Persistency for Synchronization-Free Regions

Vaibhav Gogte
University of Michigan, USA
vgogte@umich.edu

Stephan Diestelhorst
Arm Research, UK
stephan.diestelhorst@arm.com

William Wang
Arm Research, UK
william.wang@arm.com

Satish Narayanasamy
University of Michigan, USA
nsatish@umich.edu

Peter M. Chen
University of Michigan, USA
pmchen@umich.edu

Thomas F. Wenisch
University of Michigan, USA
twenisch@umich.edu

Abstract

Nascent persistent memory (PM) technologies promise the performance of DRAM with the durability of disk, but how best to integrate them into programming systems remains an open question. Recent work extends language memory models with a persistency model prescribing semantics for updates to PM. These semantics enable programmers to design data structures in PM that are accessed like memory and yet are recoverable upon crash or failure. Alas, we find the semantics and performance of existing approaches unsatisfying. Existing approaches require high-overhead mechanisms, are restricted to certain synchronization constructs, provide incomplete semantics, and/or may recover to state that cannot arise in fault-free execution.

We propose persistency semantics that guarantee failure atomicity of synchronization-free regions (SFRs) —program regions delimited by synchronization operations. Our approach provides clear semantics for the PM state recovery code may observe and extends C++11's "sequential consistency for data-race-free" guarantee to post-failure recovery code. We investigate two designs for failure-atomic SFRs that vary in performance and the degree to which commit of persistent state may lag execution. We demonstrate both approaches in LLVM v3.6.0 and compare to a state-of-the-art baseline to show performance improvement up to 87.5% (65.5% avg).

CCS Concepts • **Computer systems organization → Architectures**; • **Software and its engineering → Software notations and tools**; • **Hardware → Memory and dense storage**;

Keywords Persistent memories, persistency models, language memory models, failure-atomicity, synchronization-free regions

ACM Reference Format:
Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. 2018. Persistency for Synchronization-Free Regions. In *Proceedings of 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3192366.3192367

PLDI'18, June 18–22, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5698-5/18/06…$15.00
https://doi.org/10.1145/3192366.3192367

1 Introduction

Emerging persistent memory (PM) technologies, such as Intel and Micron's 3D XPoint [24], aim to combine the byte-addressability of DRAM with the durability of storage. Unlike traditional storage devices, which provide only an OS-managed block-based interface, PM offers a load-store interface similar to DRAM. This interface enables fine-grained updates and avoids the hardware/software layers of conventional storage, lowering access latency.

Although PM devices are nascent, the best way to integrate them into our programming systems remains a matter of fierce debate [11, 15, 22, 28, 33, 51, 57]. The promise of PM is to enable data structures that provide the convenience and performance of in-place load-store manipulation, and yet persist across failures, such as power interruptions and OS or program crashes. Following such a crash, volatile program state (DRAM, program counters, registers, etc.) are lost, but PM state is preserved. A *recovery* process can then examine the PM state, reconstruct required volatile state, and resume program execution. The design of such recovery processes is well studied in specialized domains, such as databases and file systems [9, 19, 37, 45], but open questions remain for general programming systems.

Reasoning about the correctness of recovery code requires precise semantics for the allowable PM state after a failure [7, 10, 11, 13, 51, 57]. Specifying such semantics is complicated by the desire to support concurrent PM accesses from multiple threads and optimizations that reorder or coalesce accesses.

Recent work has proposed memory *persistency models* to provide programmers with such semantics [2, 13, 21, 28, 51]. Such models say that a PM access has *persisted* when the

effects of that access are guaranteed to be observed by recovery code in the event of a failure. Similar to memory consistency models, which govern the visibility of writes to shared memory, persistency models govern the order of persists to PM. Notably, many persistency models allow the *persist* of a PM access to lag its visibility, enabling overlap of long PM writes with subsequent execution. Both industry [2, 21] and academia [13, 14, 51] have proposed candidate persistency models, but most of these have been specified at the abstraction level of the hardware instruction set architecture (ISA). Such ISA-level persistency models do not specify semantics for higher-level languages, where compiler optimizations may also reorder or elide PM reads and writes.

Language-level persistency [30] proposes extending the memory models of high-level languages, like C++11 and Java, with persistency semantics. In this paper, we argue that the language-level semantics proposed to date, *Acquire-Release Persistency* (ARP) for C++11, are deeply unsatisfying, as they fail to extend the "sequential consistency for data-race-free programs (SC for DRF)" guarantee enjoyed in fault-free execution to recovery code [5]. ARP specifies semantics that prescribe ordering constraints at the granularity of individual accesses. Although ARP bounds the latest point (with respect to other memory accesses) at which a PM store may persist, it does not generally preclude PM stores from persisting *early*, ahead of preceding accesses in memory (visibility) order. As such, the set of states a recovery program might observe includes many states that (1) do not correspond to SC program executions, and (2) could never arise in a fault-free execution, posing a daunting challenge for recovery design.

Reasoning about recovery can be greatly simplified by providing *failure atomicity* of sets of PM updates. Failure atomicity assures that either all or none of the updates in a set are visible after failure, reducing the state space recovery code might observe. Atomicity (beyond the PM access granularity) can be achieved via logging [7, 11, 29, 57], shadow buffering [13], or checkpointing [52] mechanisms, which can be implemented in hardware [29, 52], as part of the programming/runtime system [7], or within the application [11, 13, 57].

ATLAS [7] argues to simplify recovery design by guaranteeing *failure-atomicity of outer-most critical sections*. Under such semantics, the language/runtime guarantees that recovery will observe a PM state as it existed when no locks were held by an application. However, we argue that this approach suffers from three key deficiencies: (1) its semantics are unclear for PM updates outside critical sections, (2) it does not generalize to other synchronization constructs (e.g., condition variables), (3) it requires expensive cycle detection among critical sections on different threads to identify sets that must be jointly failure-atomic, which leads to high overhead.

Instead, we propose persistency semantics that provide precise failure-atomicity at the granularity of *synchronization free regions* (SFRs)—thread regions delimited by synchronization operations or system calls. Prior works have used the SFR abstraction to define language memory models [39, 41] and to identify and debug data-races [4, 16, 39]. Under failure-atomic SFRs, the state observed by recovery will always conform to the program state at a *frontier* of past synchronization operations on each thread.

We argue that failure-atomic SFRs strike a compelling balance between programmability and performance. In a well-formed program, SFRs must be data-race free. This property allows us to extend the SC-for-DRF guarantee to recovery code and avoid the unclear semantics of ARP. Moreover, our approach avoids the limitations of ATLAS-like approaches.

We implement failure-atomic SFRs in a C++11 implementation (built on LLVM [34] v3.6.0). A programmer annotates variables that should be allocated in a persistent address space. Our compiler pass and runtime system introduce undo-logging that enables recovery to PM state of a prior SFR frontier, from which application-specific recovery can then reconstruct volatile program state. We consider two designs that strike different trade-offs in simplicity vs. performance.

SFR-atomicity with coupled visibility: In this design, the persistent state lags the frontier of execution by at most a single (incomplete) SFR; recovery rolls back to the start of the SFR upon failure. This approach admits simple logging, but exposes the latency of PM flushing and commit.

SFR-atomicity with decoupled visibility: In this design, we allow execution to run ahead of the persistent state. We defer flushing and commit to background threads using a garbage-collection-like mechanism. In this design, we propose efficient mechanisms to ensure that the SFR commit order matches their execution order.

In summary, we make following contributions:

- We make a case for failure atomicity at SFR granularity and show how this approach provides precise PM semantics and is applicable to arbitrary synchronization primitives, such as C++11 atomics.
- We demonstrate how SFR-atomicity with coupled visibility simplifies logging, resulting in an average performance improvement of 63.2% over the state-of-the-art ATLAS design [7].
- We further observe that ordering of logs is sufficient for recoverability and propose SFR-atomicity with decoupled visibility. With this design, we show a further performance improvement of 65.5% over ATLAS.

2 Background and Motivation

We give brief background on memory persistency.

2.1 Memory Persistency Models

Today's systems implement several hardware structures that reorder, coalesce and buffer updates before writing them to memory. Such reordering complicates using PMs for recovery because the correctness of recovery mechanisms rely on the order in which updates are persisted to the PM [10, 11, 13, 51, 57]. Several persistency models have been proposed, both in industry [2, 21] and in academia [13, 14, 51], to guarantee persist order in modern systems. Intel has recently extended the x86 ISA with an explicit `CLWB` instruction that can flush dirty cache lines to the memory controller. Note that Intel requires platforms to ensure that any buffering in the memory controller serving PM is persistent, so completion of a `CLWB` instruction is enough to guarantee an update is persistent (without the now-deprecated `PCOMMIT` instruction, see [23]). A subsequent `SFENCE` instruction orders the `CLWB` with respect to ensuing stores. Thus, a store followed by a `CLWB-SFENCE` sequence ensures that the store has persisted before subsequent memory operations are globally visible. Note, however, that PM stores may (unexpectedly) persist well before the `CLWB` if they are replaced from the cache hierarchy, and failure atomicity is assured only at the granularity of individual persist operations. Logging mechanisms must be built if larger failure-atomicity granularity is desired [30] and recovery code must explicitly account for the possibility that PM stores are replaced from the cache well before they are explicitly flushed.

2.2 Failure Atomicity

Logging mechanisms such as shadowing [13] and write-ahead-logging (WAL) [11, 20, 22, 29, 57] provide failure atomicity for a group of persists. In shadowing, updates are made to a shadow copy of the original data. The shadow copy is then committed by atomically switching a pointer in a metadata structure (e.g., page table). WAL provides failure atomicity by either logging the updates in *redo* or *undo logs*. In redo-logging [11, 20, 57], updates are first recorded in persistent logs and then applied in-place in the original location. Thus, a store implies (at least) two PM writes, one to log the update and one to mutate the original location. In case of failure, the recovery process inspects the redo-logs and reapplies the updates. In contrast, undo-logs record the old value of a location before it is written. On failure, the recovery process rolls back partial PM updates from undo logs. Redo-logging requires isolation [11] or redirection [20, 57] of subsequent loads to the log area, which typically incurs high overhead (in fault-free execution). In contrast, undo-logs allow in-place updates to data structures, so subsequent loads can read these locations directly.

2.3 Logging Mechanisms

Prior hardware and software mechanisms use logging to provide failure atomicity, but most work has focused on transaction-based programs [11, 22, 32, 38, 57]. Hardware mechanisms create undo logs [29] or redo logs [14] transparently for transaction-based code, thus enabling failure atomicity for the transaction, but require complex hardware structures for log management. Software solutions, such as Mnemosyne [57] and NV-Heaps [11], implement libraries that enable failure atomicity for transaction-based programs. However, in addition to semantic differences, there are significant challenges in porting existing lock-based programs to a transactional execution model [6, 7]. We seek to look beyond transaction-based programs and define durability semantics for the more general synchronization primitives offered by modern programming languages. To this end, we define persistency semantics for the synchronization operations in the C++11 memory model.

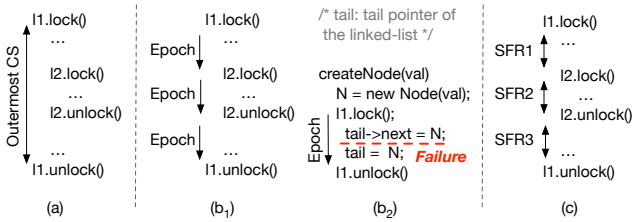

Figure 1. (a) Failure atomicity for outermost critical sections in ATLAS, (b_1) Epoch ordering in ARP, (b_2) Unclear failure atomicity semantics in ARP resulting in partial updates in PM, and (c) Our proposal: failure atomicity for SFRs.

3 Persistency Semantics for Languages

We first discuss existing proposals that add persistency semantics to the language memory model. In particular, ATLAS [7] and acquire-release persistency (ARP) [30] extend the C++ memory model with persistency semantics. The two proposals differ in the granularity of failure atomicity they guarantee and rely on different synchronization primitives to ensure correct persist ordering in PM. We discuss each proposal.

ATLAS: ATLAS provides persistency semantics for lock-based multi-threaded C++ programs. It guarantees failure atomicity at the granularity of an outermost critical section, as shown in Figure 1(a), where a critical section is the code bounded by *lock* and *unlock* synchronization primitives. The failure atomicity of outer-most critical sections ensures that recovery observes PM state as it existed when no locks were held in the program. This guarantee precludes recovery from observing state that is partially updated by an interrupted critical section. Failure-atomicity of critical sections is appealing from a programmability perspective, as it guarantees recovery may only observe sequentially consistent PM state.

However, ATLAS's persistency semantics have significant shortcomings. ATLAS fails to provide any durability semantics for synchronization operations other than mutexes. It

does not support widely used synchronization primitives, such as condition variables, and does not offer any semantics for lock-free programs. Moreover, it does not provide clear semantics for persistent updates outside of critical sections. Such updates may be partially visible after failure. In addition, ATLAS requires recording the total order of lock acquires and releases during execution and a complex cycle-detection mechanism to ensure that mutually-dependent critical sections are made failure atomic together. As we will show, the performance overhead of the required logging and cycle-detection mechanisms are high.

ARP: ARP specifies persistency semantics that provide failure atomicity of individual stores. ARP ascribes persists to ordered *epochs* using intra- and inter-thread ordering constraints prescribed via synchronization operations. As shown in Figure 1(b_1), ARP may re-order persists within epochs but disallows reordering across epochs. However, ARP constrains only the latest point at which a PM write may become durable—ARP allows for volatile write-back caches that reorder PM writes. A PM write may become durable as soon as it is globally visible. As such, a potentially unbounded set of writes may be reordered and visible even though preceding writes (in program order) are lost upon failure.

Figure 1(b_2) shows example code to append a new node to a persistent linked-list. Under fault-free execution in ARP, this code first acquires an exclusive lock on the linked-list, updates the `Next` pointer of the tail to the newly created node, and then the tail pointer is updated to the new node. As ARP does not constrain the durability of the two updates before the completion of the epoch, the update to `tail` may become durable earlier than the update to `tail->next`. In case of a failure, an incomplete update to the tail pointer will result in an inconsistent linked-list. The two updates within the critical section must be failure-atomic to ensure consistency of the linked-list. Additional logging mechanisms are required to provide failure atomicity at larger granularity.

We find ARP semantics unsatisfying. Although it may be possible to construct logging mechanisms that can tolerate writes that become persistent far earlier than expected (e.g., well before preceding store and release operations), reasoning in such a framework is difficult—a logging mechanism might have to resort to checksums or other complex, probabilistic mechanisms to detect partial log records. Importantly, a programmer must reason about non-serializable states when writing recovery code.

We argue instead for persistency semantics that provide failure atomicity for SFRs. Our approach can support arbitrary C++ synchronization operations with clear semantics and simple runtime mechanisms, avoiding the performance overheads of ATLAS.

3.1 C++ Memory Model

The C++ memory model provides synchronization operations, namely atomic loads, stores, and read-modify-writes, to order shared memory accesses. These accesses may directly manipulate synchronization variables, enabling implementation of a wide variety of synchronization primitives. In this paper, we refer to accesses to atomic variables that have load semantics as *acquire* operations, and those with store semantics as *release* operations. The C++ memory model prescribes a *happens-before* ordering relation between release and acquire operation, to enable programmers to order shared memory accesses (formalized later in Section 6.1). The happens-before relation orders the visibility of data accesses in (volatile) memory. However, C++ currently provides no durability semantics for accesses to PM. We extend the semantics of synchronization operations to also prescribe the order in which PM updates become durable.

4 Design Overview

We extend the C++ memory model with durability semantics for multi-threaded programs. We leverage synchronization operations to establish SFR boundaries and assure failure atomicity at this granularity, as shown in Figure 1(c). An SFR is a region of code delimited by two synchronization operations (or system calls) [39, 50]. If a synchronization operation has store semantics and modifies a location in PM, it forms its own, single-instruction region ordered before a second SFR it delimits comprising subsequent writes until the next synchronization. C++ requires that SFRs be data-race free, and, in turn, guarantees serializability of SFRs, despite any compiler and hardware optimizations that reorder accesses within SFRs to gain performance [1, 5]. That is, programs are guaranteed to behave as if the updates made within SFRs become visible to all other threads atomically at the synchronization operation that terminates the SFR. Note that C++ provides no semantics for programs with unannotated data-races.

The key advantage of providing failure atomicity at SFR granularity is that it allows us to extend the appearance of SC-for-DRF behavior to recovery code as well as fault-free execution. In the absence of SFR atomicity, loads that observe PM state after failure in effect race with the PM updates that may or may not have completed within the SFR running at the point of failure. As such, C++ places no constraints on the state recovery may observe. Under failure-atomic SFRs, the state in PM at recovery follows the program state at a frontier of past synchronization operations on each thread.

C++ provides synchronization operations that assure SC-for-DRF. Specifically, we study the inter-thread and intra-thread *happens-before* ordering prescribed by synchronization operations in multi-threaded applications to order memory accesses. We extend these guarantees to ensure that the memory accesses within SFRs become persistent in an order

consistent with the constraints on when they may become visible. We formalize these requirements later in Section 6.2.

Further, we propose compile and runtime mechanisms to provide failure atomicity at SFR granularity. We implement a compiler pass in LLVM v3.6.0, which instruments synchronization operations and PM accesses with undo-logging operations. In a traditional undo logging scheme, the state of the memory locations to be updated is first recorded in undo logs. Once the logs persist, in-place mutations of data structures may be made. Once the mutations are complete, state is committed by invalidating and discarding corresponding log entries. We investigate two logging designs that vary in simplicity and performance.

SFR-atomicity with coupled visibility: In this design, the visibility of the program state in volatile caches is coupled with its persistent state in PM. The in-place PM mutations are flushed at the end of each SFR and the undo log is immediately committed. Thus, the committed state lags the frontier of execution by at most a single (currently executing) SFR; recovery rolls back to the start of the SFR, minimizing the state lost on a failure. This approach admits a simple logging design where there is only a single uncommitted SFR per thread and logs are entirely thread-local. However, it exposes flush and commit latency on the critical execution path.

SFR-atomicity with decoupled visibility: Instead, we can allow execution to run ahead of the persistent state by deferring flush and commit. In this approach, the persistent state still comprises a frontier of SFRs on each thread, but may arbitrarily lag execution. We use a garbage-collection-like mechanism to periodically flush PM state and commit logs. This approach can hide the latency of flushing and commit with execution of additional SFRs. The key challenge is that the SFR commit order must match their execution order. We describe efficient mechanisms to ensure correct commit.

5 SFR Failure Atomicity

We next describe the logging mechanism we propose to provide failure-atomicity for SFRs.

5.1 Logging

In both variants of our system, we use undo logging to provide failure atomicity of SFRs. For the synchronization operation that begins an SFR, and every PM store operation within the SFR, our compiler pass emits code to construct an undo log entry in PM. Figure 2 illustrates the high-level steps our scheme must perform. Undo logs are appended to thread-local log buffers in PM. The log entry records the values PM locations had at the start of the SFR, before any mutation. The log entry is then persisted by explicitly flushing it from volatile caches to the PM (step L). Next, our compiler pass emits an ISA-level memory ordering barrier (to order the flush with subsequent writes) and the store operation that

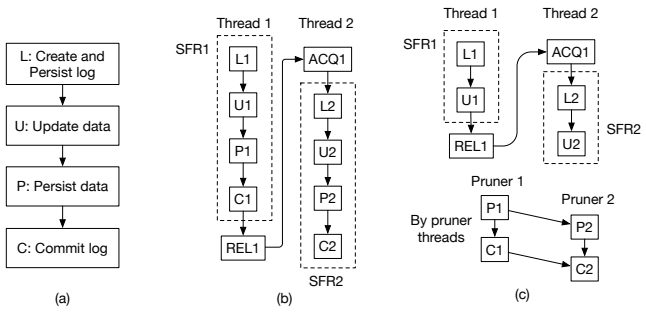

Figure 2. (a) Steps in undo logging mechanism, (b) Undo log ordering in Coupled-SFR when SFRs are durability-ordered, and (c) Undo log ordering in Decoupled-SFR when SFRs are durability-ordered.

updates the persistent data structure in place (step U). This update may remain buffered in volatile write-back caches or it may drain to PM due to cache replacement, unless we explicitly flush it. Once updates have been explicitly flushed and persisted (step P), the corresponding undo log entries may be committed (step C). The commit operation marks logs to be pruned, discarded and reused. Our two atomicity schemes differ in when and how they perform these latter two steps.

As shown in Figure 2(a), the partial updates within an SFR are recoverable only when the steps outlined above are performed in order [32]. For instance, undo logs must be created and persist before in-place mutations may be made. Otherwise, it is possible that the mutations are written-back from caches to PM before the undo log persists. If failure occurs in the interim, the state as of the start of the SFR cannot be recovered. Similarly, undo logs may be committed only after the in-place mutations persist. We ensure proper ordering between the operations by using mechanisms of an underlying ISA-level persistency model. In the case of Intel x86, this requires `CLWB` (or `CLFLUSHOPT` or `CLFLUSH` in older processors) to flush writes and `SFENCE` to order with respect to subsequent operations.

In case of a failure, recovery code begins by inspecting the uncommitted undo logs. It processes these logs, rolling back updates that may have drained from uncommitted SFRs. After rollback, the PM state will correspond to the state that existed at the start of some *frontier* of SFRs on each thread. Subsequent recovery operations (e.g., to prepare volatile data structures) are assured they will not observe updates from any partially executed SFR.

Log structure: We adopt an undo log organization similar to ATLAS [7]. Each thread manages a thread-local header, located in a pre-specified location in PM, which points to a linked list of undo log entries. As the undo logs are thread-private, threads may concurrently append to their logs. The order of entries in each undo log reflects program order. Log entries include the following fields:

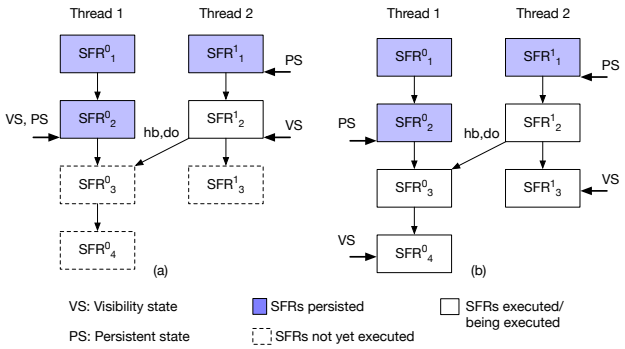

Figure 3. Persistent and execution state of SFRs in (a) Coupled-SFR, and (b) Decoupled-SFR.

- **Log type**: Entry type, one of `STORE`, `ACQUIRE`, or `RELEASE`
- **Addr**: Address of the access
- **Value**: Value to which to recover for `STORE` operations, or the log count (see Section 5.3) for `ACQUIRE`, or `RELEASE`
- **Size**: Access size
- **Next**: Link to next log entry

5.2 SFR-atomicity with Coupled Visibility

Our first design, SFR failure-atomicity with coupled visibility (Coupled-SFR), couples execution (more precisely, visibility of PM reads and writes) and persist of PM updates—persists may lag execution only until the start of the next SFR. Execution and persistency advance nearly in lock-step.

Logging: Under Coupled-SFR, updates within an SFR are flushed and persist at the end of the SFR. Our compiler pass emits code to create undo logs, mutate data in place, flush mutations, and commit logs as described in Section 5.1. We emit log creation code for each PM store as shown in Figure 4(a). Before the SFR's terminal synchronization, an `SFENCE` instruction is emitted to ensure that all PM mutations persist before any writes in the next SFR.

Failure and Recovery: Each thread maintains only undo logs for its incomplete SFR. Upon failure, recovery code rolls back updates from the partially completed SFR on each thread using the logs. Subsequent recovery code observes the PM state as it was at the last synchronization operation prior to failure on each thread.

Discussion: The central advantage of Coupled-SFR is that each thread must track only log entries for stores within its still-incomplete SFR, and does not interact with any other thread. The thread-private nature of our commit stands in stark contrast to ATLAS, which must perform a dependency analysis and cycle-detection across all threads' logs to identify log entries that must commit atomically. Because accesses within SFRs must be data-race free, there can be no dependences between accesses in uncommitted SFRs; all inter-thread dependencies must be ordered by the synchronization commencing the SFR, and hence may depend only on committed state. The PM state after recovery is easy to interpret, as it conforms to the state at the latest synchronization on each thread.

However, the downside of Coupled-SFR is that there is relatively little scope to overlap the draining of persistent writes with volatile execution—execution stalls at the end of the SFR until all PM writes are flushed and the log is committed, potentially exposing much of PM persist latency on the critical path. Figure 3(a) illustrates an example of how high persist latencies can delay execution. In Figure 3(a), the program state on Thread 2 is stalled while the updates in SFR_2^1 remain pending to persist. These stalls further delay execution on Thread 1, as SFR_3^0 is ordered after SFR_2^1 by synchronization operations.

5.3 SFR-atomicity with Decoupled Visibility

The key drawback of Coupled-SFR is that it exposes the high latency of persists and log commits on the execution critical path. Instead, we decouple the visibility of updates (as governed by cache coherence and the C++ memory model) from the frontier of persistent state; that is, we can allow persistent state to lag execution—an approach we call Decoupled-SFR. Nevertheless, Decoupled-SFR must still assure that recovery will roll PM state back to some prior state that conforms to a frontier of synchronization operations on each thread. To ensure that persistent state does not fall too far behind (which risks losing forward progress in the event of failure), we periodically invoke a flush-and-commit mechanism, much like garbage collection in managed languages. This mechanism flushes in-place updates and commits logs. However, the key invariant this mechanism must maintain is that SFRs commit in an order consistent with their execution. We next describe how we ensure this property.

Logging: Program state is recoverable if undo logs persist in the order the SFRs are executed (more precisely, the partial order in which they became visible, according to the C++11 memory model). In case of failure, undo logs are processed in reverse order to recover program state to the start of committed SFRs. The key departure of Decoupled-SFR from Coupled-SFR is that we defer flush and commit to perform them in the background, off the critical execution path.

In Figure 2(c), we illustrate logging under Decoupled-SFR. Like Coupled-SFR, our compiler pass emits logging code in advance of in-place PM mutations. In addition, we emit log entries for all synchronization operations. Read synchronization operations create `ACQUIRE` log entries, while write and read-modify-write emit `RELEASE` entries. If a `RELEASE` is to a location in PM, we emit first a `STORE` and then a `RELEASE` log for it. Log entries are appended to thread-local logs in creation (program) order. Pseudo-code for the instrumentation of store, acquire and release operations are shown in

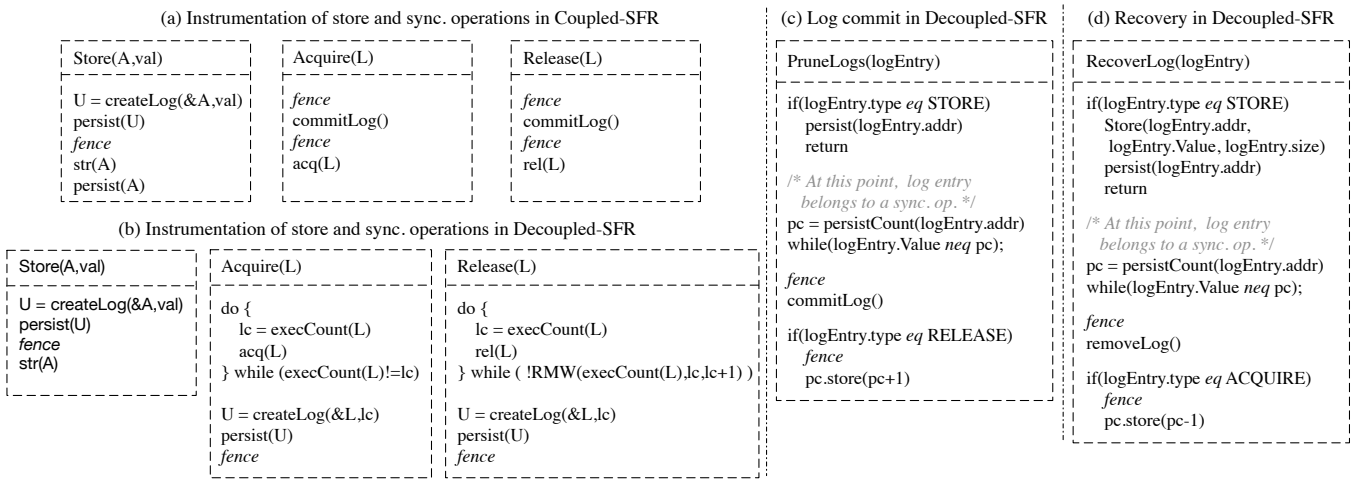

Figure 4. (a) Instrumentation for store and synchronization operations in Coupled-SFR design. (b) Instrumentation for store and synchronization operations in Decoupled-SFR design. Acquire (acq) and release (rel) operations are atomic loads and atomic stores to the synchronization variable L. (c) Pseudo-code for log commit operation by pruner threads in Decoupled-SFR design. (d) Pseudo-code for recovery operation at failure in Decoupled-SFR design.

Figure 4(b). Unlike Coupled-SFR, we do not emit flush or commit code as part of the SFR. Instead, we delegate these operations to *pruner threads*, which operate periodically on the logs. We next explain how we maintain correct commit order for SFRs.

Ordering commit: Each program thread has an accompanying *pruner thread* that flushes mutations and commits the log on its behalf. Like garbage-collection, pruner threads are invoked periodically to commit and recycle log space.

Recoverability requires that logs are pruned—committing the updates in the corresponding SFR—in the same order as the SFRs are executed, else the state after recovery will not correspond to a state consistent with fault-free execution. As such, our logging mechanism must log the *happens-before* ordering relations between SFRs (as governed by the C++11 memory model) and commit according to this order. We record happens-before by: (1) adding acquire / release annotations to the per-thread logs, (2) maintaining per-thread logs in program order (thereby capturing intra-thread ordering), and (3) tracking order across threads by maintaining a monotonic sequence number across release / acquire pairs. We refer throughout to Figure 4(b), which illustrates pseudo-code for our instrumentation.

We associate a sequence number `execCount(L)` with each synchronization variable `L`. We use a lock-free hashmap to record `execCount(L)` for each synchronization operation, allowing lock-free concurrent access/update of the counters. The hashmap is located in volatile memory for faster accesses, because we do not need the hashmap for recovery and can reinitialize it post-failure. For simplicity, our implementation assumes `execCount(L)` is large enough that we can ignore wrap-around.

For operations with release semantics (see Figure 4(b) Release), the instrumentation code observes the current value of `execCount(L)`. Then, `execCount(L)` is incremented with an atomic memory access. The loop in this pseudo-code accounts for the possibility of racing `RELEASE` operations. A log entry is emitted reflecting the identity of the synchronization variable `L` and the observed value of `execCount(L)`, which is recorded in the log entry's `Value` field.

A subsequent `ACQUIRE` operation that synchronizes-with a `RELEASE` observes the sequence number of that release (see Figure 4(b) Acquire). Note that it is critical that the acquire operation and the observation of the sequence number are atomic, which we arrange by reading the `execCount(L)` field twice, before and after the acquire—a mismatch indicates two racing release operations (unlikely in well-structured code), which we handle by synchronizing again.

Log commit: The pruner threads must together commit logs in sequence number order. We use a second monotonic counter per synchronization variable, the *persist counter* (`persistCount(L)`), also placed in a lock-free hashmap, to synchronize and order SFR commit across pruner threads.

The pseudo-code for log commit and pruning is depicted in Figure 4(c). Each pruner thread processes its thread-private log starting at the entry indicated in its corresponding log header. Upon reaching an entry for a synchronization operation, the pruner thread may need to wait for other pruner threads to ensure commit is properly ordered. We consider each kind of log entry in turn:

`STORE`: The pruner thread ensures the corresponding mutation is persistent by flushing the corresponding address with a `CLWB` operation (using the `Addr` field recorded in the log).

`ACQUIRE`: The pruner thread spins on `persistCount(L)` until it is equal to the `execCount(L)` recorded in the `Value` field of the log entry. This spin awaits commit of the SFR with which the acquire synchronized. The SFR is then committed.

`RELEASE`: The pruner thread spins on `persistCount(L)` until it is equal to the `execCount(L)` recorded in the `Value` field of the log entry. This spin waits for commit of the preceding release of the same synchronization variable. Then, a *fence* is issued to ensure the `CLWB` operations of any preceding `STORE` log entries are ordered before commit. The SFR may then be committed. After commit, `persistCount(L)` is incremented, which unblocks the pruner thread that will commit the next SFR for this synchronization variable. Note, again, the need for a memory fence after commit to ensure that the commit operation is ordered before subsequent commits and the increment of `persistCount(L)`.

Log pruning: Pruner threads prune (discard) log entries when an SFR is committed. To prune a group of entries belonging to an SFR, the pruner atomically modifies the pointer in its log header to point to a later log entry. The log space may then be freed/recycled. As the log entries belonging to an SFR are committed atomically, only after the updates within the SFR have persisted, the pruner threads guarantee SFR failure-atomicity.

Failure and Recovery: In Decoupled-SFR, the state after failure and recovery must conform to a frontier of past synchronization operations on each thread. Recovery code inspects the uncommitted undo logs and rolls back updates in the reverse order of log creation. Much like the commit operation of pruner threads, the recovery code uses the `execCount(L)` sequence number recorded in the `Value` field of log entries to apply undo logs in reverse order. The pseudo-code for this recovery is shown in Figure 4(d). First, the recovery process scans all undo logs and records the highest observed sequence number for each synchronization variable in a hashmap. Then, `STORE` log entries are replayed in reverse creation order to roll back values in PM. As the logs roll back, replayed log entries are pruned when traversing `ACQUIRE` or `RELEASE` entries, thus allowing recovery even in the event of multiple/nested failures. Once PM state is recovered, application-specific recovery code takes over to reconstruct any necessary volatile state.

Optimizations: We enable certain optimizations to make log pruning more efficient. First, we can often commit batches of SFRs atomically. If `persistCount(L)` matches the `Value` in all synchronization log entries for consecutive SFRs (i.e., no need to wait), we commit them together. Second, a pruner thread processes `STORE` log entries for a single SFR together: it issues multiple `CLWB` operations to flush updates in parallel. Importantly, processing entries as a group allows us to coalesce multiple updates to the same address within an SFR. Note that we still log all writes to the same memory addresses within the SFR separately, which avoids the need to check if the memory address has previously been logged within the SFR on the critical execution path.

Finally, if a pruner thread commits its last log entry, it blocks to conserve CPU. Execution threads wake all pruners when log entries accumulate above some threshold. Note that, since pruner threads may have to wait for one another to process dependent log entries, they should be gang-scheduled.

Discussion: Under Decoupled-SFR, persistent state may arbitrarily lag execution state. Hence, although recovery arrives at a state consistent with a synchronization frontier, forward progress may be lost. Programmers must be aware of this possibility. If state loss is not desired (e.g., if the program will perform an operation with an irrecoverable side-effect), Decoupled-SFR provides a *psync* operation, which stalls execution and triggers pruner threads to drain their logs.

6 Durability Invariants

We briefly discuss invariants that a logging implementation must meet to ensure failure-atomicity of SFRs and describe how the Coupled-SFR and Decoupled-SFR implementations ensure these invariants.

6.1 Preliminaries

We introduce a notation to describe persist ordering, following the approach in prior works [30, 31], and present a summary of persist ordering as it relates to the C++ memory model. C++ provides atomic (`std::atomic<>`) primitives, which allow programmers explicit control over the ordering of memory accesses. Atomic variables may be loaded and stored directly (without, e.g., a separate mutex) and hence facilitate the implementation of a wide variety of synchronization primitives. We formalize persist ordering using the following notation for memory operations to a location l from a thread i.

- ACQ_l^i: an atomic load or read-modify-write
- REL_l^i: an atomic store or read-modify-write
- M_x^i: a non-atomic operation on memory location `x`

We indicate ordering constraints among memory events with the following notation:

- $M_x^i \leq_{sb} M_y^i$: M_x^i is sequenced-before M_y^i in thread i
- $REL_l^i \leq_{sw} ACQ_l^j$: A release operation on location l in thread i "synchronizes with" an acquire operation on location l in thread j.
- $M_x^i \leq_{hb} M_y^j$: M_x^i in thread i happens-before M_y^j in thread j

The C++ memory model achieves inter-thread ordering using the "synchronizes-with" ordering relation and intra-thread ordering using the "sequenced-before" ordering relation. The "happens-before" relation is the transitive closure

of "synchronizes-with" $\leq_{sw}$ and "sequenced-before" $\leq_{sb}$ orderings.

Memory operations must follow the *sequenced-before* ordering relations within a thread. A release operation REL_l^i orders prior memory access M_x^i and an acquire operation ACQ_l^i orders subsequent memory access M_y^i on thread `i`. Further, the C++ memory model achieves the inter-thread ordering using the "synchronizes-with" order relation between an acquire and release operation. A release operation REL_l^i in thread i synchronizes-with the acquire operation ACQ_l^j in thread j. The synchronizes-with relation orders memory access M_x^i in thread i with memory access M_y^j in thread j:

$$(M_x^i \leq_{sb} REL_l^i \leq_{sw} ACQ_l^j \leq_{sb} M_y^j) \rightarrow M_x^i \leq_{hb} M_y^j \quad (1)$$

We now use the happens-before ordering relation between the memory accesses to define the order in which SFRs must be made durable in PM.

6.2 SFR Durability

Atomic loads, stores, and read-modify-write operations delimit SFRs. We say that a store operation is *visible post-recovery* if the effects of the store may be observed by code that runs after failure and recovery. Our logging designs must ensure that an SFR is failure-atomic:

***Atomicity Invariant**: If there exists a PM update within an SFR that is visible post-recovery, then all updates in the SFR must be visible post recovery.*

The *Atomicity Invariant* guarantees that the updates within an SFR are not partially visible after failure. We say that an SFR is *durable* if all its updates are visible post-recovery.

Further, our logging must ensure that SFRs become durable in an order consistent with the C++11 memory model. We use the happens-before ordering relation between the memory accesses to prescribe the order SFRs must be made durable.

Suppose SFR^i and SFR^j denote SFRs on threads i and j respectively. Consider memory operations M_x^i and M_y^j on threads i and j respectively, such that $M_x^i \in SFR^i$, and $M_y^j \in SFR^j$. We say that SFR^i is *durability-ordered* before SFR^j if:

$$\exists \left(M_x^i \in SFR^i, M_y^j \in SFR^j \right) | M_x^i \leq_{hb} M_y^j, SFR^i \leq_{do} SFR^j \quad (2)$$

where $SFR^i \leq_{do} SFR^j \rightarrow SFR^i$ must be made durable before SFR^j.

Finally, we require that durability-order between SFRs is transitive and irreflexive:

$$(SFR^i \leq_{do} SFR^j) \wedge (SFR^j \leq_{do} SFR^k) \rightarrow SFR^i \leq_{do} SFR^k \quad (3)$$

Following Equation 2, logging must satisfy:

***Durability Invariant**: If an SFR is durable, SFRs that are durability-ordered before it must also be durable.*

Note that the SFRs are unordered if there exists no transitive durability-ordering relation between them. The key correctness requirement of the recovery mechanism is that the state that the recovery code observes after failure must be consistent with the ordering constraints expressed in Equation 2-3. We now describe how our designs, Coupled-SFR and Decoupled-SFR, satisfy the atomicity invariant to guarantee SFR failure-atomicity and the durability invariant to ensure SFR durability is properly ordered.

6.3 Coupled-SFR

Under Coupled-SFR, each thread maintains a thread-local pointer to a list of log entries for at most one incomplete SFR. The log is committed atomically using `commitLog` as shown in Figure 4(a) before the synchronization operation that ends the SFR is executed. `CommitLog` atomically prunes the entire list of undo log entries by zeroing the pointer in the thread-local header. The SFR is durable when the logs commit. This atomic commit satisfies the *Atomicity Invariant*, thereby ensuring failure-atomicity of SFRs.

Figure 2(b) illustrates the SFRs, `SFR1` and `SFR2`, as ordered by the happens-before ordering relation. Note that execution of the memory accesses in `SFR1` are ordered before those in `SFR2` by the happens-before ordering relation between `REL1` on thread 1 and `ACQ1` on thread 2. The ordering relation between `REL1` and `ACQ1`, implies `SFR1` is durability-ordered before `SFR2` by Equation 2. As shown in Figure 2(b), `SFR1` becomes durable in the commit stage (step `C1` in Figure 2(b)) before the release operation. Further, the subsequent acquire operation is sequenced-before the commit operation (step `C2` in Figure 2b) in `SFR2`. The two ordering relations guarantee that `SFR1` becomes durable before `SFR2` in Coupled-SFR.

6.4 Decoupled-SFR

Similar to Coupled-SFR, under Decoupled-SFR, each thread maintains a thread-local pointer to the head of its undo logs. The pruner threads commit logs atomically by adjusting the log header to point to a subsequent log entry for a synchronization operation, as shown in Figure 4(c). The atomic commit ensures that one (or more) SFRs are made durable atomically.

Figure 2(c) shows the order of creation of undo logs for `SFR1`, which is *durability-ordered* with `SFR2`. The durability-order relation implies that `SFR1` must be made durable before `SFR2`. During execution, Decoupled-SFR assigns ascending sequence numbers to the synchronization operations. The log entry corresponding to the release operation records a sequence number from `execCount(L)`, atomically increments it and then performs the release operation. Consequently, the acquire operation that synchronizes-with the release operation records the updated sequence number in its log entry followed by executing `SFR2`. As shown in Figure 4(c), the pruner threads commit the log entry in ascending sequence

Table 1. Benchmarks.

Benchmark	Description
Concurrent queue (CQ)	Insert/Delete nodes in a queue
Array Swap (SPS)	Random swap of array elements
Persistent Cache (PC)	Update entries in persistent hash table
RB-tree	Insert/Delete nodes in RB-Tree
TATP	Update location trans. from TATP [49]
Linked-List (LL)	Update/Insert/Delete nodes in a linked-list
TPCC	New Order trans. from TPCC [56]

number order. Thus, the logs for `SFR1`, which are sequenced-before the release operation, commit before `SFR2`, which are sequenced-after the acquire operation. The two ordering relations guarantee the durability of `SFR1` before `SFR2`.

7 Evaluation

We implement a compiler pass that can emit code for both our logging approaches in LLVM [34] v3.6.0. The compiler pass instruments stores and synchronization operations to create undo logs according to the pseudo-code in Figure 4. We also provide a library containing the recovery code that rolls back undo logs upon failure, recovering to a frontier of past synchronization operation, and the runtime code for log pruning in Decoupled-SFR. We first describe our experimental framework including our system configuration, the benchmark suite that we use, and the designs we consider in our experiments.

System configuration: We perform our experiments on an Intel E5-2683 v3 server class machine with 14 physical cores, each with 2-way hyper-threading, operating at 2.00GHz. Since byte-addressable persistent memory devices are not yet commercially available, we use Linux `tmpfs` [55], memory-mapped in DRAM, to mimic the persistent address space of a PM-enabled system. Note that it is widely expected that the access latency of actual PM devices will be higher than that of DRAM (likely by 2-10x) [60]. In our experimental setup, we expect to underestimate the cost of flushing mutations to PM in ATLAS and Coupled-SFR. In Decoupled-SFR, because we delegate flush operations for in-place updates to the pruner threads, we expect to hide the flush latency. Hence, we expect to obtain similar performance for Decoupled-SFR even with slower PM devices. As such, we believe our evaluation is conservative in estimating the performance advantage of Decoupled-SFR over the alternatives.

Our Haswell-class server machine does not offer `clwb` instructions, instead providing a `clflush` operation to flush the data out of the cache hierarchy to the memory controller. Systems supporting `clwb`, which avoids some undesirable overheads of `clflush`, are expected to be available in the near future. To our knowledge, no available x86 platform provides mechanisms to ensure that data are indeed flushed to memory. Instead, Intel presently requires the memory controller in PM-enabled systems to guarantee durability (e.g., via battery backup or flush-upon-failure) [23]. As a result, we rely on `sfence` operation to order the drain of updates.

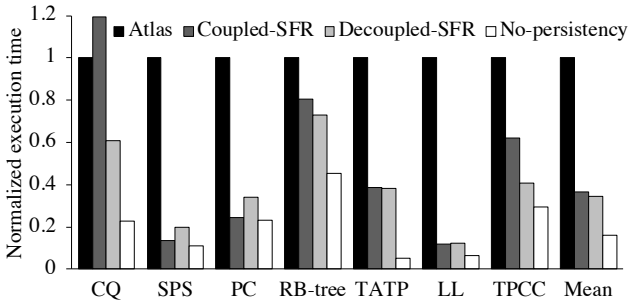

Figure 5. Execution time of Coupled-SFR and Decoupled-SFR designs normalized to ATLAS. No-persistency design, with no durability guarantees, shows an upper bound on performance.

Benchmarks: We study a suite of seven write-intensive multi-threaded micro-benchmarks and benchmarks, listed in Table 1, which have been used in prior studies of persistent memory systems [11, 29, 30, 51]. The Concurrent Queue (CQ), similar to that of prior works [30, 51], inserts and removes nodes from a shared persistent queue. The Array Swap, RB-tree and Persistent cache (PC) are similar to the implementations in NV-Heaps [11]. Our TATP benchmark executes the update location transaction of the TATP database workload [49], which models the home location registration database of a telecommunications provider. Our TPCC benchmark executes the new-order transaction from the TPCC database workload [56], which models an order processing system. The Linked-List benchmark uses a hand-over-hand locking mechanism to update, insert, and remove nodes in a persistent linked-list. All the benchmark run 12 concurrent execution threads and perform 10M operations on the persistent data structure.

Design options: We compare the following designs: (a) ATLAS: a state-of-the-art logging approach that provides failure-atomicity of outermost critical sections, (b) Coupled-SFR: our mechanism for SFR failure-atomicity with coupled visibility, (c) Decoupled-SFR: our mechanism for SFR failure-atomicity with decoupled visibility, and (d) No-persistency: a design that provides no recoverability of the program. We include No-persistency to show an upper bound for our performance improvements and quantify the cost of recoverability. No-persistency provides no recovery guarantees.

7.1 Performance Comparison

Figure 5 contrasts the execution time of Coupled-SFR and Decoupled-SFR with that of ATLAS. In this experiment, we perform two operations per SFR for the concurrent queue (CQ), persistent cache (PC), array swap (SPS), RB-tree, and

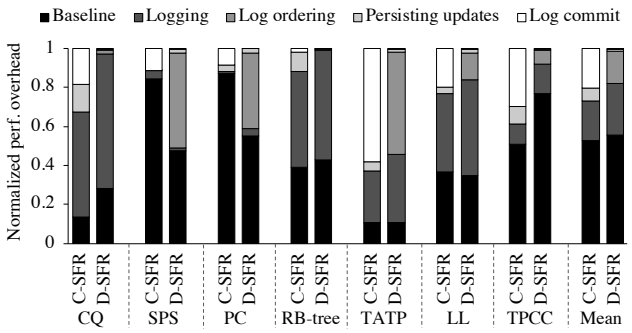

Figure 6. Distribution of logging overhead in Coupled-SFR and Decoupled-SFR designs.

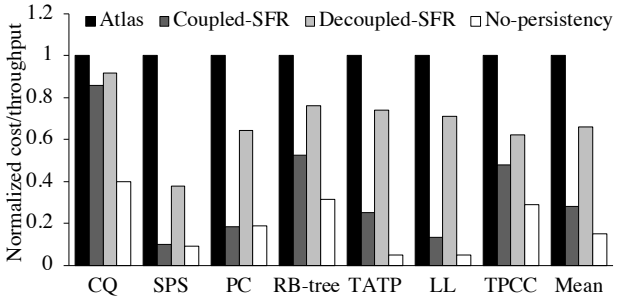

Figure 7. Cost per throughput of Coupled-SFR and Decoupled-SFR normalized to ATLAS. The No-persistency design shows cost/throughput for a non-recoverable implementation.

linked-list (LL). The other two benchmarks TATP, and TPCC implement open specifications and so each SFR includes as many write operations as are required to implement the mandated behavior of update location and new order transactions, respectively. ATLAS performs the slowest in all benchmarks (except in CQ) because it records the order of execution of critical sections (as opposed to Coupled-SFR), and flushes the PM mutations within each critical section on the critical execution path (as opposed to Decoupled-SFR). Decoupled-SFR enables light-weight recording of SFR order and performs flush and commit operations on pruner threads, off the critical execution path. As a result, Decoupled-SFR achieves up to 80.1% and 66.0% performance improvement in array swap and persistent cache, respectively, which employ fine-grained locking and have the highest concurrency. Linked-list uses hand-over-hand locking and must acquire several locks in the linked-list before operating on a node. Decoupled-SFR performs best with 87.5% improvement in Linked-list, as it greatly simplifies logging as compared to the ATLAS.

It is interesting to note that Coupled-SFR performs better than Decoupled-SFR in array swap, persistent cache, and linked-list. This might seem counter-intuitive, as Coupled-SFR admits simpler logging at the cost of committing logs at every synchronization operation. However, these benchmarks perform only two stores per SFR. As a result, the cache flush operations on the critical path under Coupled-SFR incur less overhead than the more complex logging code of Decoupled-SFR.

As the number of stores per critical section grows, ATLAS fails to scale. ATLAS does not support concurrent commit and must rely on only a single helper thread to commit and recover log entries. Therefore, as the number of PM writes scales with the number of execution threads, the single helper thread can no longer keep up with the required commit rate and the log grows until available log capacity is exhausted. On the contrary, both Coupled-SFR and Decoupled-SFR perform distributed pruning and do not suffer from this issue.

CQ has no concurrency as all the threads contend to acquire a single lock to access the queue. Coupled-SFR performs worse than ATLAS in CQ as the flush and commit operations are done in the critical execution path by each thread, incurring delay. We show a separate comparison between Coupled-SFR and Decoupled-SFR with a varying number of PM writes per SFR in Section 7.4.

7.2 Logging Overhead

We study the overhead of each of the various steps performed in logging for our Coupled-SFR and Decoupled-SFR designs. In Figure 6, we incrementally enable steps in undo logging and study the distribution of execution time in each step. Note that none of these incomplete designs implement a recoverable system; we study them only to quantify overheads.

In Coupled-SFR, the majority of time is spent in creating the logs entries and flushing them to PM. Note that there is no overhead in Coupled-SFR due to log ordering as the log entries are committed at the end of each SFR. Overall, Coupled-SFR spends 39% of the execution time in flush and log commit when there are two operations per SFR.

In contrast, Decoupled-SFR spends less than 1% of execution time flushing updates and committing logs as these operations are performed by pruner threads in the background. The remaining 1% overhead is due to the pruning of the final few logs when the benchmarks complete. Our result indicates that the pruner threads are able to keep up with program execution. We also measure the log size overhead in the Decoupled-SFR design. Across our experiments, the log size in Decoupled-SFR is typically less than a few KB and never grows above 100 KB. On average, log creation costs 26.6% and recording of log order costs 16.3% of the total execution time in Decoupled-SFR.

7.3 CPU Cost per Throughput

We next evaluate the cost of the background activity required by both ATLAS and Decoupled-SFR to commit their logs. Although the pruner/helper threads do not delay execution on the critical path, they nonetheless consume CPU resources

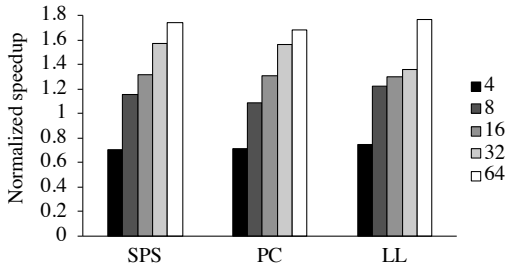

Figure 8. Sensitivity study showing speedup of Decoupled-SFR normalized to Coupled-SFR with increasing number of stores per SFR.

and therefore can increase the total CPU cost to complete the benchmarks. We measure this overhead by dividing the total CPU utilization (in CPU-seconds) consumed by all threads over the course of benchmark execution by the achieved throughput (operations/transactions per second). For this metric, lower is better (less CPU overhead per unit of forward progress). Figure 7 shows the normalized CPU-cost per throughput of each benchmark for all four designs. We find that the cost of Coupled-SFR is the lowest as compared to ATLAS and Decoupled-SFR, as the threads executing the program commit the logs themselves. As we create as many pruner threads as there are execution threads in the program, Decoupled-SFR requires higher CPU resources to flush and commit the logs. In concurrent queue, which (despite its name) has no concurrency, the cost per throughput of Decoupled-SFR is equivalent to ATLAS, because there exists a single total order across all logs on all threads, and so the actions of the pruner threads are serialized. As No-persistency does not create any logs, it has the lowest cost per throughput of all designs, and illustrates the cost of recoverability. Overall, Coupled-SFR and Decoupled-SFR have 72.1% lower and 33.2% lower CPU-cost per throughput than ATLAS.

7.4 Sensitivity Study of Operations/SFR

The size of logs varies with the number of store operations performed in SFRs. We perform a sensitivity analysis to study how the performance of our designs compare as the number of stores per SFR increases. Figure 8 illustrates the performance of Decoupled-SFR for the four benchmarks, normalized to Coupled-SFR. With two stores per SFR, we see that Coupled-SFR performs better than Decoupled-SFR. Decoupled SFR is slower because the overhead of creating and updating the `execCount(L)` and `persistCount(L)` to maintain undo log order in Decoupled-SFR is higher than the performance gain of delegating flush operations for only two stores to pruner threads. As the number of store operations increase, the flush operations and log commits delay execution in Coupled-SFR. As a result, at 64 stores per SFR, Decoupled-SFR performs 1.74x faster than Coupled-SFR. For benchmarks such as CQ, RB-tree and TPCC, we have already shown in Figure 5 that Decoupled-SFR performs 1.98×, 1.53× and 1.10× better than Coupled-SFR, respectively.

8 Related Work

We briefly address related works that propose hardware and software solutions for PM systems.

Library-based solutions: NV-Heaps [11] and Mnemosyne [57] provide library-based application-level interfaces for building persistent objects in PM. Both provide libraries to create virtually mapped regions in persistent memory, along with primitives to update persistent data mapped to the memory. They use write-ahead logging to provide failure atomicity for transactions. SoftWrAP [18] and REWIND [8] provide software libraries to perform transactional updates to PM. SoftWrAP uses alias tables to redirect the updates within the failure-atomic transactions to a log space in DRAM and commits the updates when the transactions retire. Similar to SoftWrAP, DUDETM [35] updates the redo logs for transactions in DRAM, and then persists and merges the logs in PM. Kamino-Tx [42] avoids logging by replicating the heap, performing updates within transactions on a working copy of the heap, and copying changes to the backup heap when transactions commit. Transactions simplify logging, both in hardware and software. However, our approach differs from software-annotated transaction-based solutions in that it is applicable to general-purpose programs that are not transaction-based, especially those that use synchronization mechanisms like conditional waits or complex locks that do not readily compose with transactional models. In this work, we seek to provide persistency semantics for arbitrary (non-transactional) synchronization.

Runtime logging solutions: NVthreads [20] extends ATLAS [7] to provide durability guarantees to lock-based programs. NVthreads uses copy-on-write to make updates within a critical section and then merges the updates to the live data at a 4KB page granularity at the end of outermost critical sections. Due to the expensive merge operations at the end of the critical sections, NVthreads suffers a high performance overhead in applications with frequent lock acquisition and release operations like the benchmarks that we study in this paper. Moreover, we extend durability semantics to more general synchronization constructs that NVthreads and ATLAS do not support. Boehm et al. [6] elaborates on the ATLAS programming model further and defines recovery semantics for updates to persistent locations both within and outside critical sections. ARP [30] and Izraelevitz et al. [26] propose language-level persistency models. Both works provide persist ordering, but fail to provide failure atomicity at a granularity larger than individual persists. Moreover, they offer unclear semantics at failure, as writes may be replaced from the cache hierarchy and persist well before other, earlier writes, exposing non-SC state to recovery. TARP [31]

and Izraelevitz et al. [27] offer x86 and ARM ISA encodings of language-level persistency models. Kolli et al. [32] introduces efficient implementation of transactions, namely synchronous-commit and deferred-commit transactions, that minimize persist dependencies by deferring commit of undo logs until the transactions conflict.

WSP [47] proposes mechanisms to flush the precise architectural state of a program at the moment of failure to PM. JUSTDO logging [25] recovers an application to its state right before the failure, and requires persisting of architectural state including stack-local variables before executing a critical section. It assumes the cache hierarchy is persistent to avoid high PM access latency when preserving volatile program state. Both WSP and JUSTDO logging fail to provide recoverability from failures other than power interruptions (*e.g.* kernel panic or application crash). SCMFS [59], BPFS [13], NOVA [61], NOVA-Fortis [62] and PMFS [15] propose filesystems that leverage low latency of PMs.

Checkpointing-based solutions: ThyNVM [52] proposes dual-scheme checkpointing mechanism to provide crash consistency support for DRAM+NVM systems. It eliminates stalls for checkpointing by overlapping execution and checkpointing. CC-HTM [17] leverages HTM to provide fine-grained checkpointing of transactions to PM. Survive [44] provides a fine-grained incremental checkpointing for hybrid DRAM+PM systems. Other works checkpoint the volatile state using cache persistence by ensuring that a battery backup is available to flush the volatile state to PM upon power failure [48], or by bypassing caches altogether [58].

Energy harvesting systems: A group of studies look at application consistency requirements for energy harvesting devices. As the energy supply for this class of devices is intermittent, these works explore mechanisms to maintain data consistency in PM while ensuring forward progress. Alpaca [40] provides a task-based programming model, where tasks present an abstraction for the atomicity of updates in PM. Alpaca requires programmers to annotate tasks and task-specific shared variables in the program. We provide a more generic mechanism built upon existing C++ synchronization. Idetic [43] and Hibernus [3] detect imminent power failure and periodically checkpoint volatile state, but may leave data in PM inconsistent [12]. This group of works propose consistency mechanisms for power failures alone, whereas we consider more general fail-stop failures.

Hardware-based solutions: Pelley et al. [32, 51] proposes persistency models closely aligned with the hardware memory consistency model to order writes to PM. His work proposes strict and relaxed persistency models that vary in the constraints imposed on the updates as they persist to PM. BPFS [13] uses epoch barriers to order persists in hardware. The persists within an epoch can be reordered while persist reordering across epochs is disallowed. DPO [33], Doshi et al. [14], HOPS [46], and Shin et al. [54] propose hardware mechanisms for efficiently implementing epoch persistency models. They implement hardware structures in the cache hierarchy that record and drain persists to the PM in order. ATOM [29] improves upon undo-logging mechanism for PM by decoupling the update of undo log from the in-place update to the persistent data-structure. It relies on hardware structures in the memory hierarchy that order logs before the actual updates to PM. Proteus [53] implements a software-assisted hardware solution to persist transactions atomically to PM. It involves significant modifications to the processor pipeline to record logs and order logs with respect to subsequent stores. Liu et al. [36] proposes an encryption mechanism based on counter-mode encryption. It employs hardware mechanisms to ensure atomicity of data and the associated counter used for its encryption in PM. Kiln [63] and LOC [38] provide a storage interface to PM to programmers, but rely on programmers to ensure isolation. Unlike hardware-based solutions, we use synchronization primitives in the C++ memory model to provide ordering and failure-atomicity to the PM updates.

9 Conclusion

Past works have proposed language-level persistency models prescribing semantics for updates to PM. However, we showed that the existing language-level persistency models either lack precise durability semantics or incur a high performance overhead. We made a case that failure-atomic SFRs strike a compelling balance between programmability and performance. We then examined two designs, Coupled-SFR and Decoupled-SFR, for failure-atomic SFRs that vary in performance and the amount by which the PM state may lag execution. We show that our designs simplify logging and outperform the state-of-the-art implementation by 87.5% (65.5% avg).

Acknowledgements

We would like to thank our shepherd, Tobias Wrigstad, and the anonymous reviewers for their valuable feedback. We would also like to thank Akshitha Sriraman for her feedback on this work. This work was supported by ARM and the National Science Foundation under the award NSF-CCF-1525372.

References

[1] Sarita V. Adve and Kourosh Gharachorloo. 1996. Shared Memory Consistency Models: A Tutorial. *Computer* 29, 12 (Dec. 1996), 66–76. https://doi.org/10.1109/2.546611

[2] ARM. 2016. ARMv8-A architecture evolution. https://community.arm.com/groups/processors/blog/2016/01/05/armv8-a-architecture-evolution.

[3] D. Balsamo, A. S. Weddell, A. Das, A. R. Arreola, D. Brunelli, B. M. Al-Hashimi, G. V. Merrett, and L. Benini. 2016. Hibernus++: A Self-Calibrating and Adaptive System for Transiently-Powered Embedded Devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 12 (2016), 1968–1980. https://doi.org/10.1109/TCAD.2016.2547919

[4] Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Brandon Lucia. 2015. Valor: Efficient, Software-only Region Conflict Exceptions. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 241–259. https://doi.org/10.1145/2814270.2814292

[5] Hans-J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ Concurrency Memory Model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 68–78. https://doi.org/10.1145/1375581.1375591

[6] Hans-J. Boehm and Dhruva R. Chakrabarti. 2016. Persistence Programming Models for Non-volatile Memory. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management (ISMM 2016)*. ACM, New York, NY, USA, 55–67. https://doi.org/10.1145/2926697.2926704

[7] 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 (OOPSLA '14)*. ACM, New York, NY, USA, 433–452. https://doi.org/10.1145/2660193.2660224

[8] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. 2015. REWIND: Recovery Write-Ahead System for In-Memory Non-Volatile Data-Structures. *PVLDB* 8, 5 (2015), 497–508. http://www.vldb.org/pvldb/vol8/p497-chatzistergiou.pdf

[9] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher M. Aycock, Gurushankar Rajamani, and David Lowell. 1996. The Rio File Cache: Surviving Operating System Crashes. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*.

[10] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2013. Optimistic Crash Consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 228–243. https://doi.org/10.1145/2517349.2522726

[11] 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 (ASPLOS XVI)*. ACM, New York, NY, USA, 105–118. https://doi.org/10.1145/1950365.1950380

[12] Alexei Colin and Brandon Lucia. 2016. Chain: Tasks and Channels for Reliable Intermittent Programs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 514–530. https://doi.org/10.1145/2983990.2983995

[13] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 133–146. https://doi.org/10.1145/1629575.1629589

[14] K. Doshi, E. Giles, and P. Varman. 2016. Atomic persistence for SCM with a non-intrusive backend controller. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 77–89. https://doi.org/10.1109/HPCA.2016.7446055

[15] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM, New York, NY, USA, Article 15, 15 pages. https://doi.org/10.1145/2592798.2592814

[16] Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-J. Boehm. 2012. IFRit: Interference-free Regions for Dynamic Data-race Detection. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*. ACM, New York, NY, USA, 467–484. https://doi.org/10.1145/2384616.2384650

[17] 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 (ISMM 2017)*. ACM, New York, NY, USA, 70–81. https://doi.org/10.1145/3092255.3092270

[18] E. R. Giles, K. Doshi, and P. Varman. 2015. SoftWrAP: A lightweight framework for transactional support of storage class memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. 1–14. https://doi.org/10.1109/MSST.2015.7208276

[19] Jim Gray and Andreas Reuter. 1993. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc.

[20] 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 (EuroSys '17)*. ACM, New York, NY, USA, 468–482. https://doi.org/10.1145/3064176.3064204

[21] Intel. 2014. Intel Architecture Instruction Set Extensions Programming Reference (319433-022). https://software.intel.com/sites/default/files/managed/0d/53/319433-022.pdf.

[22] Intel. 2015. Persistent Memory Programming. http://pmem.io/.

[23] Intel. 2016. Deprecating the PCOMMIT Instruction. https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction.

[24] Intel and Micron. 2015. Intel and Micron Produce Breakthrough Memory Technology. http://newsroom.intel.com/community/intel_newsroom/blog/2015/07/28/intel-and-micron-produce-breakthrough-memory-technology.

[25] 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 (ASPLOS '16)*. ACM, New York, NY, USA, 427–442. https://doi.org/10.1145/2872362.2872410

[26] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Distributed Computing: 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings*, Cyril Gavoille and David Ilcinkas (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 313–327. https://doi.org/10.1007/978-3-662-53426-7_23

[27] Joseph Izraelevitz and Michael L. Scott. 2014. Brief Announcement: A Generic Construction for Nonblocking Dual Containers. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing (PODC '14)*. ACM, New York, NY, USA, 53–55. https://doi.org/10.1145/2611462.2611510

[28] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2015. Efficient Persist Barriers for Multicores. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 660–671. https://doi.org/10.1145/2830772.2830805

[29] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra. 2017. ATOM: Atomic Durability in Non-volatile Memory through Hardware Logging. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 361–372. https://doi.org/10.1109/HPCA.2017.50

[30] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2017. Language-level Persistency. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 481–493. https://doi.org/10.1145/3079856.3080229

[31] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2017. TARP: Translating Acquire-Release Persistency. http://nvmw.eng.ucsd.edu/2017/assets/abstracts/1.

[32] 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 (ASPLOS '16)*. ACM, New York, NY, USA, 399–411. https://doi.org/10.1145/2872362.2872381

[33] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M. Chen, and Thomas F. Wenisch. 2016. Delegated persist ordering. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*. 1–13. https://doi.org/10.1109/MICRO.2016.7783761

[34] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

[35] 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 (ASPLOS '17)*. ACM, New York, NY, USA, 329–343. https://doi.org/10.1145/3037697.3037714

[36] S. Liu, A. Kolli, J. Ren, and S. Khan. 2018. Crash Consistency in Encrypted Non-volatile Main Memory Systems. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 310–323. https://doi.org/10.1109/HPCA.2018.00035

[37] David E. Lowell and Peter M. Chen. 1997. Free Transactions with Rio Vista. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP '97)*. ACM, New York, NY, USA, 92–101. https://doi.org/10.1145/268998.266665

[38] Youyou Lu, Jiwu Shu, Long Sun, and Onur Mutlu. 2014. Loose-Ordering Consistency for persistent memory. In *32nd IEEE International Conference on Computer Design, ICCD 2014, Seoul, South Korea, October 19-22, 2014*. 216–223. https://doi.org/10.1109/ICCD.2014.6974684

[39] Brandon Lucia, Luis Ceze, Karin Strauss, Shaz Qadeer, and Hans-J. Boehm. 2010. Conflict Exceptions: Simplifying Concurrent Language Semantics with Precise Hardware Exceptions for Data-races. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. ACM, New York, NY, USA, 210–221. https://doi.org/10.1145/1815961.1815987

[40] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: Intermittent Execution Without Checkpoints. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 96 (Oct. 2017), 30 pages. https://doi.org/10.1145/3133920

[41] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. 2010. DRFX: A Simple and Efficient Memory Model for Concurrent Programming Languages. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, New York, NY, USA, 351–362. https://doi.org/10.1145/1806596.1806636

[42] 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 (EuroSys '17)*. ACM, New York, NY, USA, 499–512. https://doi.org/10.1145/3064176.3064215

[43] Azalia Mirhoseini, Ebrahim M Songhori, and Farinaz Koushanfar. 2013. Idetic: A high-level synthesis approach for enabling long computations on transiently-powered ASICs. In *2013 IEEE International Conference on Pervasive Computing and Communications (PerCom)*. 216–224. https://doi.org/10.1109/PerCom.2013.6526735

[44] Amirhossein Mirhosseini, Aditya Agrawal, and Josep Torrellas. 2017. Survive: Pointer-Based In-DRAM Incremental Checkpointing for Low-Cost Data Persistence and Rollback-Recovery. *IEEE Computer Architecture Letters* 16, 2 (July 2017), 153–157. https://doi.org/10.1109/LCA.2016.2646340

[45] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *ACM Trans. Database Syst.* 17, 1 (March 1992), 94–162. https://doi.org/10.1145/128765.128770

[46] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. 2017. An Analysis of Persistent Memory Use with WHISPER. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 135–148. https://doi.org/10.1145/3037697.3037730

[47] Dushyanth Narayanan and Orion Hodson. 2012. Whole-system Persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, New York, NY, USA, 401–410. https://doi.org/10.1145/2150976.2151018

[48] Faisal Nawab, Dhruva Chakrabarti, Terence Kelly, and Charles B. Morey III. 2014. *Procrastination Beats Prevention: Timely Sufficient Persistence for Efficient Crash Resilience*. Technical Report HPL-2014-70. Hewlett-Packard.

[49] Simo Neuvonen, Antoni Wolski, Markku Manner, and Vilho Raatikka. 2011. Telecom Application Transaction Processing Benchmark. http://tatpbenchmark.sourceforge.net/.

[50] Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2013. . . . And Region Serializability for All. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Parallelism*.

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

[52] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. 2015. ThyNVM: Enabling Software-transparent Crash Consistency in Persistent Memory Systems. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 672–685. https://doi.org/10.1145/2830772.2830802

[53] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. 2017. Proteus: A Flexible and Fast Software Supported Hardware Logging Approach for NVM. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17)*. ACM, New York, NY, USA, 178–190. https://doi.org/10.1145/3123939.3124539

[54] Seunghee Shin, James Tuck, and Yan Solihin. 2017. Hiding the Long Latency of Persist Barriers Using Speculative Execution. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 175–186. https://doi.org/10.1145/3079856.3080240

[55] Peter Snyder. 1990. tmpfs: A virtual memory file system. In *In Proceedings of the Autumn 1990 European UNIX UsersâĂŹ Group Conference*. 241–248.

[56] Transaction Processing Performance Council (TPC). 2010. TPC Benchmark B. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5-11.pdf.

[57] 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 (ASPLOS XVI)*. ACM, New York, NY, USA, 91–104. https://doi.org/10.1145/1950365.1950379

[58] 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

[59] Xiaojian Wu and A. L. Narasimha Reddy. 2011. SCMFS: A File System for Storage Class Memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*. ACM, New York, NY, USA, Article 39, 11 pages. https://doi.org/10.1145/2063384.2063436

[60] C. Xu, D. Niu, N. Muralimanohar, R. Balasubramonian, T. Zhang, S. Yu, and Y. Xie. 2015. Overcoming the challenges of crossbar resistive memory architectures. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 476–488. https://doi.org/10.1109/HPCA.2015.7056056

[61] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST'16)*. USENIX Association, Berkeley, CA, USA, 323–338. http://dl.acm.org/citation.cfm?id=2930583.2930608

[62] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. 2017. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 478–496. https://doi.org/10.1145/3132747.3132761

[63] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. 2013. Kiln: Closing the Performance Gap Between Systems with and Without Persistence Support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 421–432. https://doi.org/10.1145/2540708.2540744