Strand Persistency

V. Gogte*, W. Wang†, S. Diestelhorst†, P. M. Chen*, S. Narayanasamy*, T. F. Wenisch*
*University of Michigan {vgogte,pmchen,nsatish,twenisch}@umich.edu
†ARM {william.wang,stephan.diestelhorst}@arm.com

I. Introduction

Persistent memory (PM) technologies, such as Intel and Micron's 3D XPoint, are here—OEMs are already evaluating engineering samples and volume shipments are expected in early 2019. PMs aim to combine byte-addressability of DRAM and durability of storage devices. Unlike traditional block-based storage devices, such as hard disks and SSDs, PMs can be accessed using a byte-addressable load-store interface, avoiding the expensive software layers required to access storage, and allowing for fine-grained PM manipulation.

As PMs are durable, they retain data across failures such as power interruptions and program crashes. Upon failure, the volatile program state in hardware caches, registers, and DRAM is lost. In contrast, PM retains its contents—a *recovery* process can inspect these contents, reconstruct required volatile state, and resume program execution.

Several *persistency models* have been proposed in the past to enable writing recoverable software, both in hardware [1], [2] and programming languages [3]–[7]. Like prior works, we refer to the act of completing a store operation to PM as a *persist*. Persistency models enable two key properties. First, they allow programmers to reason about the order in which persists are made. Similar to memory consistency models, which order visibility of shared memory writes, memory persistency models govern the order of persists to PM. Second, they enable failure-atomicity for a set of persists. In case of failure, either all or none of the updates within a failure-atomic set are visible to recovery.

Recent works [3], [4] extend the memory models of high-level languages, such as C++ and Java, with persistency semantics. Specifically, ATLAS [3], Coupled-SFR [4], and Decouped-SFR [4] employ synchronization primitives in C++ to prescribe the ordering and failure-atomicity of PM operations. ATLAS ensures failure-atomicity of PM updates within outermost critical-sections—program regions bounded by lock and unlock operations. Coupled- and Decoupled-SFR designs assure failure-atomicity for synchronization-free regions—program regions delimited by synchronization operations, such as acquire and release. These models implement undo logging to ensure failure-atomicity of PM updates. As shown in Figure 1b, an undo logging mechanism creates undo logs in PM that preserve old PM values before they are overwritten. On a failure, these undo logs are used to roll back partial PM updates. Note that undo logging exposes high PM write latency during execution as undo logs must be created and flushed to PM before in-place updates may be made.

These persistency models ensure that PM updates within a failure-atomic region become persistent in an order consistent with the constraints on when they may become visible, as shown in Figure 1c. For instance, Intel x86 systems employ `CLWB` (or `CLFLUSHOPT` in older systems) instruction to explicitly flush dirty cache lines to the memory controller and a subsequent `SFENCE` instruction to order ensuing stores with prior `CLWB`s and stores. Ideally, logging requires a pairwise ordering between logs and over-writing stores for correct recovery. In Figure 1c, `SFENCE` is required to order log creation and flush to PM with a subsequent store to a memory location `A`. Unfortunately, `SFENCE` enforces additional ordering constraints on persists that are not required for ensuring correct recovery. For instance, in Figure 1c, `SFENCE` additionally orders log creation and flush to L_A with log creation and flush to L_B. These ordering constraints limit persist concurrency.

We propose employing *strand persistency* [8] to minimally constrain orderings on persists to PM. Strand persistency decouples persist ordering from visibility of PM operations enforced by memory consistency models. It provides two key primitives—a *persist barrier* that explicitly enforces ordering constraints on persists, and a *strand barrier* that removes ordering constraints on subsequent persists. While memory consistency barriers enforce visibility order of memory operations, persist barriers enforce the order in which persists propagate to PM. In contrast, a strand barrier *removes* ordering constraints on subsequent persists. Persists after a strand barrier can propagate to PM concurrently with prior persists, ignoring preceding persist barriers. Note that the visibility of memory operations is still governed by the memory consistency model in the presence of a strand barrier.

Strand persistency requires hardware mechanisms to manage visibility and persist ordering separately. Moreover, it requires explicit annotations of programs with persist and strand barriers, making it error-prone and difficult to program. To this end, we implement persist barrier and strand barrier primitives in hardware and demonstrate how hardware can decouple visibility and persist ordering. We build logging design that employs persist and strand barriers to enforce only the minimal ordering constraints on persists required for correct recovery. We integrate our logging design with language-level persistency models that enable programmer-friendly persistency semantics using synchronization primitives in high-level programming languages. We show that our strand persistency mechanism relaxes persist ordering constraints and improves performance by up to 34.5%.

II. Design

Next, we explain our approach to relax persist ordering.

A. Strand Persistency

Strand persistency employs two primitives to prescribe persist ordering: a persist barrier to enforce persist ordering and a strand barrier to remove ordering constraints on subsequent

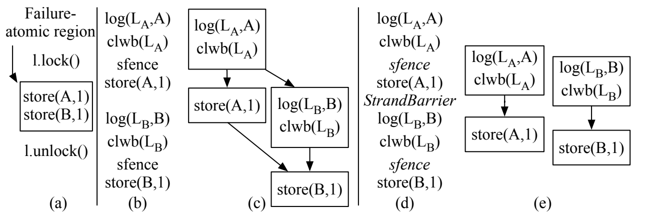

Fig. 1: (a) Example failure-atomic region, (b) Baseline logging, (c) Additional ordering constraints in baseline, (d) Logging under strand persistency, (e) Ordering constraints due to strand persistency.

persists. PM accesses on a thread separated by a persist barrier are ordered. The recovery may never observe persists that are separated by persist barriers on a thread out of order after failure. Conversely, a strand barrier removes ordering constraints on subsequent PM operations. A strand barrier initiates a new *strand*—a strand behaves as a separate logical thread in a persist order. Persists on different strands can be issued concurrently to PM. Note that persist barriers, within a strand, continue to order persists on that strand.

Strand persistency decouples the visibility and persist order of PM operations. The consistency model continues to order visibility of PM operations—PM operations on separate strands are visible in an order enforced by consistency model.

Inter-strand ordering: Orderings can arise between persists on different strands through *strong persist atomicity* [8]. Strong persist atomicity governs the ordering on memory operations to the same memory location. A subsequent PM operation on a new strand is ordered with the prior persist to the same memory location – persists to the same PM location are serialized even when they lie on different strands.

Hardware implementation: We implement memory dependencies enforced by persist and strand barriers in an out-of-order processor pipeline. A persist barrier must enforce the order in which updates drain to PM—it orders prior stores and `CLWB`s with subsequent stores and `CLWB`s in a program thread (or strand). In Intel x86, TSO memory consistency orders store retirement. Additionally, the `SFENCE` orders prior `CLWB`s with subsequent `CLWB`s. Thus, in Intel x86 systems, a persist barrier maps to an `SFENCE` instruction.

We implement a strand barrier instruction that removes prior memory ordering constraints and initiates a new strand. On encountering a strand barrier, our hardware implementation discards ordering constraints imposed by the prior in-flight persist barriers. Thus, any subsequent `CLWB`s can be issued concurrently with the `CLWB`s on previous strands provided they flush different PM locations. Our hardware implementation also performs dependency checks for incoming memory operations to ensure strong persist atomicity between strands.

B. Logging using strand barriers

We design an undo logging scheme that relies on strand and persist barriers for persist ordering. We create an undo log entry in PM to record the old value of a location that is being overwritten. The subsequent persist barrier ensures that the log is created and flushed to PM before the update is made. As the subsequent logs and stores can independently persist, we issue a strand barrier after a store operation to initiate a new strand, as shown in Figure 1(d-e).

We initialize and manage a per-thread log buffer in PM as an array of fixed-sized log entries. The *tail* pointer points to the location where the next log entry will be placed—we increment the tail upon log-entry creation. We maintain the tail pointer in volatile memory to ensure that subsequent log entries created on different strands are not ordered due to strong persist atomicity on the tail pointer. We maintain the `head` log pointer in PM. We update and flush the head pointer at the end of a failure-atomic region to atomically commit log entries. Upon a failure, the head pointer is used to initiate recovery. As persists to the logs on different strands are concurrent, the failure can expose log write reorderings. The recovery process scans and rolls back old values recorded in valid log entries from the end of the log space.

C. Language-level persistency models

We integrate our undo-logging mechanism with Coupled-SFR, Decoupled-SFR, and ATLAS designs, the state-of-the-art language persistency models. In each design, we maintain a thread-local log space and order undo log creation with the corresponding PM store within each strand. In Coupled-SFR, we flush all PM mutations and ensure that they persist before committing logs at the end of a failure-atomic region. In Decoupled-SFR and ATLAS, the persistent state lags execution. In both the designs, we record *happens-before* ordering relation in log entries on their creation.

III. Evaluation

We implement strand persistency in Gem5. We study a suite of five write-intensive micro-benchmarks and benchmarks used in prior works [4], [5]. Due to space limitations, we list only the average improvement obtained in our designs. Strand persistency enables high persist concurrency in all designs. We compare performance achieved due to our strand implementation as compared to the baseline Coupled-SFR, Decoupled-SFR, and ATLAS designs. Overall, we achieve performance improvement of up to 21.8% (14.3% avg.) in Coupled-SFR, 34.5% (21.4% avg.) in Decoupled-SFR, and 29.9% (18.2% avg.) in ATLAS with our strand implementation.

References

[1] Intel, "Instruction set extensions programming reference," 2014. https://software.intel.com/sites/default/files/managed/0d/53/319433-022.pdf.
[2] ARM, "Armv8-a architecture evolution," 2016. https://community.arm.com/groups/processors/blog/2016/01/05/armv8-a-architecture-evolution.
[3] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," in *OOPSLA '14*.
[4] V. Gogte, S. Diestelhorst, W. Wang, S. Narayanasamy, P. M. Chen, and T. F. Wenisch, "Persistency for synchronization-free regions," in *PLDI '18*.
[5] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Language-level persistency," ISCA '17.
[6] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Tarp: Translating acquire-release persistency," 2017. http://nvmw.eng.ucsd.edu/2017/assets/abstracts/1.
[7] V. Gogte, S. Diestelhorst, W. Wang, S. Narayanasamy, P. M. Chen, and T. F. Wenisch, "Failure-atomic synchronization-free regions," 2018. http://nvmw.ucsd.edu/nvmw18-program/unzip/current/nvmw2018-final42.pdf.
[8] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *ISCA '14*.