshot outperforms Kyoto's transaction implementation between 1.4× and 8.0×.

For crash consistency, Kyoto Cabinet combines WAL and `msync()`. For the Snapshot version, we disable Kyoto Cabinet's WAL implementation. This version, when compiled with Snapshot's compiler, is automatically crash-consistent.

### *F. Memory Semantic SSDs*

We evaluate Linked list, B-tree, and KV-Store on our emulated memory semantic SSD and observe that Snapshot significantly outperforms PMDK for linked list and b-tree (Figure 11). For linked list, Snapshot outperforms PMDK by 1.7×, 3.2×, and 171.0× for insert, delete, and read, respectively. For b-tree, Snapshot outperforms PMDK by 3.4×, 4.1×, and 364.5× for insert, delete, and read workloads, respectively.

For the KV-Store benchmark, Snapshot outperforms PMDK by up to 10.9× for all but the 'E' workload, where PMDK is 1.23× faster. As our emulation is software-based, it does not support the `msync()` system call, so we did not evaluate Kyoto Cabinet.

### *G. Programming effort*

Implementing Snapshot did not require any changes to `boost.interprocess` as any changes to the memory allocator's state are automatically persisted with the `msync()` call by the workloads. For Kyoto Cabinet, we changed 11 lines of code, including disabling its built-in crash-consistency

mechanism. Thus, demonstrating the utility of Snapshot's simple programming interface for achieving crash consistency.

## VI. Related Work

Many prior works have proposed techniques to simplify the persistent memory programming model. Romulus [21] uses a twin-copy design, storing both copies on PM for fast persistence. Romulus uses a redo log to synchronize the active copy with the backing copy on a transaction commit. Pisces [22] is a similar PM transaction implementation that uses dual version concurrency control (DVCC), where one of the versions is the application data on persistent memory, and the other is the redo log. Pisces improves performance by using a three-stage commit protocol where the stages where the data is durable, visible, and propagated are decoupled.

Libnvmmio [23] provides an `msync()`-based interface, but it does so by intercepting filesystem IO calls, e.g., `read()` and `write()`. Thus, unlike Snapshot, Libnvmmio does not support the memory mapped file interface. Similarly, DudeTM [24] while uses memory mapped interface and stages working copy in the DRAM, requires the programmer to use PMDK-like transactional interface, increasing programming effort.

Automated solution like compiler passe to simplify the programming effort includes Atlas [25] which adds crash-consistency to existing lock-based programs by using the outermost lock/unlock operations. Synchronization Free Regions (SFR) [26] extends this idea and provides failure-atomicity between every synchronization primitive and system call.

Similarly, some works use language support to automatically add persistence to applications written for volatile memory. Breeze [27] uses compiler instrumentation for logging updates to PM but requires the programming to explicitly wrap code regions in transactions and annotate PM objects. NVTraverse [28] and Mirror [29] convert lock-free data structures persistent using programmer annotation and providing special compare and swap operations.

Memory allocators are important in achieving crash-consistency. To resume after a crash, persistent memory allocators need to save their metadata state along with the allocated data. Several works in the past have proposed PM allocators. Romulus [21] supports porting any sequential memory allocator designed for volatile memory allocation to PM by wrapping all the persistent types in a special class that interposes store accesses. This is similar to Snapshot's compiler-based instrumentation, but in contrast to Romulus, Snapshot requires no programmer effort to use a volatile memory allocator for PM.

LRMalloc [30] is a PM allocator that persists only information that is needed to reconstruct the allocator state after a crash. Metall Allocator [31] provides a coarse crash-consistency mechanism by using the underlying DAX filesystem's copy-on-write mechanism. However, Metall only guarantees persistency when the Metall allocator's destructor is called, making it impractical for applications that need higher frequency persistency (e.g., databases). Kelly et al. [32] present a persistent memory allocator (PMA) to provide a persistent heap for conventional volatile programs and create a persistent variant of gawk, `pm-gawk` [33].

## VII. Conclusion

Snapshot provides a userspace implementation of failure atomic `msync()` (FAMS) that overcomes its performance limitation. This advantage is especially apparent against `msync()` with huge pages enabled. 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. Further, Snapshot guarantees that the persistent memory state of the application is only updated on a call to `msync()`.

Further, we study the latency difference between the different ways to write to persistent memory, NT-Stores vs. cacheline writebacks for uncached data.

Finally, Snapshot alleviates limitations of FAMS, enabling applications to take advantage of faster, byte-addressable storage devices. Moreover, Snapshot, unlike FAMS, completely avoids any system calls for crash consistency or manual annotation and transactional semantics required by PMDK.

## Acknowledgement

This work was supported in part by the ACE Center for Evolvable Computing, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

## References

[1] S. 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.

[2] Viking Technologies, "DDR4 NVDIMM." [Online]. Available: https://www.vikingtechnology.com/non-volatile-memory/ddr4-nvdimm/

[3] C. Inc., "Rethink embedded memory with ReRAM," https://www.crossbar-inc.com/products/high-performance-memory/.

[4] S. Liu, K. Seemakhupt, Y. Wei, T. Wenisch, A. Kolli, and S. 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*, 2020, pp. 1187–1202.

[5] A. Jhingran and P. Khedkar, "Analysis of recovery in a database system using a write-ahead log protocol," *ACM Sigmod Record*, vol. 21, no. 2, pp. 175–184, 1992.

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

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

[8] S. Park, T. Kelly, and K. 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*, ser. EuroSys '13. Association for Computing Machinery, 2013, p. 225–238.

[9] N. Amit, "Optimizing the TLB shootdown algorithm with page access tracking," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 27–39.

[10] Z. Wang, M. Taram, D. Moghimi, S. Swanson, D. Tullsen, and J. Zhao, "NVLeak: Off-chip side-channel attacks via non-volatile memory systems," in *USENIX Security Symposium*, vol. 2023.

[11] *Compute Express Link (CXL) Specification*, Compute Express Link Consortium, Inc., October 2020, revision 2.0.

[12] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson, "Basic performance measurements of the intel optane DC persistent memory module," *CoRR*, vol. abs/1903.05714, 2019.

[13] "Sqlite, write-ahead logging," https://www.sqlite.org/wal.html.

[14] M. Hoseinzadeh and S. Swanson, "Corundum: Statically-enforced persistent memory safety," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 2021, p. 429–442.
[15] I. Gaztanaga, *Boost 1.79.0 Documentation*, 2022, ch. Boost.Interprocess.
[16] T. Kelly, "Good old-fashioned persistent memory." *;login: USENIX Mag.*, vol. 44, no. 4, 2019.
[17] R. Verma, A. A. Mendez, S. Park, S. Mannarswamy, T. Kelly, and C. B. Morrey, "Failure-atomic updates of application data in a Linux file system," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, ser. FAST'15. USA: USENIX Association, 2015.
[18] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul, "SPDK: A development kit to build high performance storage applications," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2017, pp. 154–161.
[19] H. A. Maruf, H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhattacharya, C. Petersen, M. Chowdhury, S. Kanaujia, and P. Chauhan, "TPP: Transparent page placement for CXL-enabled tiered memory," *arXiv preprint arXiv:2206.02878*, 2022.
[20] W. Cheong, C. Yoon, S. Woo, K. Han, D. Kim, C. Lee, Y. Choi, S. Kim, D. Kang, G. Yu *et al.*, "A flash memory controller for 15$\mu$s ultra-low-latency ssd using high-speed 3d nand flash with 3$\mu$s read time," in *2018 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 2018, pp. 338–340.
[21] A. Correia, P. Felber, and P. Ramalhete, "Romulus: Efficient algorithms for persistent transactional memory," in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, 2018.
[22] J. Gu, Q. Yu, X. Wang, Z. Wang, B. Zang, H. Guan, and H. Chen, "Pisces: A scalable and efficient persistent transactional memory," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 913–928.
[23] J. Choi, J. Hong, Y. Kwon, and H. Han, "Libnvmmio: Reconstructing software io path with failure-atomic memory-mapped interface," in *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, 2020, pp. 1–16.
[24] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren, "Dudetm: Building durable transactions with decoupling for persistent memory," *ACM SIGPLAN Notices*, vol. 52, no. 4, 2017.
[25] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '14. ACM, 2014, pp. 433–452.
[26] V. Gogte, S. Diestelhorst, W. Wang, S. Narayanasamy, P. M. Chen, and T. F. Wenisch, "Persistency for synchronization-free regions," *SIGPLAN Not.*, vol. 53, no. 4, p. 46–61, jun 2018. [Online]. Available: https://doi.org/10.1145/3296979.3192367
[27] A. Memaripour and S. Swanson, "Breeze: User-level access to non-volatile main memories for legacy software," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, 2018.
[28] M. Friedman, N. Ben-David, Y. Wei, G. E. Blelloch, and E. Petrank, "NVTraverse: In NVRAM data structures, the destination is more important than the journey," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020, 2020, p. 377–392.
[29] M. Friedman, E. Petrank, and P. Ramalhete, *Mirror: Making Lock-Free Data Structures Persistent*, 2021, p. 1218–1232.
[30] W. Cai, H. Wen, H. A. Beadle, C. Kjellqvist, M. Hedayati, and M. L. Scott, "Understanding and optimizing persistent memory allocation," in *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management*, 2020, pp. 60–73.
[31] K. Iwabuchi, L. Lebanoff, M. Gokhale, and R. Pearce, "Metall: A persistent memory allocator enabling graph processing," in *2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, 2019, pp. 39–44.
[32] T. Kelly, Z. F. Tan, J. Li, and H. Volos, "Persistent memory allocation," *ACM Queue magazine 2022*, 2022.
[33] T. Kelly, *Persistent-Memory `gawk` User Manual*, August 2022, `pm-gawk` version 2022.08Aug.03.1659520468.