Hardware Implementation of 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—cloud vendors have already started public offerings with support for Intel's Optane DC persistent memory. PMs combine the byte-addressability of DRAM and durability of storage devices. Because 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, 5, 7, 6]. Like prior works [9], we refer to the act of completing a store operation to PM as a *persist*. Recent works [5, 3] extend the memory models of high-level languages, such as C++ and Java, with persistency semantics. These language-level persistency models differ in the synchronization primitives that they employ to provide varying granularity of failure atomicity. They enable failure atomicity for a set of persists. In case of failure, either all or none of the updates within a failure-atomic region are visible to recovery. Specifically, ATLAS [3], Coupled-SFR [5], and Decoupled-SFR [5] employ general synchronization primitives in C++ to prescribe the ordering and failure atomicity of PM operations. Other works [1] ensure failure atomicity at a granularity of transactions using software libraries or high-level language extensions.

These language-level models rely on low-level hardware ISA primitives to order PM operations [1, 2]. For instance, Intel x86 systems employ the `CLWB` instruction to explicitly flush dirty cache lines to the point of persistence and the `SFENCE` instruction to order subsequent `CLWB`s and stores with prior `CLWB`s and stores [1]. Under Intel's persistency model, `SFENCE` enforces a bi-directional ordering constraint on subsequent persists and introduces high-latency stalls until prior `CLWB`s and stores complete; `SFENCE` imposes stricter ordering constraints than required by language-level models.

Prior research proposals relax ordering constraints by proposing relaxed persistency models [9] in hardware and/or building hardware logging mechanisms to ensure failure-atomic updates to PM. These works propose relaxed persistency models, such as epoch persistency, that implement *persist barriers* to divide regions of code into *epochs*; they allow persist reordering within epochs and disallow persist reordering across epochs. Unfortunately, epoch persistency labels only consecutive persists that lie within the same epoch as concurrent. It fails to relax ordering constraints on persists that may be concurrent, but do not lie in the same epoch. In contrast, hardware logging mechanisms aim to provide efficient implementations for ensuring failure atomicity for PM updates in hardware. These mechanisms impose fine-grained ordering constraints (*e.g.*, between log and PM updates) on persists but propose inflexible hardware that fails to extend to a wide range of evolving language-level persistency models.

In this work, we propose StrandWeaver, which formally defines and implements the *strand persistency* model to minimally constrain ordering on persists to PM. The principles of the strand persistency model were proposed in earlier work [9, 4], but no hardware implementation, ISA primitives, or software use cases have yet been reported. The strand persistency model defines the order in which persists may drain to PM. It decouples persist order from the write visibility order (defined by the memory consistency model)—memory operations can be made visible in shared memory without stalling for prior persists to drain to PM.

II. Strand Persistency Model

Strand persistency divides thread execution into *strands*. Strands constitute sets of PM operations that lie on the same logical thread. Ordering primitives enforce persist ordering within strands, but persists are not individually ordered across strands. We use the term "strand" to evoke the idea that a strand is a part of a logical thread, but has independent persist ordering. 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 follow visibility order enforced by system's consistency model.

Strand primitives. Strand persistency employs three primitives to prescribe persist ordering: a *persist barrier* to enforce persist ordering among operations on a strand, *NewStrand* to initiate a new strand, and a *JoinStrand* to merge prior strands initiated on the logical thread. PM accesses on a thread separated by a persist barrier are ordered. Conversely, `NewStrand` removes ordering constraints on subsequent PM operations. `NewStrand` 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. The hardware must guarantee that recovery software never observes a mis-ordering of two PM writes on the same strand that are separated by a persist barrier. Finally, `JoinStrand` merges strands that were initiated on the logical thread. It ensures the persists issued on the prior strands complete before any subsequent persists are issued.

A. Hardware Implementation

We now describe StrandWeaver's hardware mechanisms.

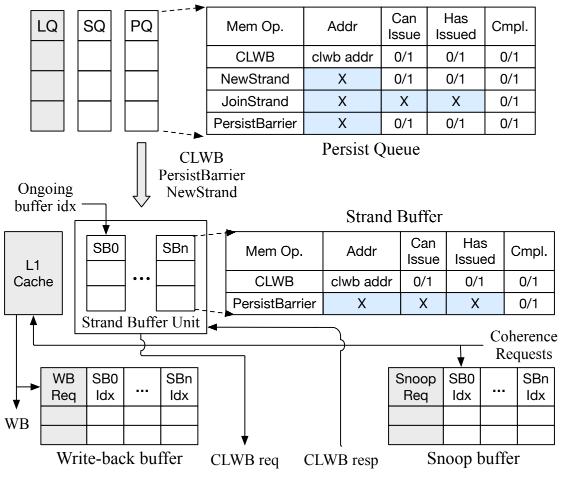

Fig. 1: StrandWeaver architecture.

Microarchitecure. We implement StrandWeaver's persist barrier, `NewStrand`, and `JoinStrand` primitives as ISA extensions. A persist occurs due to a voluntary data flush from volatile caches to PM using a `CLWB` operation, or a writeback resulting from cacheline replacement.

Architecture overview. Figure 1 shows the high-level architecture of StrandWeaver. The persist queue and strand buffer unit jointly enforce persist ordering. The persist queue, implemented alongside the load-store queue (LSQ), ensures that `CLWB`s and stores separated by a persist barrier within a strand are *issued* to the L1 cache in order, and `CLWB`s separated by `JoinStrand` *complete* in order. The strand buffer unit is primarily responsible for leveraging inter-strand persist concurrency to schedule `CLWB`s to PM. It resides adjacent to the L1 cache and comprises an array of strand buffers that may issue `CLWB`s from different strands concurrently. Each strand buffer manages persist order within a strand and guarantees that persists separated by persist barriers within that strand complete in order. The strand buffer unit also coordinates with the L1 cache to ensure that persists due to cache writebacks are ordered as per PMO. It also tracks cache coherence messages to ensure that inter-thread persist dependencies are preserved.

Persist queue architecture. The persist queue manages entries that record ongoing `CLWB`s, persist barriers, `NewStrand`, and `JoinStrand` operations. It tracks persist barriers to monitor intra-strand persist dependencies. On insertion, a persist barrier imposes a dependency so that `CLWB`s and stores are ordered within its strand. These constraints ensure that stores do not violate persist order by updating the cache and draining to PM via a cache writeback before preceding `CLWB`s. The persist queue also coordinates with the store queue to guarantee that younger `CLWB`s are issued to the strand buffer unit only after elder store operations to the same memory location. `JoinStrand` ensures that `CLWB`s and stores on prior strands complete before any subsequent `CLWB`s and stores are issued.

Strand buffer unit architecture. The strand buffer unit coordinates with the L1 cache to guarantee `CLWB`s and cache writebacks drain to PM and complete in order specified by the primitives. It maintains an array of strand buffers—each strand buffer manages persist ordering within one strand. `CLWB`s that lie in different strand buffers can be issued concurrently to PM. The strand buffer unit receives `CLWB`, persist barriers, and `NewStrand` operations from the persist queue. In the strand buffer unit, the ongoing buffer index points to the strand buffer to which an incoming `CLWB` or persist barrier is appended. This index is updated when the strand buffer unit receives a `NewStrand` operation indicating the beginning of a new strand.

Each strand buffer manages intra-strand persist order arising from persist barriers. On insertion in a strand buffer, a persist barrier creates a dependency that orders any subsequent `CLWB`s appended to the buffer. A persist barrier completes when `CLWB`s ahead of it complete and retire from the strand buffer.

Cache writebacks. PM writes can also happen due to cache line writebacks, in the same or the other cores. We provision fields in the writeback and snoop buffers that track these dependencies. We skip this discussion due to space limitation.

B. Designing Language-level Persistency Models

Recent efforts [3, 5, 1] extend persistency semantics and provide ISA-agnostic programming frameworks in high-level languages, such as C++ and Java. The models enable undo logging as a part of language semantics—compiler implementations emit logging for persistent stores in the program, transparent to the programmer. We propose logging based on strand persistency primitives. We integrate our logging mechanisms into compiler passes to implement language-level persistency model, that provide failure-atomic transactions [1], outermost critical sections [3], and SFRs [5].

III. Evaluation

We implement StrandWeaver in Gem5. We study a suite of five write-intensive micro-benchmarks and benchmarks used in prior works [5]. We compare failure-atomic transactions, Decoupled-SFR, and ATLAS, built on StrandWeaver's primitives with the Intel x86 [1] and state-of-the-art HOPS [8] designs. Due to space limitations, we list only the average improvement obtained in our designs. Strand persistency enables high persist concurrency in all designs. Overall, we achieve speedup of up to 74.6% (45.5% avg.) in failure-atomic transactions, 96.5% (46.1% avg.) in Decoupled-SFR, and 79.7% (39.9% avg.) in ATLAS over Intel x86. Compared to the epoch persistency model in HOPS [8], StrandWeaver achieves a speedup of up to 55.5% (19.8% avg.).

References

[1] "pmem.io: Persistent memory programming," https://pmem.io/pmdk/.
[2] ARM, "Armv8-a architecture evolution," 2016, tinyurl.com/arm-nvm.
[3] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," in *OOPSLA '14*, 2014.
[4] V. Gogte, W. Wang, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Strand persistency," in *NVMW'19*.
[5] V. Gogte, S. Diestelhorst, W. Wang, S. Narayanasamy, P. M. Chen, and T. F. Wenisch, "Persistency for synchronization-free regions," in *PLDI'18*.
[6] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, W. Wang, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Language support for memory persistency," *IEEE Micro '18*.
[7] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Language-level persistency," in *ISCA '17*.
[8] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with whisper," in *ASPLOS '17*.
[9] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *ISCA '14*.