PMThreads: Persistent Memory Threads Harnessing Versioned Shadow Copies

DOI:
10.1145/3385412.3386000

Document Version
Accepted author manuscript

Published in:
Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20)

PMThreads: Persistent Memory Threads Harnessing Versioned Shadow Copies

Zhenwei Wu
National University of Defense Technology, China
University of Manchester, UK
zhenweiwu@nudt.edu.cn

Kai Lu
National University of Defense Technology, China
kailu@nudt.edu.cn

Andrew Nisbet
University of Manchester, UK
andy.nisbet@manchester.ac.uk

Wenzhe Zhang
National University of Defense Technology, China
zhangwenzhe@nudt.edu.cn

Mikel Luján
University of Manchester, UK
mikel.lujan@manchester.ac.uk

Abstract

Byte-addressable non-volatile memory (NVM) makes it possible to perform fast in-memory accesses to persistent data using standard load/store processor instructions. Some approaches for NVM are based on durable memory transactions and provide a *persistent programming paradigm*. However, they cannot be applied to existing multi-threaded applications without extensive source code modifications. Durable transactions typically rely on logging to enforce failure-atomic commits that include additional writes to NVM and considerable ordering overheads.

This paper presents PMThreads, a novel user-space runtime that provides transparent failure-atomicity for lock-based parallel programs. A shadow DRAM page is used to buffer application writes for efficient propagation to a dual-copy NVM persistent storage framework during a *global quiescent state*. In this state, the *working* NVM copy and the crash-*consistent* copy of each page are atomically updated, and their roles are switched. A *global quiescent state* is entered at timed intervals by intercepting *pthread* lock acquire and release operations to ensure that no thread holds a lock to persistent data.

Running on a dual-socket system with 20 cores, we show that PMThreads substantially outperforms the state-of-the-art Atlas, Mnemosyne and NVthreads systems for lock-based benchmarks (Phoenix, PARSEC benchmarks, and microbenchmark stress tests). Using Memcached, we also investigate the scalability of PMThreads and the effect of different time intervals for the quiescent state.

PLDI '20, June 15–20, 2020, London, UK

CCS Concepts: • **Hardware** → **Non-volatile memory**; • **Computer systems organization** → *Processors and memory architectures*; *Reliability*.

Keywords: non-volatile memory, memory persistence, parallel programs

ACM Reference Format:
Zhenwei Wu, Kai Lu, Andrew Nisbet, Wenzhe Zhang, and Mikel Luján. 2020. PMThreads: Persistent Memory Threads Harnessing Versioned Shadow Copies. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3385412.3386000

1 Introduction

Non-volatile memory (NVM) access latency improvements potentially enable data persistence to be implemented with much lower overhead by replacing costly serial accesses to block devices with accesses to NVM resources. To leverage NVM technology, it is necessary to address the fundamental challenge of providing, and ensuring correct consistent recovery of data from persistent memory in the presence of system crashes (e.g., unexpected power failures) [20, 36–38].

Durable memory transactions are one approach for enforcing memory persistence that have been investigated in [4, 7, 10, 22, 24, 30, 42], where, a group of persistent updates are atomically committed to NVM with respect to any potential system failure. Note that the implementation of durable memory transactions imposes strict constraints on the ordering of writes to persistent memory [22, 35], and this can severely limit performance. Typically, most durable memory transaction implementations rely on logging [10, 24, 42], where additional write accesses to NVM are used to record each update to persistent memory as a log entry [47]. This results in a twice-write overhead, making logging less desirable because of the limited write endurance of NVM [2]. Further, legacy applications must be rewritten to adhere to a transactional programming paradigm, and to directly exploit durable transaction APIs for NVM [5].

Beyond durable memory transactions, research projects have applied persistent semantics to lock-based programs via the inference of program-defined *Failure-Atomic SEctions* (FASEs) [5] from conventional critical sections. These systems either adopt logging mechanisms [5, 16, 25], or target (not available) non-volatile on-chip cache features [19].

We propose PMThreads, a novel runtime that leverages a dual-copy scheme to enforce crash-consistent FASE execution. More specifically, the PMThreads runtime maintains dual-versioned persistent data in NVM and only a single DRAM copy of each page is transparently accessible to the application. PMThreads safely exploits its dual-versioned operation by intercepting lock acquire and release operations to enforce *global quiescent states* to occur at regular timed intervals where no thread can hold a lock on persistent data. Write updates are buffered in the DRAM copy of a page before persistently propagating the updates in-place to the NVM working copy without any kind of logging. After a successful update, PMThreads atomically switches the roles between the *working copy* and the *consistent copy* in NVM.

Our main contributions are summarized as follows:

- PMThreads introduce a new dual-copy mechanism to perform in-place persistent writes over one copy and to obtain a consistent fallback using the other copy. Thus we reduce the two writes to NVM present in conventional log-based solutions to a single write.
- *Global quiescent states* are enforced to occur at timed intervals by intercepting `pthread` operations. Buffered writes to DRAM are safely and crash-consistently persisted to NVM within quiescent states.
- We demonstrate substantial performance improvements for PMThreads over NVthreads and Atlas using microbenchmarks for stress-tests (at least 4.7× faster), as well as the Parsec and Phoenix benchmark suites (at least 2.6× faster).
- Using Memcached, we also investigate the scalability of PMThreads and the effect of different time intervals for the quiescent state. We show that PMThreads is less than 2× average slowdown compared to a volatile pthread-based Memcached baseline.
- The recovery time of PMThreads is more than 1000× faster than Atlas, and the NVM space overheads of Atlas are at least 16× greater than that of PMThreads.

The remainder of the paper is organized as follows. Section 2 presents the background and related work. Section 3 provides an overview of the design of PMThreads, while Section 4 describes the implementation. Section 5 discusses the limitations of PMThreads. Section 6 describes the experimental evaluation, and Section 7 presents the conclusions.

2 Background and Motivation

NVM technology trends and the main challenges for persistence and crash recovery support in hybrid systems with DRAM and NVM, are presented along with related work.

2.1 Non-Volatile Memory Trends

RAM (STT-RAM) [6], Phase Change Memory (PCM) [44], and Intel and Micron's 3D XPoint memory [33], offer improved write durability/endurance as well as byte-addressing with DRAM-like performance and higher storage density than DRAM. Current technology trends suggest that future NVM will offer larger main memory resources than DRAM, but with relatively longer access latency. Consequently, memory systems based on volatile caches and hybrid combinations of DRAM and NVM are expected to become one of the preferred design choices for incorporating NVM into existing storage hierarchies [45].

Intel Optane DC Persistent Memory (OPM) [1] became generally available in 2019 in a hybrid DRAM/NVM style. OPM features two different operating modes. In Memory Mode, OPM serves as a volatile cost-effective DRAM replacement with its large memory capacity. In App Direct Mode, both volatile DRAM and non-volatile OPM are explicitly available to applications. To make effective use of Optane, applications need to flexibly decide what is to be maintained in DRAM vs. what is stored in OPM.

Persistent memory must be able to determine when an update has propagated to main memory, and this is conventionally determined by the cache line eviction policy that is implemented using a combination of operating system and hardware functionality.

Processors typically expose explicit cache flushing instructions to software, such as `clflush` and `clflushopt` on the x86 architecture. Such instructions lead to excessive and unnecessary cache line invalidations that can eliminate the performance gains of cache memory. To address this problem, Intel has proposed the `clwb` instruction to explicitly write back a cache line without invalidation. However, each individual `clwb` may still incur a significant stall overhead. We found `clwb` to have a similar overhead to `clflushopt` when executing a simple microbenchmark that uses a for-loop to increment and flush each integer in an array. The microbenchmark was executed on an Alibaba cloud server with an Intel Xeon Platinum 8269 Cascade Lake processor.

Enforcing cache flushing on its own is insufficient to ensure consistent updates to persistent memory because out-of-order processor microarchitectures typically allow stores to different addresses to be reordered and stores may not reach NVM in program specified order [36]. Memory barriers, like `sfence` on x86, are therefore necessary to control the order of persistent updates to NVM, but guarding each persistent write with a memory barrier is prohibitively expensive [28].

Furthermore, it is even more challenging to perform failure-atomic updates to NVM [11, 17, 23, 24, 29]. To ensure correct recovery from NVM in the case of a failure, persistent data requires atomic updates. Currently, only word-granularity (e.g. 64 bits) failure-atomic memory operations are provided, and a logical update to NVM may consist of a series of writes to different (word) locations.

Write ahead logging (WAL) is a commonly-used technique to enforce memory persistence [7, 42]. WAL constrains modifications to target NVM locations to be delayed (not applied) until the corresponding log entries are persistent, thus, NVM data can always be recovered to a consistent state after a system crash using the log. Undo logging and redo logging are two essential WAL paradigms. Undo logging records old values of NVM data to log entries and performs in-place update after persisting the log, this incurs persistent ordering overhead for each individual persistent store. Redo logging redirects persistent updates to a redo log instead of performing an in-place update, thus, this only requires a single persistent ordering for each transaction (or FASE). A single persistent ordering can be enforced by a single memory fence operation between all log entries, and all writes in a completed FASE. However, the cost induced by the update redirection in redo logging can itself also be expensive [41]. For example, with redo-logging, one must sequentially go through the redo log to obtain the latest value.

2.2 Related Work

Lock-delineated Failure-Atomic Section. NVthreads [16], Atlas [5], JUSTDO logging [19], iDo [25] have in common that they enforce failure-atomicity to persistent data at the boundaries of Failure Atomic SEctions (FASEs). This is inferred from lock acquisition, and lock release operations. They maintain a persistent log to ensure correct crash recovery, which introduces extra writes to NVM because they must maintain logging entries for their in-progress updates. In contrast, PMThreads benefits from performing direct in-place updates to persistent data using only a single NVM round-trip. Among these systems, PMThreads is closer to NVthreads [16], which aims at a hybrid DRAM/NVM architecture, whereas the other three runtime are designed for NVM-only memory systems.

Persistent Memory Transactions. Persistent memory transactions [4, 12, 15, 24, 30, 42, 46] are sequences of instructions guaranteed to take effects in an all-or-nothing way with respect to a system crash. DudeTM [24] and NV-HTM [4] leverage shadow DRAM copies of NVM to buffer modifications to persistent memory, and to decouple data persistence from transaction execution. For example with DudeTM, each persistent location L in NVM has a shadow location Shadow-L in DRAM. When application writes to L (a persistent write), DudeTM will direct the write to Shadow-L and return to the user. The real target L will be asynchronously updated. For workloads that present hot-spots[1], NV-HTM and DudeTM checkpoint only the most recent hot-spot update to persistent memory, and this is used to supersede repeated redo logs for the hot persistent spots. In contrast, the periodic-persistence used by PMThreads offers the potential to incur only one propagation to persistent memory for repeated updates to the same persistent location within one execution period. The quiescence enforcement of PMThreads makes it possible for updates issued by different threads over a concentrated hot-spot region to be coalesced, instead of maintaining per-thread log entries as in DudeTM and NV-HTM.

Pisces [15] leverages Multi-Version Concurrency Control (MVCC) to exploit snapshot isolation on persistent memory. Pisces maintains two data versions in NVM, and reuses redo logs to provide its newest data version. Crafty [21] has developed a novel technique called nondestructive undo logging where undo log updates are decoupled from program memory writes for persistent transactions.

Kamino-Tx [30] relies on dual NVM copies to achieve memory persistence. In particular, Kamino-Tx synchronizes its dual persistent heaps, whereas PMThreads updates its dual persistent NVM copies separately, in turns. In Kamino-Tx, any dependent transactions (transactions with read/write sets intersecting with those of a prior transaction) must wait for the main and backup copies to become consistent with each other. This requires the dependent transaction to hold a read-write lock for each object in its working set until committing, and this introduces significant programming efforts. PMThreads adopts page versioning to dynamically switch between the working and the persistent NVM copies of pages. During each interval, only the working copies are modified, and the persistent copies serve as a consistent fallback. For DudeTM, NV-HTM, Kamino-Tx and other systems in this category, the underlying assumption is that applications contain transactions rather than lock-based applications. PMThreads does not require transactional applications. Furthermore, Kamino-Tx and Crafty target pure NVM systems, whereas DudeTM, NV-HTM and PMThreads are designed for hybrid DRAM/NVM systems.

Periodic Persistence. Periodic persistence solutions, such as [8, 34], partition data structure operations into epochs. Persistent data can be recovered to the state at the end of the last completed epoch before a crash. By only issuing cache line flush operations at the end of an epoch, periodic persistence solutions achieve considerable improved performance over traditional approaches that adopt eager cache flushing for each individual persistent store. PMThreads extends this idea to general-purpose persistent memory programs, by

[1] A hot-spot occurs when large streams of updates are issued by different transactions over a small memory region [4].

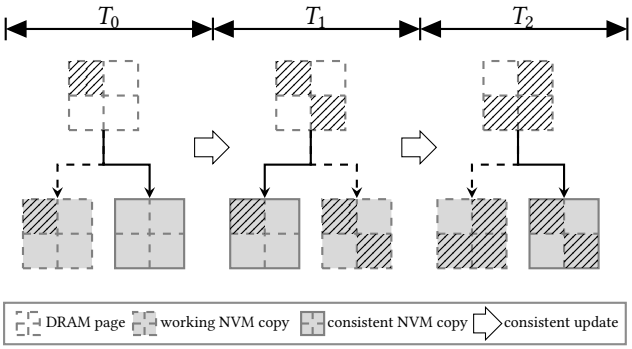


Figure 1. Crash-consistent page modification based on dual-versioned persistent copies.

periodically enforcing a global quiescent state to occur. Additionally, PMThreads integrates this with dual-copy based memory persistence.

3 PMThreads Design Overview

PMThreads provides persistence for FASEs delineated by the outermost lock and unlock pairs in lock-based programs following programming model of NVthreads. Persistent writes must be issued only within critical sections (FASEs), and all applications must rely on locking primitives provided by the `pthread` library. A persistent memory allocation interface `nvmalloc` is provided for user programs.

Overall, PMThreads works following the five aspects that are described in the next subsections: Section 3.1 the dual persistent copy framework for data objects allocated by `nvmalloc`, Section 3.2 memory access write monitoring, Section 3.3 enforcing a quiescent state to capture consistent state to NVM, Section 3.4 failure-atomic version switching, and Section 3.5 failure recovery.

3.1 The Dual-Copy Framework

The dual-copy framework is similar to a conventional dynamic memory allocation using `malloc`, except that user programs acquire persistent memory using our `nvmalloc` interface. For each persistent page allocated to a user program, the PMThreads runtime maintains a shadow DRAM page that is assigned and associated with one *working copy* and one *consistent copy* in NVM.

A user application directly interacts only with the DRAM copy, and the PMThreads runtime is responsible for quiescent state operations where DRAM shadow page write updates are transferred to the corresponding durable NVM regions, and for performing the role-switching between *working* and *consistent copies*.

As illustrated in Figure 1, each time we persist data from the DRAM-side to the NVM-side, only one persistent copy, referred to as the *working copy*, will be modified. The other persistent page will be regarded as the *consistent copy*. Rather than persisting undo or redo log entries before touching the target NVM region (this involves two-round-trips to NVM), during a global quiescent state, we perform in-place updates directly to the *working copy* and leverage the *consistent copy* as a fallback to ensure failure-atomicity. Dynamic role switching changes between *working copy* and *consistent copy* are part of the version update policy of PMThreads. As we only make the DRAM-side memory accessible to user programs, version switching on the NVM-side will be fully transparent, and no application modifications are required. The resource consumption costs of the dual-copy mechanism incurs the additional cost of double-sized persistent NVM memory allocation compared to a conventional memory allocator.

For ease of implementation, and to ensure that our results are comparable to NVthreads, we have used eager NVM page allocation, but lazy allocation could be used to reduce additional memory overheads to 2x of the working set size of persistent memory.

Note that the dual-copy mechanism *avoids the costs of logging, and also eliminates the extra memory usage for storing log entries*. Furthermore, because of dynamic version switching between dual copies, PMThreads enforces memory persistence with only one-round-trip to NVM. As in Dthreads [26], PMThreads leverages page-diffing to determine page updates. Byte-by-byte comparisons between the DRAM-side copy and the *working copy* are performed to minimise persistent writes to NVM. Therefore, the dual-version mechanism should achieve better wear-leveling than logging-based solutions due to a reduced number of writes. PMThreads only stores changes, rather than every store instruction to the consistent copy.

3.2 Memory Access Monitoring

In PMThreads, we prefer transparent memory modification tracking approaches, based on compiler instrumentation as used by Atlas, and a virtual memory protection based approach as used by NVthreads. This enables PMThreads to avoid the need for programmers to wrap persistent writes with special library functions, as in [13]. Though PMThreads employs page-level dual-copy management, we still observe that excessive `TLB misses` triggered by page protection faults can prevent PMThreads from accomplishing efficient memory tracking.

On the other hand, compiler instrumentation, could be used as a page-level tracking scheme, for example, by hashing tracked addresses to their corresponding page boundaries as a substitute for expensive page-based memory protection solutions. However, word-level memory tracking still needs to intercept each store instruction.

Regarding the above design considerations, we provide two versions of PMThreads, where each persists data at page-level granularity, *PMThreads-I* tracks DRAM memory modifications at word-level, and *PMThreads-P* that leverages page protection mechanisms to monitor DRAM writes

at page-level. We observe that fine-grained tracking and coarse-grained persistence results in substantially improved performance, Section 6 discusses this in detail.

3.3 Capture Consistent State to NVM

Memory access tracking records the dirty DRAM locations, but we must also determine when to persist the buffered updates to NVM. Determining when to capture and preserve the memory state for persistence is challenging for lock-based code, and significantly different from transactional memory systems, that typically guarantee isolation between concurrent memory transactions.

In lock-based programs, especially ones having nested critical sections, intermediate updates in one FASE can be visible to other ones. For a FASE F_B that observes the in-progress persistent states of FASE F_A, then F_B is dependent on F_A. Thus, to ensure correct recovery, if the early FASE F_A fails, then the effects of F_B have to be aborted, and this can lead to complex dependencies. Dependencies between critical sections can be described as related to i), an inner critical section that is completely surrounded by an outer critical section (perfect nesting), ii) overlapping nested critical sections, such as occurs in lock-chaining, and iii), critical sections that require condition variables to signal and wait on changes in application synchronization states.

To address the problem of dealing with arbitrary interleaving patterns for critical sections, NVthreads adopts the multi-threading paradigm proposed by Dthreads [26], this converts conventional threads running in a shared address space into child processes with separated address spaces, that enables isolated thread execution. To preserve the original multi-threaded semantics, key synchronization points in program execution, such as thread creation/destruction, thread joining, lock acquisition, lock release, etc., are intercepted by the runtime to sequentially merge the modifications to shared state. However, this is at the cost of heavyweight process context switches that are more expensive than switching threads within a single shared process address space.

In addition, when threads in isolated address spaces modify the same shared page, each thread will generate its own private redo log entries, leading to further increases in writes to NVM. Essentially, this involves merging threads updates in different process address spaces to the shared memory. Further, the isolated threads will suffer from inefficient fine-grained synchronization operations due to excessive global barriers and memory page copies between private and shared address spaces. This is because at each synchronization point, isolated threads need to commit their local changes from private memory to shared memory resources.

Addressing these issues, PMThreads does not employ isolated execution in different processes, instead it keeps threads in a single shared memory space and takes control over isolation by enforcing a quiescent application state that exploits careful interception of lock acquisition and lock release operations. As mentioned previously, PMThreads expects writes to persistent regions to be protected by critical sections. Thus, PMThreads assumes that when a thread does not hold a lock, it will not issue writes to persistent memory. We consider such a thread to be in a *quiescent thread state*.

```
bool crashed();
void* nvmalloc(size_t sz, char* handle);
void* nvrecover(void* v, size_t s, char* handle);
int main() {
  int* p;
  if(crashed())
    nvrecover(p, sizeof(int), "Integer");
  else
    p = (int*) nvmalloc(sizeof(int), "Integer");
  return 0;
}
```

Figure 2. Failure recovery API [16].

Through intercepting lock acquisition and lock release operations, PMThreads can capture the states of each live thread and how many locks it holds. When locally observing a point at which a thread becomes quiescent, we can instruct it to enter a barrier to wait for the globally quiescent thread state. The exact operation of PMThreads in this context is described in Section 4. Once a global quiescent state is reached, PMThreads can ensure that data in the DRAM-side will not be corrupted while it is moved to the NVM-side home locations. After making the captured consistent states durable, PMThreads will return control back to the application program to continue from its suspended execution.

3.4 Failure-Atomic Version Switching

PMThreads can execute in-place modifications from DRAM onto the NVM *working copy* because it maintains dual persistent copies. Each time PMThreads successfully makes the *working copy* consistent, roles between the *working copy* and *consistent copy* are switched. Thus, persistent data is updated to point to the new consistent version. However, the dual-copy mechanism is only capable of ensuring failure-atomicity for updating a single page. When persisting multiple pages together, role switching itself needs to occur failure-atomically to guarantee memory persistence for the whole persistent data. Addressing this problem, we propose a failure-atomic version switching algorithm, to guarantee that roles switching between multiple dual-copy pairs will be atomically visible to application programs.

3.5 Failure Recovery

NVthreads, must replay redo log entries to the post-recovery address space, while PMThreads can directly map consistent durable copies into shadow memory via the copy-on-write mechanism of the operating system. Having put the persistent heap into a consistent state, further recovery tasks

are considered as application-specific in PMThreads. We implement the recovery interfaces proposed by NVthreads to facilitate the user-defined recovery process (see Figure 2, each durable object is attached with a user-defined `handle` in `nvmalloc`, that is used in conjunction with the `nvrecovery` method to recover a variable after a crash. Note that the `crashed` function returns a boolean value to differentiate between initial execution and a recovered one. As is the case with other failure-atomicity runtimes, failure recovery in PMThreads is orthogonal to the object-specific recovery as proposed in [9] and [31].

4 Implementation of PMThreads

This section describes (i) the implementation for the interception of lock acquisition and lock release functions related to algorithm 1, (ii) the details of failure-atomic version switching in algorithm 2, and (iii) the recovery from crash procedure for PMThreads.

4.1 Persist Interval and Lock/Unlock Interception

PMThreads requires a global quiescent period, during which there will be no thread holding any mutex lock, in order to be able to move DRAM buffered writes consistently to the NVM pages. Quiescent states are enforced periodically, using a configurable Δ_T persist interval, where all threads execute a `pmthreads_barrier` in order to safely persist modified shared data.

PMThreads intercepts the mutex lock and unlock functions from the standard `pthread` library, and maintains thread local lock counters L_t for each thread `t` to observe the quiescent status of each individual thread. In addition, PMThreads assigns each thread t with a thread local timing duration δ_t. Each time $L_t$drops to zero, the thread will check its local δ_t against Δ_T to determine whether it is necessary to continue to execute or, if it must wait on a `pmthreads_barrier` in order to persist the DRAM buffered writes to shared storage. Therefore, each thread locally determines when it must wait on a `pmthreads_barrier`. In this way, we enforce global quiescent states to occur where it is safe to persist data. Threads can only exit the barrier, once all threads that potentially issue persistent memory writes in the current persist interval have entered the barrier, and the persist operation has completed.

Hereafter, we refer to such potential persistent writer threads as *live pmthreads*. PMThreads keep records of the total number of *live pmthreads* as Σ_{pm}. In detail, following the practice of Dthreads[26], we intercept `pthread` library functions to let them call into `pmthreads_enter` or `pmthreads_exit` to register or unregister the intention of a thread to write to persistent memory. It is intuitive to intercept `pthread_create` with `pmthreads_enter`, and intercept `pthread_exit` with our `pmthreads_exit`. For function calls that may result in the calling thread blocking, such as condition wait, thread join, etc., our interceptions will firstly deregister the calling thread with the PMThreads

Algorithm 1: *pthread* interception

```
globally shared : Δ_T, Σ_pm, ω, PM_cond, PM_mtx
thread local    : L_t, δ_t
func pthread_mutex_lock_hook(mutex)
    L_t ← L_t + 1 ;
    return pthread_mutex_lock(mutex);
end
func pthread_mutex_unlock_hook(mutex)
    pthread_mutex_unlock(mutex);
    L_t ← L_t - 1;
    if L_t == 0 then
        update δ_t;
        if δ_t > Δ_T then
            pmthreads_barrier();
        end
    end
end
func pmthreads_barrier()
    pthread_mutex_lock(PM_mtx);
    ω ← ω + 1;
    if ω < Σ_pm then
        pthread_cond_wait(PM_cond, PM_mtx);
    else
        persist();
        ω ← 0;
        pthread_cond_broadcast(PM_cond);
    end
    pthread_mutex_unlock(PM_mtx);
end
func pmthreads_enter()
    pthread_mutex_lock(PM_mtx);
    Σ_pm ← Σ_pm + 1;
    pthread_mutex_unlock(PM_mtx);
end
func pmthreads_exit()
    pthread_mutex_lock(PM_mtx);
    Σ_pm ← Σ_pm - 1;
    if ω == Σ_pm ∧ Σ_pm>0 then
        persist();
        ω ← 0;
        pthread_cond_broadcast(PM_cond);
    end
    pthread_mutex_unlock(PM_mtx);
end
```

Algorithm 2: Failure-atomic version switching

```
func persist()
    for each dirty page p in DRAM do
        p′ ← NVM copy of p with smaller seq;
        p′.seq ← VER_d + 1;
        clflush p′.seq;
        sfence;
        update p′ with diff_bytes(p, p′);
        clflush diff_bytes(p, p′);
    end
    sfence;
    VER_d ← VER_d + 1;
    clflush VER_d ;
    sfence;
end
```

runtime, then execute the original `pthread` function. Meanwhile, we intercept the return point where the blocked thread will be woken up, and there a thread is re-registered to the PMThreads runtime. If a thread performs unregistration with the PMThreads runtime in a locally non-quiescent state, such as a condition wait, the PMThreads runtime will check if a thread has performed any modifications to DRAM shadow pages. If it has issued no such writes, the PMThreads runtime will allow a thread to directly leave the runtime. Otherwise, the runtime will temporally turn off the periodical quiescence enforcement until this thread is back to active.

Note that Dthreads injects a global barrier for all synchronization operations, thereby serializing all parallel operations and incurring significant synchronization overhead. PMThreads need only periodically enforce a global barrier to ensure the persist interval Δ_T is respected.

4.2 Failure-Atomic Version-Switching

Algorithm 2 enforces failure-atomicity for version switching, we maintain a global latest durable version number (VER_d), that is atomically incremented at the end of the *persist* procedure. Meanwhile, each persistent copy is associated with a local version number (*seq*) that is used to distinguish the *working copy* from the *consistent copy* in a dual-version shadow pair. More specifically, during the *persist* procedure, the persistent copy with smaller *seq* in a dual-version pair becomes the *working copy*. Right before being updated, the *seq* of the *working copy* is set with the version number next to the current one (VER_d + 1).

In the case that the *persist* procedure is interrupted by a system crash, the *recover* process leverages the VER_d and the *seqs* to reach the last consistent state before the crash. During crash recovery, the retained VER_d will effectively denote persistent copies with *seqs* larger than the VER_d as inconsistent. In case both *seqs* of a dual-version pair are

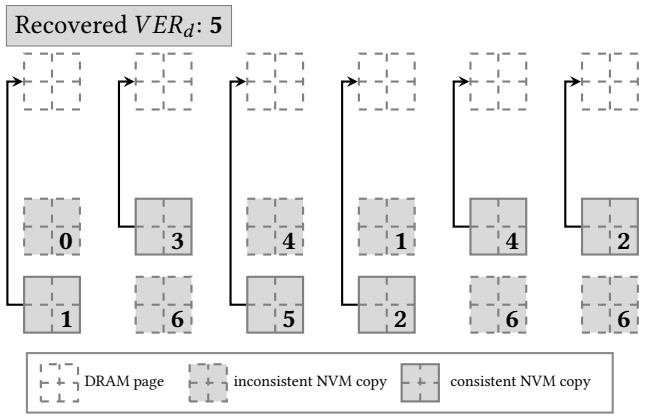


Figure 3. Failure recovery process.

no larger than the retained VER_d, the one with larger *seq* will be the *consistent copy*. In summary, updating the *seq* of a *working copy* is represented as a version-switch local to the dual-version shadow pair, which cannot be visible to the post-crash stage unless the *persist* procedure completes normally. Thus, version switching for multiple dual-copy pairs will take effect atomically with the atomic increment of the global VER_d.

4.3 Recovery

With dual-copy based memory persistence assurance and failure-atomic version-switching, the PMThreads runtime is capable of recovering a persistent region to the consistent state before a crash occurred. The recovery process of PMThreads essentially involves making the consistent copy in each dual-copy pair visible to the user program.

As illustrated in Figure 3, the consistent versions can be recognized through the global VER_d and the *seqs*. The recovered VER_d (5 in the example) indicates the last consistent global version before a crash. The retained *seq* value (depicted in the bottom-right part of each persistent copy) reveals whether it has been enforced to be consistent. Accordingly, the *seqs* that are larger than the recovered VER_d reflect interrupted updates, that were assigned to modified pages during the interrupted update procedure. We could infer that the runtime was trying to move forward to version 6, but a failure occurred. Thus, persistent copies with *seqs* larger than 5 are recognized as holding inconsistent information. For dual-version durable pairs with both *seqs* no larger than the recovered VER_d, the copy with the larger *seq* in the pair will be the consistent copy.

We abstract persistent memory pages as NVM-backed files and leverage the `mmap` interface to map consistent copies into the shadowing address space. We make the recovered DRAM-side shadow persistent pages as copy-on-write mappings of the consistent copies to facilitate the recovery procedure, which is implemented through calling `mmap` in combination

with the MAP_PRIVATE flag. After recovering the persistent heap to a consistent state, further recovery is application-specific, and directly related to how to restart the application's processing from a consistent persistent storage state. We keep track of the base address of the DRAM-side memory in persistent memory and try to map that region at the same virtual address (calling `mmap` with MAP_FIXED) after restarting. If we cannot establish the fixed mapping, offsets between the new base and the previous one will be provided to the application-specific recovery procedure.

5 Discussion

PMThreads-I requires the source code of user programs and third-party libraries that could access persistent memory to be available for compile-time instrumentation. PMThreads intercepts standard C library functions such as *memcpy*, *strcpy*, etc. to track memory locations being touched by function calls. Related work, such as Breeze [31], complements PMThreads to address these issues. Durable writes should only be issued inside critical sections, this allows the coexistence of transient and persistent memory allocations. As transient contents will not be retained, user programs need to carefully manage the interaction between the two kinds of dynamic memory, for example during application-specific failure recovery routines. Further, any memory pointers values in persistent fields can become invalid if the recovery procedure maps NVM pages into different virtual memory addresses. Object-oriented recovery systems, such as [9], complement our work.

PMThreads can be adjusted to operate its dual-copy and failure-atomic role-switching at sub-page granularity. Memory tracking can be adjusted by, i), compiler instrumentation hashing schemes that track writes at sub-page boundaries, and ii), page protection monitoring using schemes such as proposed in [27] to achieve sub-page granularity. The failure-atomic role switching can be adjusted to use similar techniques. Making PMThreads work at sub-page granularity mainly adds additional complexity to the recovery procedure. With sub-page granularity, consistent contents of a single page may be distributed into multiple dual-copy pairs. The recovery procedure needs to integrate the subpage-sized *consistent copies* into complete compact page(s), as sub-pages cannot benefit from existing operating system mechanisms, such as `mmap`.

With Kamino-Tx [30] and NV-HTM [4], the write updates of a transaction will effectively reach a durable state immediately after committing and be persisted in NVM. In contrast, PMThreads provides lower overhead and latency (see Section 6 using DRAM plus a dual copy NVM memory without providing immediate durability guarantees for FASEs. Instead, PMThreads achieves durability periodically.

For example, with DudeTM [24] and PMThreads, the write set of a transaction, or FASE, is buffered in DRAM and then asynchronously persisted to NVM (durable). This helps to hide the NVM access latency. To bound the asynchronicity, and differently to DudeTM, PMThreads use an interval Δ_T to specify the maximum duration between FASEs persisted to NVM.

For some applications, it may be necessary to ensure that a FASE, or a transaction, is durable to perform an externally visible event; e.g. an operating system system call that is expected to modify persistent storage consistently. Such applications require an explicit acknowledgement of the durability of a transaction or FASE. For these applications, PMThreads can implement an acknowledgment process similar to DudeTM (see next paragraph), or can implement an NVM-fence operation for insertion between FASEs to force a specific FASE to wait until its predecessors have completed and persisted their write updates.

DudeTM allow applications to read the latest globally persisted durable transaction ID and compare this with a thread local transaction ID to determine if a transaction has been committed to NVM. Following this approach, PMThreads can exploit its failure-atomic version switching algorithm (described in Section 4.2) so that application threads can obtain the latest global durable version number VER_d. In addition, each thread needs a FASE ID F_i that captures the maximum local durable version number so that durable FASEs can be determined by evaluating whether $F_i \leq VER_d$.

6 Evaluation

All the experiments are executed on a two-socket NUMA machine running Ubuntu 16.04, kernel 4.15.0 (Hyper-threading disabled). Each socket has an Intel(R) Xeon(R) E5-2670 v2 @2.50GHz, containing 10 physical cores, giving a system total of 20 physical cores (no hyperthreading). Each Xeon chip shares 25MB of L3 cache between its 10 cores, and each core has 64KB L1 cache and 256KB L2 cache. Each socket is populated with 16GB of DRAM, providing a total of 32GB main memory. The frequency of each core is fixed to 2.0GHz using `cpufrequtils`. All benchmarks are built with clang++ 7.0.1, except for the Mnemosyne counterparts that required g++ 5.4.0. The compiler instrumentation is implemented in LLVM 7.0.1. All figures report the average of 10 execution runs.

6.1 NVM Emulation

NVM is emulated with DRAM as in [10, 13, 43], where NVM read latency is discarded and an extra delay of 100ns is injected to a `clflush` operation (`clflush-opt` was not available on our machine.) to model the increased write latencies of NVM technologies. The delay is obtained by looping on the timestamp counter of the processor via the `RDTSC` instruction as in [18, 24]. The `sfence` instruction is also used to enforce ordering for non-volatile writes.

Table 1. Geometric mean slowdown of PMThreads-I, PMThreads-P, NVthreads and Atlas (lower is better).

System	PMT-I	PMT-P	NVthreads	Atlas
Parsec	**1.43**	1.95	7.70	2.35
Phoenix	2.22	**1.39**	2.08	4.69

6.2 Evaluated NVM Systems

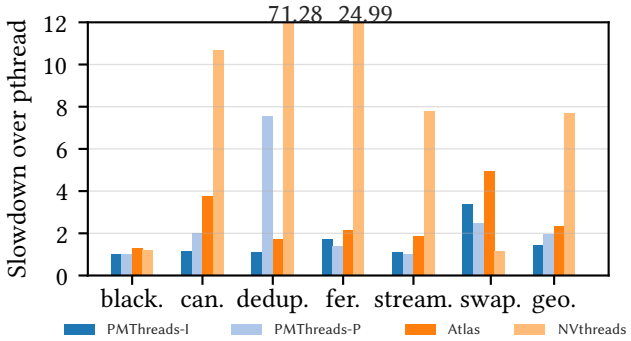
Atlas [5] uses word-granularity for compiler instrumentation to track and persist memory writes, and also for undo logging to enforce memory persistence.
Mnemosyne [42] is a transactional memory system based on word-level redo logging.
NVthreads [16] executes a single shared address space multi-threaded program as a multi-process program having multiple address spaces where virtual memory protection is used to track persistent memory access at page granularity. It leverages NVM filesystem interfaces to maintain at page-granularity redo log entries for crash recovery.
PMThreads-I tracks persistent stores at word-granularity with compiler instrumentation. Failure-atomicity of persistent data modification is guaranteed by our dual-copy mechanism.
PMThreads-P is similar to PMThreads-I except that it uses the hardware page protection mechanism to monitor persistent memory accesses. Note that the **persist interval**, Δ_T, of all PMThreads implementations is configured to be 100ms (i.e. equal to the time quantum of the OS scheduler on the dual-socket test machine) for all experiments, unless otherwise specified.
Pthread acts as the 'ideal' baseline in the experiments. It runs the benchmarks on DRAM using the standard `pthread` library; no persistent data.

6.3 Parsec and Phoenix

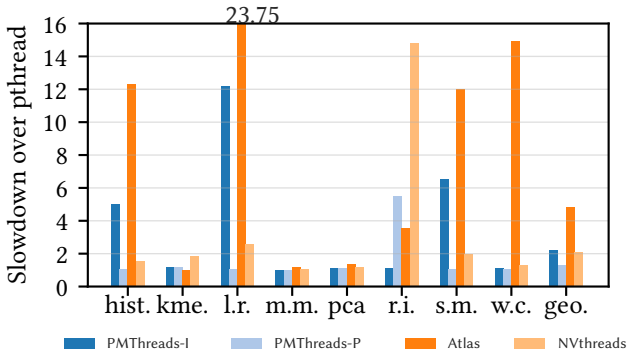
The 14 multi-threaded benchmarks from PARSEC [3] and Phoenix [39] suites (with our ports to ATLAS) are used for evaluation, following the NVthreads methodology [16]. We analyze the potential overheads imposed by memory access monitoring and memory persistence enforcement by considering all memory allocations to be persistent.

Normalized to Pthread using 20 threads, Figures 4a and 4b present the runtime overhead of PMThreads-I, PMThreads-P, Atlas, and NVthreads. Table 1 summarizes the geometric mean (geo.) performance slowdown in comparison to PThread and find that PMThreads-I (1.43), and PMThreads-P (1.39) achieve the best performance for Parsec and Phoenix benchmarks, respectively.

PMThreads-I mostly incurs less than 20% slowdown for Parsec, except for `ferret` (fer.), and `swaptions` (swap.); with approximately 2× and 3.4× slowdowns respectively. PMThreads-I has the best performance for 3 Parsec benchmarks while PMThreads-P has the best performance for the other



(a) Parsec slowdown results with native as baseline.



(b) Phoenix slowdown results with native as baseline.

Figure 4. Parsec and Phoenix.

two; `ferret` (fer.) and `stream`. Despite the big slowdowns with `dedup.` and `fer.`, NVthreads achieves the best performance for the `swaptions` benchmark.

The main source of PMThreads-P overhead comes from hardware page protection based memory tracking. NVthreads and PMThreads-P incur the overhead of write protecting pages in the persistent heap (at least the already allocated region) to track potential memory writes, which inevitably results in extra `TLB misses` and page fault handling overheads. This is demonstrated by the high performance penalty imposed by NVthreads (circa 71x slowdown) and PMThreads-P (circa 7x slowdown) in the more lock-intensive benchmarks, such as `dedup`.

NVthreads and PMThreads-P outperform Atlas and PMThreads-I when the workloads contain no locking, such as in `blackscholes` (black.) and `swaptions` (swap.). In `streamcluster` (`stream.`), a condition variable is used for synchronization. As a consequence, the results for `streamcluster` reveal the dependence tracking overhead of NVthreads and Atlas. Figure 4a shows that both PMThreads-I and PMThreads-P incur moderate overheads of circa 1.13× and 1.04× slowdown in comparison with nearly 2× and almost 8× slowdown faced by Atlas and NVthreads. The greater than 10×

slowdown of NVthreads in canneal (can.) is due to excessive memory copying between thread-private memory and shared memory. This overhead is specific to the isolated thread implementation using separate process address spaces.

Figure 4b gives the results for Phoenix benchmarks that contain no lock except for reverse index. Such lock-free benchmarks serve mainly to demonstrate the overheads of running applications under persistent runtimes where no updates to persistent memory are present, because all systems assume that writes occur inside critical sections that are protected by a lock. As a consequence, NVthreads and PMThreads-P degrade performance marginally in many Phoenix benchmarks. Three outliers are kmeans (kme.), linear regression (l.r.) and string match (s.m.), where NVthreads presents 1.8× to 2.6× slowdown, whereas the overhead introduced by PMThreads-P is below 20% in these workloads.

The kmeans (kme.) benchmark creates and destroys numerous short-lived threads during its execution, consequently, its results demonstrate the performance penalty of converting threads into separate processes for isolated thread execution in NVthreads at nearly 2x. In contrast, the Pthread library interceptions of PMThreads incur less than 20% overhead, even for kmeans (kme.). Atlas suffers high tracking overhead in histogram (hist.), linear regression (l.r.), string match (str.) and word count (w.c.), where word-granularity write instrumentation costs dominates the total execution time of the program. PMThreads-I uses a similar memory access monitoring mechanism to Atlas, and it incurs significant memory tracking overhead in histogram, linear regression and string match, where PMThreads-I gives respective slowdowns in comparison to PThread of circa 5.0×, 12.1× and 6.5×. The slowdown of Atlas in word count (w.c.) is due to extensive logging overhead incurred from memmove function calls, whereas PMThreads-I performs library interception, and only performs logging when such calls are invoked inside a critical section.

Overall, Table 1 shows that PMThreads-I achieves more than 5.38× speedup over NVthreads and is 1.64× faster than Atlas on average in the 6 Parsec benchmarks. In addition, PMThreads-P is over 50% slower than PMThreads-I in these workloads. For the Phoenix benchmarks, the memory tracking overhead incurred by PMThreads-I is less than 50% of that of Atlas. NVthreads is 40% slower than PMThreads-P.

6.4 Stress Tests

PMThreads scalability is evaluated in comparison to Atlas, NVthreads and Mnemosyne, using different write-update loads, on two different lock-based concurrent data structures. The overheads of mutex interception are also analyzed. The microbenchmarks issue repeated concurrent access to a shared data structure with varied ratio of read and write operations. The data structures considered are:

- FAST-FAIR [18], a concurrent persistent B+Tree. We comment out the persistence-related operations in FAST-FAIR, such as clflush, memory fence, etc., and instruct the tree node constructor (destructor) to allocate (release) tree nodes with nvmalloc (nvfree).
- Lock-Coupling List, a concurrent ordered linked list from ***synchrobench*** [14]. Again, we replace malloc and free with nvmalloc and nvfree for persistent heap management of linked list nodes.

Considering Mnemosyne, we remove the lock and unlock operations in FAST-FAIR and Lock-Coupling Linked List to obtain the corresponding sequential versions. Then, we port the sequential data structure to Mnemosyne via wrapping data structure manipulations with the transactional interfaces of Mnemosyne.

We analyse the scalability of our two-socket target system via execution with the 1, 2, 4, 8, 10, 16, 20, 32, and 40 threads. We capture deviations in performance trends expected beyond one socket with 10 cores, i.e with threads between 10 to 20, and also when the number of threads per core is over committed across both sockets with 32 and 40 threads. In our experiments, we consistently pin each thread to a specific processor core. We avoid use of the second CPU socket when the thread count is 10 or less, and for the over committing cases, we pin threads in a round-robin order across the 20 available cores.

In this section, we refer to data structure operations that issue writes to shared memory as PUT operations, and this includes both insert and delete functions. While performing a PUT operation, the threads will randomly insert or delete an element from the data structure. Read-only operations, such as search, are regarded as GET operations. We vary the percentage of PUT operations for all the experiments as 10%, 50% and 90%. For each configuration, we run for a fixed number of operations and measure the aggregated throughput achieved by all the threads. For the B+tree, we generate 2,000,000 8-byte keys following the uniform distribution. The microbenchmark program dynamically generates 8-byte values for each insert. For all the evaluations towards B+tree, we insert 1,000,000 key-value pairs during the warming-up phase, and measure the aggregated throughput achieved in 1,000,000 operations under different PUT/GET ratios. For the ordered linked list, the stress test program inserts 40,000 uniformly distributed elements to the list during warming up, and performs an extra 40,000 list operations with the configured PUT/GET ratio. Also, we measure the aggregated throughput of all the threads.

Figure 5 gives our stress test results of B+tree. We use the volatile FAST-FAIR (with persistence-related operations commented out) running with standard pthread library as the baseline program (PThread in Figure 5). To investigate the impact of pthread interception in PMThreads, we also present the results of PMThreads-ZERO, which does not

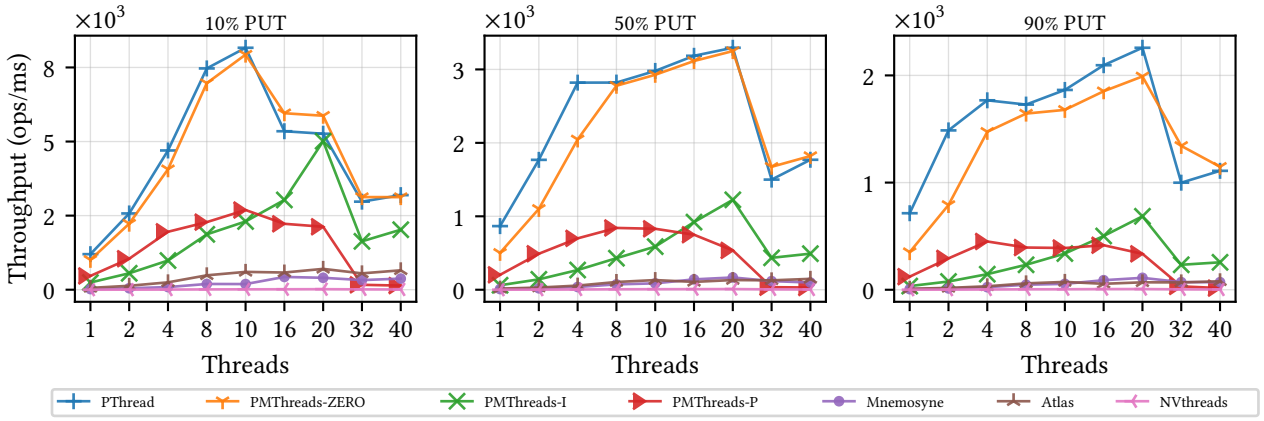


Figure 5. B$^+$tree stress tests.

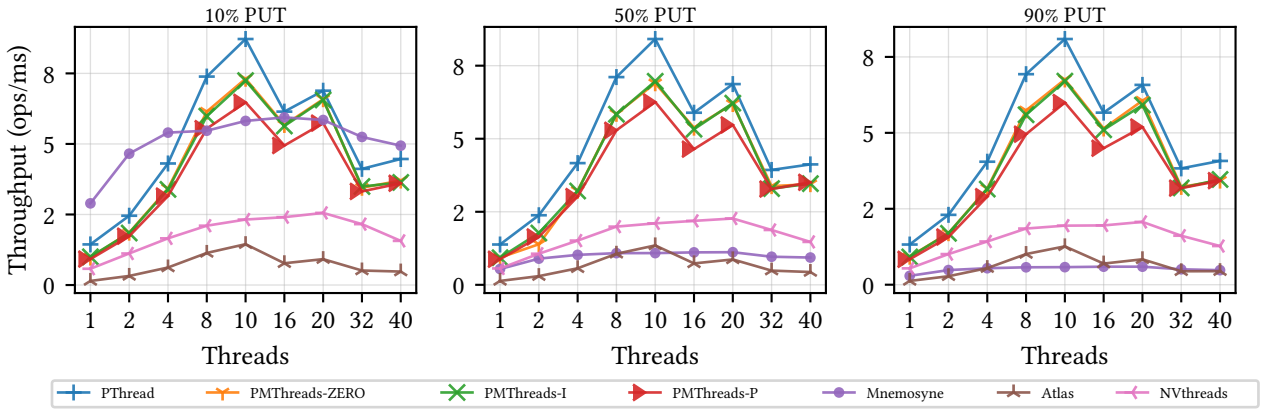


Figure 6. Linked List stress tests.

enforce durability for persistent memory accesses (it neither instruments write operations nor performs page protection) but merely performs the necessary interceptions to `pthread` library by PMThreads. The figures demonstrate that PMThreads-ZERO scales in a similar manner to Pthread with all the stress-test evaluated workloads, therefore we conclude that interception overhead is not significant. PMThreads-P outperforms PMThreads-I when the number of worker threads is less than 10. PMThreads-I achieves improved performance to PMThreads-P as the number of threads increases over 10. Considering scalability, PMThreads-I scales to 20 threads with all workloads, whereas PMThreads-P only scales to 10 threads with the 10% PUT workload. Under 50% and 90% PUT rates, PMThreads-P only scales to 8 and 4 threads, respectively. With overcommitted workloads, the throughput of PMThreads-P drops dramatically, it incurs 2.8× to 7.8× slowdown over a single thread. whereas the slowdown trend of PMThreads-I is similar to that of the native PThread.

Stress results over the ordered linked list appear in Figure 6. In the less-contended cases (10% PUT with 1, 2, 4 threads), Mnemosyne achieves the best performance. However, its performance reduces significantly in comparison to PMThreads-I and PMThreads-P with increasing contention. Note that we port sequential versions of the evaluated data structures, containing no function calls to lock and unlock, whereas for all other systems, such calls are still invoked with 1 thread. Pthread scales up to 10 threads with all the evaluated inputs. Both PMThreads-I and PMThreads-P present a similar scalability curve to the Pthread baseline. Significantly less data is persisted in the linked list (where lock/unlock interception costs dominate overheads, leading to similar performance for PMThread-I and PMThread-P) in comparison to the B+tree.

6.5 Memcached

Memcached [32] is a memory object querying system, that caches key-value pairs in main memory to boost web service performance. We investigate the performance overhead of data persistence enforcement in Memcached-1.2.4 (adding `nvmalloc` calls to create a persistent heap) using the

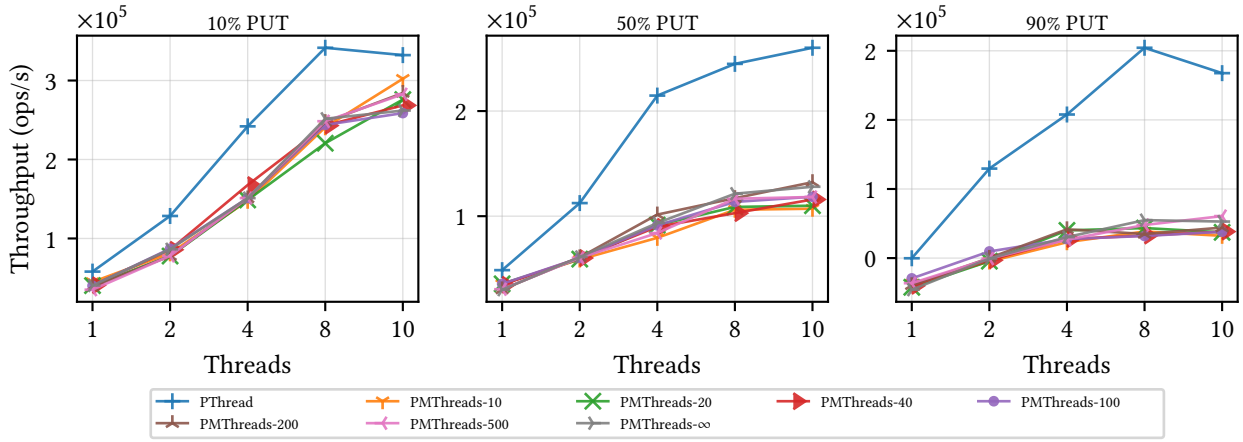


Figure 7. Memcached throughput scalability test.

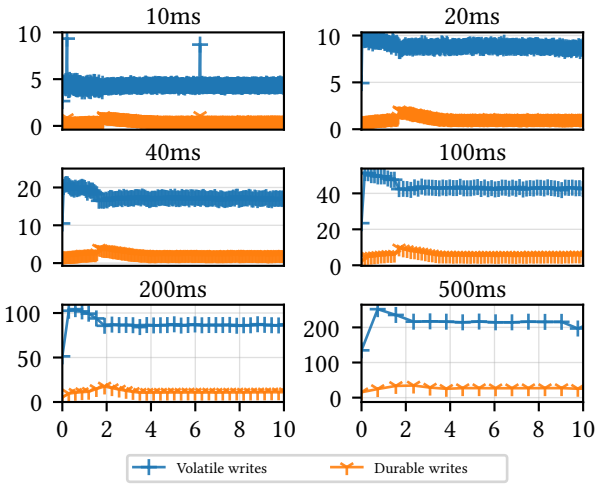


Figure 8. Memcached volume of NVM data written at each quiescent state. The x-axis depicts running time in seconds. The Y-axis depicts the volume of data in MB written to NVM with Δ_T varying from 10ms to 500ms. Each configuration runs for 10 seconds.

memtier_benchmark [40] to generate requests to the Memcached server. On the dual-socket machine, we bind the Memcached server to one socket and the *memtier_benchmark* executes on the other socket. An unmodified Memcached running with standard `pthread` (and no persistence) is used as an 'ideal' baseline.

Figure 7 presents the performance of PMThreads-I with different Δ_T configurations. The label PMThreads-∞ represents PMThreads with no quiescence enforcement. We use different numbers of Memcached server threads to examine the scalability of PMThreads.

Figure 8 presents for each quiescent point enforced by PMThreads (x-axis), the volume of data persisted (y-axis)

Table 2. Average volume of persistent writes reduced by PMThreads per interval.

Δ_T (ms)	10	20	40	100	200	500
Data Volume (MB)	3.7	7.7	15.2	37.8	76.1	176.5

with different Δ_T configurations for the `memtier_benchmark` running for a fixed time of 10s, using a 1:1 PUT-GET ratio workload. In the figure, (*volatile writes* are updates to dynamic persistent memory objects allocated by `nvmalloc` issued by user programs, that are buffered by the DRAM-side shadowing memory, whereas *durable writes* are the exact volume of dirtied shadowing memory moved by the PMThreads runtime from DRAM to NVM at the end of one persist interval.). PMThreads significantly reduces the writes to NVM through buffering persistent writes leveraging the DRAM-side shadow memory, and the average volume of persistent writes reduced by PMThreads per interval is listed in Table 2 (i.e. the average of total writes to shadow DRAM minus actual writes to NVM at each persistent interval). Such reductions will improve NVM wear-levelling. With longer Δ_T, PMThreads generates fewer interventions, while moving more data at each persistence point. Figure 8 also clearly demonstrates that PMThreads is capable of enforcing quiescent states.

To measure the overhead of `pmthreads_barrier` in Memcached, we collect statistics of time consumption in crossing the barrier from the 100ms test with 50% PUT rates. As depicted in Figure 9, we observe rising average delays as the number of worker threads increases. This is expected behaviour when barriers involve more threads.

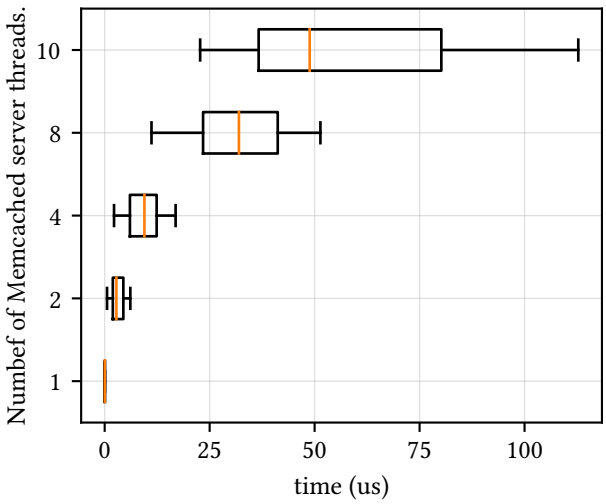


Figure 9. Memcached synchronization overhead boxplots.

Table 3. B+Tree. Very little lost inserts with Δ_T=10ms, whereas PM-100 and PM-500 suffers 1.3% and 22.3% data lost, respectively.

Configuration	PM-10	PM-100	PM-500	Atlas
Inserted	599992	1146872	1271418	164970
Recovered	599989	1132363	979564	164970
Recovery Time	4.39ms	6.55ms	5.30ms	8.20s
Volume	29MB	51MB	36MB	860MB

Table 4. Linked List. PM-10 incurs negligible lost inserts. Percentage of data lost in PM-100 and PM-500 are 1.2% and 15.4%, respectively.

Configuration	PM-10	PM-100	PM-500	Atlas
Inserted	24675	25231	24233	6379
Recovered	24675	24940	20505	6379
Recovery Time	0.71ms	0.76ms	0.57ms	108.8s
Volume	4.4MB	4.5MB	3.5MB	602MB

6.6 Recovery and NVM Consumption

Tables 3 and 4 present crash recovery results for Atlas and PMThread-I running 10-threads B+Tree and Linked List write-heavy workloads (insert only), respectively. The workloads are executed for a fixed duration of 2000ms, then the program is forced to 'crash'. A measurement is taken of how many keys were *Inserted* prior to forcing the crash. The *Recovery Time* taken to recover the persistent crashed program, and the *Recovered* number of successfully persisted keys are displayed along with the total *Volume* of persistent storage used. Elements that are failed to be durably written to persistent memory (because the abort occurs before they are persisted) are denoted as *lost* ones. We compute the percentage of *lost* elements to compare the recovery efficiency of PMThreads with different Δ_T configurations.

Atlas presents longer recovering time than PMThreads; at least three orders of magnitude — see row labeled Recovery Time in Tables 3 and 4). Atlas needs to replay its fine-grained (word-level) log entries during crash recovery. We also observe significantly higher `Volume` NVM consumption in Atlas, because Atlas maintains per-thread logging and tracks each persistent write at word-level. On the other hand, PMThreads maintains dual NVM pages, and its writes are buffered in the DRAM shadow pages. The results show that the NVM memory overheads are significantly lower for PMThreads than logging based solutions such as Atlas.

We perform a best-effort fix-mapping in preference. Otherwise, a non-volatile pointer fixed tool-chain, such as [9], would be a practical fallback for the crash recovery of PMThreads. As with NVthreads, the recovery procedure is application-specific, PMThreads relies on programming efforts to establish the recovery logic. For pointer-based data structures, the recovery process of PMThreads only requires the recovery of a *root* persistent pointer, the head of a linked list, root node of a B+Tree, etc. Whereas NVthreads must traverse over all the non-volatile pointers, imposing significant further burdens to programmers and the runtime system. Thus, we omit further recovery information for NVthreads.

7 Conclusions

Motivated by NVM technologies, we have presented the PMThreads system, a novel user-space runtime that provides transparent memory persistence for traditional lock-based parallel programs. The runtime maintains dual-versioned persistent data in NVM and only a single DRAM copy of each page is transparently accessible to the application. Write updates are buffered in the DRAM copy of a page before persistently propagating the updates in-place to the NVM working copy without any kind of logging. After a successful update, PMThreads atomically switches the roles between the *working copy* and the *consistent copy* in NVM.

Running on a dual-socket system with 20 cores, the experimental evaluation results have showed that PMThreads substantially outperforms the state-of-the-art persistent runtimes targeting lock-based programs. We have evaluated PMThreads with 14 multi-threaded applications from PARSEC [3] and Phoenix [39] as well as Memcached. In addition, we have used two stress case scenarios.

The results have showed that PMThreads on average outperforms NVthreads and Atlas by 3.1× and 2.6× on the Parsec and Phoenix benchmarks, respectively. For the stress test, PMThreads is at least 4.7× faster than NVthreads, Mnemosyne and Atlas. For Memcached, PMThreads incurs within 2× average slowdown to the baseline program running with standard `pthreads`; a theoretical best case.

Acknowledgments

We thank the reviewers and our shepherd Michael Bond for the feedback and help. The research at NUDT is supported by the Tianhe Supercomputer Project 2018YFB0204301, National High-level Personnel for Defense Technology Program (2017-JCJQ-ZQ-013), NSF 61902405. The research at the University of Manchester is supported by the EU H2020 ACTiCLOUD 732366, and EPSRC LAMBDA EP/N035127/1 projects. Mikel Luján is funded by an Arm/RAEng Research Chair Award and a Royal Society Wolfson Fellowship.

References

[1] Intel Corporation. 2019. Intel Optane DC Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html. Accessed: 2019-11-11.

[2] Amro Awad, Sergey Blagodurov, and Yan Solihin. 2016. Write-Aware Management of NVM-based Memory Extensions. In *Proceedings of the 2016 International Conference on Supercomputing* (Istanbul, Turkey) *(ICS '16)*. ACM, New York, NY, USA, Article 9, 12 pages. https://doi.org/10.1145/2925426.2926284

[3] Christian Bienia and Kai Li. 2009. PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*.

[4] D. Castro, P. Romano, and J. Barreto. 2018. Hardware Transactional Memory Meets Memory Persistency. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 368–377. https://doi.org/10.1109/IPDPS.2018.00046

[5] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. ACM, New York, NY, USA, 433–452. https://doi.org/10.1145/2660193.2660224

[6] E. Chen, D. Lottis, A. Driskill-Smith, D. Druist, V. Nikitin, S. Watts, X. Tang, and D. Apalkov. 2010. Non-volatile spin-transfer torque RAM (STT-RAM). In *68th Device Research Conference*. 249–252. https://doi.org/10.1109/DRC.2010.5551975

[7] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS XVI)*. ACM, New York, NY, USA, 105–118. https://doi.org/10.1145/1950365.1950380

[8] Nachshon Cohen, David T. Aksun, Hillel Avni, and James R. Larus. 2019. Fine-Grain Checkpointing with In-Cache-Line Logging. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. ACM, New York, NY, USA, 441–454. https://doi.org/10.1145/3297858.3304046

[9] Nachshon Cohen, David T. Aksun, and James R. Larus. 2018. Object-oriented Recovery for Non-volatile Memory. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 153 (Oct. 2018), 22 pages. https://doi.org/10.1145/3276523

[10] Nachshon Cohen, Michal Friedman, and James R. Larus. 2017. Efficient Logging in Non-volatile Memory by Exploiting Coherency Protocols. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 67 (Oct. 2017), 24 pages. https://doi.org/10.1145/3133891

[11] Nachshon Cohen, Rachid Guerraoui, and Igor Zablotchi. 2018. The Inherent Cost of Remembering Consistently. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures* (Vienna, Austria) *(SPAA '18)*. ACM, New York, NY, USA, 259–269. https://doi.org/10.1145/3210377.3210400

[12] Ellis Giles, Kshitij Doshi, and Peter Varman. 2017. Continuous Checkpointing of HTM Transactions in NVM. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management* (Barcelona, Spain) *(ISMM 2017)*. ACM, New York, NY, USA, 70–81. https://doi.org/10.1145/3092255.3092270

[13] Ellis R Giles, Kshitij Doshi, and Peter Varman. 2015. SoftWrAP: A lightweight framework for transactional support of storage class memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 1–14.

[14] Vincent Gramoli. 2015. More Than You Ever Wanted to Know About Synchronization: Synchrobench, Measuring the Impact of the Synchronization on Concurrent Algorithms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Francisco, CA, USA) *(PPoPP 2015)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/2688500.2688501

[15] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. Pisces: A Scalable and Efficient Persistent Transactional Memory. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 913–928. https://www.usenix.org/conference/atc19/presentation/gu

[16] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. 2017. NVthreads: Practical Persistence for Multi-threaded Applications. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. ACM, New York, NY, USA, 468–482. https://doi.org/10.1145/3064176.3064204

[17] Qingda Hu, Jinglei Ren, Anirudh Badam, Jiwu Shu, and Thomas Moscibroda. 2017. Log-Structured Non-Volatile Main Memory. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 703–717. https://www.usenix.org/conference/atc17/technical-sessions/presentation/hu

[18] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*. USENIX Association, Oakland, CA, 187–200. https://www.usenix.org/conference/fast18/presentation/hwang

[19] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. ACM, New York, NY, USA, 427–442. https://doi.org/10.1145/2872362.2872410

[20] Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. 2016. Linearizability of persistent memory objects under a full-system-crash failure model. In *International Symposium on Distributed Computing*. Springer, 313–327.

[21] Kaan Genç, Michael Bond, Harry Xu. 2020. Crafty: Efficient, HTM-Compatible Persistent Transactions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2020*. ACM. https://doi.org/10.1145/3385412.3385991

[22] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. 2016. High-Performance Transactions for Persistent Memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. ACM, New York, NY, USA, 399–411. https://doi.org/10.1145/2872362.2872381

[23] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 462–477. https://doi.org/10.1145/3341301.3359635

[24] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. ACM, New York, NY, USA, 329–343. https://doi.org/10.1145/3037697.3037714

[25] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung. 2018. iDO: Compiler-Directed Failure Atomicity for Nonvolatile Memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 258–270. https://doi.org/10.1109/MICRO.2018.00029

[26] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2011. Dthreads: Efficient Deterministic Multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 327–336. https://doi.org/10.1145/2043556.2043587

[27] Kai Lu, Wenzhe Zhang, Xiaoping Wang, Mikel Luján, and Andy Nisbet. 2017. Flexible Page-level Memory Access Monitoring Based on Virtualization Hardware. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Xi'an, China) *(VEE '17)*. ACM, New York, NY, USA, 201–213. https://doi.org/10.1145/3050748.3050751

[28] Y. Lu, J. Shu, and L. Sun. 2015. Blurred persistence in transactional persistent memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. 1–13. https://doi.org/10.1109/MSST.2015.7208274

[29] Y. Lu, J. Shu, L. Sun, and O. Mutlu. 2014. Loose-Ordering Consistency for persistent memory. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*. 216–223. https://doi.org/10.1109/ICCD.2014.6974684

[30] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. 2017. Atomic In-place Updates for Non-volatile Main Memories with Kamino-Tx. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. ACM, New York, NY, USA, 499–512. https://doi.org/10.1145/3064176.3064215

[31] A. Memaripour and S. Swanson. 2018. Breeze: User-Level Access to Non-Volatile Main Memories for Legacy Software. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*. 413–422. https://doi.org/10.1109/ICCD.2018.00069

[32] memcached.org. 2019. Memcached – a distributed memory object caching system. https://memcached.org/. Accessed: 2019-08-10.

[33] Micron Technology. 2019. Breakthrough Nonvolatile Memory Technology. https://www.micron.com/products/advanced-solutions/3d-xpoint-technology. Accessed: 2019-01-28.

[34] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B Morrey III, Dhruva R Chakrabarti, and Michael L Scott. 2017. Dalí: A periodically persistent hash map. In *31st International Symposium on Distributed Computing (DISC 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[35] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Minneapolis, Minnesota, USA) *(ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 265–276. http://dl.acm.org/citation.cfm?id=2665671.2665712

[36] Azalea Raad and Viktor Vafeiadis. 2018. Persistence Semantics for Weak Memory: Integrating Epoch Persistency with the TSO Memory Model. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 137 (Oct. 2018), 27 pages. https://doi.org/10.1145/3276507

[37] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. 2019. Persistency Semantics of the Intel-X86 Architecture. *Proc. ACM Program. Lang.* 4, POPL, Article 11 (Dec. 2019), 31 pages. https://doi.org/10.1145/3371079

[38] Azalea Raad, John Wickerson, and Viktor Vafeiadis. 2019. Weak Persistency Semantics from the Ground up: Formalising the Persistency Semantics of ARMv8 and Transactional Models. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 135 (Oct. 2019), 27 pages. https://doi.org/10.1145/3360561

[39] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. 2007. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. 13–24. https://doi.org/10.1109/HPCA.2007.346181

[40] Redis Labs. 2019. Memtier Benchmark. https://github.com/RedisLabs/memtier_benchmark. Accessed: 2019-08-10.

[41] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutiu. 2015. ThyNVM: Enabling software-transparent crash consistency in persistent memory systems. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 672–685. https://doi.org/10.1145/2830772.2830802

[42] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS XVI)*. ACM, New York, NY, USA, 91–104. https://doi.org/10.1145/1950365.1950379

[43] Tianzheng Wang and Ryan Johnson. 2014. Scalable Logging Through Emerging Non-volatile Memory. *Proc. VLDB Endow.* 7, 10 (June 2014), 865–876. https://doi.org/10.14778/2732951.2732960

[44] H-S Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E Goodson. 2010. Phase change memory. *Proc. IEEE* 98, 12 (2010), 2201–2227.

[45] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 323–338. https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu

[46] P. Zardoshti, T. Zhou, Y. Liu, and M. Spear. 2019. Optimizing Persistent Memory Transactions. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 219–231. https://doi.org/10.1109/PACT.2019.00025

[47] Wenzhe Zhang, Kai Lu, Mikel Luján, Xiaoping Wang, and Xu Zhou. 2015. Write-combined Logging: An Optimized Logging for Consistency in NVRAM. *Sci. Program.* 2015, Article 25 (Jan. 2015), 1 pages. https://doi.org/10.1155/2015/398369