TARP: Translating Acquire-Release Persistency

A. Kolli*, V. Gogte*, A. Saidi †, S. Diestelhorst †, P. M. Chen *, S. Narayanasamy *, T. F. Wenisch *
*University of Michigan {akolli,vgogte,pmchen,nsatish,twenisch}@umich.edu
† ARM {ali.saidi,stephan.diestelhorst}@arm.com

I. Introduction

The commercial release of byte-addressable persistent memories, such as Intel/Micron 3D XPoint memory, is imminent. Ongoing research has sought mechanisms to allow programmers to implement recoverable data structures in these new main memories. Ensuring recoverability requires programmer control on the order of stores to the persistent memory. We refer to the act of writing a store durably as a *persist*.

Recent works [1]–[3] have proposed *persistency models* as an extension to memory consistency models to specify an order on persists. All these persistency models have been specified at the instruction set architecture (ISA) level. That is, programmers must reason about recovery correctness at the abstraction of assembly instructions, an approach that is error prone and places an unreasonable burden on the programmer. Furthermore, since the ISA mechanisms differ in sometimes subtle ways, it is difficult to write portable recoverable programs. Just as language-level memory models make shared-memory parallel programs portable over different systems, we believe that a *language-level persistency model* is critical for writing portable recoverable software.

In this work, we propose a language-level persistency model that extends the data-race-free (DRF) memory model espoused by high-level programming language such as C++11 and Java. The DRF model guarantees a sequentially consistent execution for properly-labeled programs. Moreover, the lack of data races ensures that synchronization-free regions (*SFR*, code executed between two sync. accesses) executed on one thread become visible to other threads atomically. However, this guarantee is lost in case of unpredictable failures (*e.g.*, power-loss). Such failures may cause a partially completed SFR to become visible to recovery code, causing a data race. So, extensions to the DRF model are required to provide guarantees on the state of persistent memory in the presence of failures.

There are a range of semantics that a language-level persistency model can provide for the state of persistent memory after failure. Stronger guarantees can provide failure atomicity for SFRs or even entire critical sections. With such a strong failure-atomicity guarantee, the implementation (compiler+hardware) must provide a logging/recovery mechanism. While such guarantees make writing recoverable programs easier, they require inefficient or complex implementations. On the contrary, weaker guarantees only ensure atomicity of individual persists, allowing high-performance implementations. Existing ISA extensions for persistent memory programming from hardware vendors suggest industry's preference for such weaker guarantees. It is important to note that ordering guarantees on individual persists always allow programmers to ensure failure-atomicity at higher granularities via software logging.

In this paper, we present a language-level persistency model, *Acquire-Release Persistency* (ARP), as an extension to the C++11 memory model, that provides persist ordering guarantees required for writing recoverable programs. ARP is a single, ISA-agnostic framework for reasoning about persistency and enables porting of recoverable software across language implementations (compiler, runtime, ISA, and hardware). Furthermore, we present TARP, the compiler transformations required to translate ARP's memory events to ISA-level persistency models proposed by x86 [2] and ARM v8.2 [3].

II. Acquire-release persistency

C++11 provides low-level atomic loads and stores (`std::atomics`) with memory order semantics, acquire and release, as synchronization operations to order memory accesses. Currently, these atomic operations order accesses to volatile memory locations and do not provide any ordering guarantees on persists. ARP extends the C++11 memory model to ensure that low level atomics order persists as well.

We formally define ARP as an ordering relation over memory events—loads and stores on data variables, acquire and release memory operations on atomic variables, and full fences (`std::atomic_thread_fence`). We denote an acquire, release, and a full fence operation from thread i (on an atomic variable x in persistent memory) as PA_x^i, PR_x^i and PF^i respectively. PM_x^i represents a memory event (data load/data store/acquire/release/full fence) to persistent memory by thread i. *Persist Memory Order* (PMO) is the order enforced by ARP on all memory events and governs order of persists.

We use the following notation for ordering dependencies between memory events:

- $PM_x^i \leq_{po} PM_y^i$: PM_x^i is program ordered before PM_y^i
- $PR_x^i \leq_{sw} PA_x^j$: A release operation on atomic variable x in thread i "synchronizes with" [4] an acquire operation on atomic variable x in thread j.
- $PM_x^i \leq_p PM_y^j$: PM_x^i is ordered to persist before PM_y^j in PMO.

Ensuring intra-thread ordering: If two memory events on the same thread are separated by a full fence in program order, then they are ordered in PMO. Formally:

$$(PM_x^i \leq_{po} PF^i \leq_{po} PM_y^i) \rightarrow PM_x^i \leq_p PM_y^i \tag{1}$$

Ensuring inter-thread ordering: Inter-thread ordering is achieved using the "synchronizes with" [4] relationship between a release and a subsequent acquire operation. If two memory events are ordered via synchronization accesses, then they are ordered in PMO. Formally:

$$(PM_x^i \leq_{po} PR_s^i \leq_{sw} PA_s^j \leq_{po} PM_y^j) \rightarrow PM_x^i \leq_p PM_y^j \tag{2}$$

Furthermore, PMO is a *transitive* (and irreflexive) ordering relationship. Formally:

$$(PM_x^i \leq_p PM_y^j) \wedge (PM_y^j \leq_p PM_z^k) \rightarrow PM_x^i \leq_p PM_z^k \qquad (3)$$

A programmer can use the above three guarantees to express the desired order of persists. It is the responsibility of the compiler to translate these constraints to machine code using the ISA-level persistency model, and it is the responsibility of the hardware to enforce these constraints.

III. Intel and ARM persistency support

Next, we describe the ISA extensions that Intel x86 and ARM v8.2 provide for persistent memory programming. Writeback caches make persist order unpredictable and compromise software recoverability. x86 proposes a cache-line write back `clwb` instruction that writes-back a dirty cache line to a persistent write queue in the memory controller. Similarly, ARM v8.2 proposes a cache-line writeback instruction `dc cvap` which writes-back the dirty cache block to the point of persistence. In this work, we assume the point of persistence to be the write queue at the persistent memory controller (e.g., via Asynchronous DRAM Refresh support). The writebacks issued by `clwb` in x86 and `dc cvap` in ARM also need to be ordered to ensure the desired order of persists.

As explained in Section II, the desired inter-thread persist orderings are specified in the program by atomic operations with acquire or release semantics. ARM v8.2 implements the release consistency memory model to order memory operations. It introduces load `ldar` and store `stlr` instructions with acquire and release memory ordering semantics, respectively. The `ldar` instruction orders the memory operations, including writebacks caused by `dc cvap`s, occurring after it in program order so that they may not be observed to happen before the `ldar`. Similarly, the `stlr` instruction orders the memory operations, including `dc cvap`s, that occur prior to it in program order so that they may not be observed to happen after `stlr`. However, x86 exposes only a full fence instruction, `sfence`, that orders stores and writebacks caused by `clwb`s occurring prior to it in program order with those occurring after. Note that x86 does not exploit the unidirectionality of acquire and release fences in C++11 and enforces a stronger ordering with `sfence`.

Finally, the intra-thread ordering between different memory events is ensured using the full fence instruction `dmb ish` in case of ARMv8.2 and `sfence` in case of x86. Both `dmb ish` and `sfence` instructions enforce bidirectional ordering between prior and subsequent memory operations including the writebacks caused by `dc cvap`s and `clwb`s respectively.

IV. Compiler transformations

We now present the transformations in TARP that compiler has to provide for translating the ARP persistency model into x86 and ARM v8.2 instructions, as illustrated in Table I. We then propose compiler strategies that can optimize and reduce the cost of enforcing persist order in hardware.

ARP memory events	**Intel x86**	**ARM v8.2**
Store to addr a	mov; clwb a;	str a; dc cvap a;
Load from addr a	mov;	ldr;
Load Acquire on addr a	mov; clwb a; sfence;	ldar a; dc cvap; dmb ish
Store Release on addr a	sfence; mov; clwb a;	stlr a; dc cvap a;
Full Fence	sfence;	dmb ish;

TABLE I: Mapping from ARP memory events to Intel x86 and ARM v8.2 ISA, given address `a` resides in persistent memory

- *Memory accesses*: A store to a persistent address translates to a `mov` instruction followed by a writeback, `clwb`, in x86. Similarly, `dc cvap` is emitted after every `str` in ARM v8.2. Loads do not initiate any persistent actions.
- *Store release*: A store release translates to an `sfence` followed by a `mov` instruction in x86. If release operation is to a persistent address `a`, a `clwb` is also emitted to write back the cache-line. The `sfence` is required to guarantee the "release"-ness of the store. In ARM v8.2, a store release directly translates to store with release semantics, `stlr`. Similar to x86, a writeback instruction, `dc cvap` is emitted if the release is to a persistent memory address.
- *Load acquire*: A load acquire translates to a `mov` instruction (to load the value) followed by an `sfence` instruction. Note that a `clwb` is also issued after the load if the acquire is to a persistent address. The `clwb` and the following `sfence` ensure that the observed value of `a` indeed persists before subsequent stores that depend on the observed value. In ARM v8.2, a load acquire directly translates to a load with acquire semantics, `ldar`. Similar to x86, if an acquire is to a persistent address, a `dc cvap` followed by a full fence `dmb ish` is emitted after the `ldar`. The full fence ensures that dependent stores do not persist before the store that produces the observed value of the load acquire.
- *Full fence*: In x86, a full fence is implemented using a `sfence`. The `sfence` ensures that the prior stores are persisted (using cache-line write backs) before the later stores. In ARM v8.2, the intra-thread ordering of a full fence is implemented using `dmb ish`.

Compiler optimizations: TARP's first pass transforms the memory operations in a C++11 program to the ISA-level persistency primitives as described above. Then, peephole optimizations can reduce the number of cache line writebacks. For instance, multiple writeback instructions for the same cache block in an SFR may coalesced. Similarly, writebacks can be scheduled within an SFR to overlap their latency with execution and minimize stalls at a fence. TARP can also use memory access re-ordering optimizations that are allowed across uni-directional fences to improve performance.

References

[1] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *International Symposium on Computer Architecture*, 2014.

[2] Intel, "Intel architecture instruction set extensions programming reference (319433-022)," 2014. https://software.intel.com/sites/default/files/managed/0d/53/319433-022.pdf.

[3] ARM, "Armv8-a architecture evolution," 2016. https://community.arm.com/groups/processors/blog/2016/01/05/armv8-a-architecture-evolution.

[4] H.-J. Boehm and S. V. Adve, "Foundations of the c++ concurrency memory model," in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2008.