Persistency programming 101

Aasheesh Kolli[§], Steven Pelley[†], Ali Saidi[‡], Peter M. Chen[§], and Thomas F. Wenisch[§]

[§]University of Michigan, {akolli,pmchen,twenisch}@umich.edu
[†]Snowflake Computing, steven.pelley@snowflakecomputing.com
[‡]ARM, ali.saidi@arm.com

Emerging non-volatile memory (NVRAM) technologies offer the durability of disk with byte-addressability and access latencies similar to DRAM. Future systems will likely attach these NVRAMs to a DRAM-like memory bus [1, 2]. Such systems enable the construction of high performing, in-memory, recoverable data structures (RDS) [1, 2]. The tenets of creating RDSs revolve around ordering writes to the data structure. However, existing architectures do not provide efficient mechanisms to order writes all the way through NVRAM. Pelley, Chen, and Wenisch [3] introduce persistency models (drawing on memory consistency model research) to reason about and order writes to NVRAM. Here, we introduce notation to concisely and precisely define the models to help draw better parallels to existing memory consistency models. These precise definitions make it easier to reason about the interaction between instruction execution and NVRAM write order. Achieving the desired order of NVRAM writes across threads is tricky. We show two generic coding patterns to illustrate how to leverage persistency models to achieve the desired order of writes. In particular, one code pattern (*observe*) leverages relaxed persistency models and can be used to enforce only the absolute minimum NVRAM write orderings required for correct recovery.

1. Memory persistency models

Memory events: We formalize Pelley's persistency models in terms of order relations over memory events. We consider two kinds of events, *loads* and *stores* (to persistent or volatile address spaces), which we collectively call *memory accesses*. We use the term "persist" to refer to the act of durably writing a store to persistent memory. We assume persists are performed atomically (with respect to failures) at 8-byte granularity. By "thread", we refer to execution contexts—cores or hardware threads. We use the following notation:

- L_a^i: A load from thread i to address a
- S_a^i: A store from thread i to address a
- M_a^i: A load or store by thread i to address a

Persist memory order: We reason about two ordering relations over memory events. *Volatile memory order* (VMO) is an ordering relation over all memory events (loads, stores and their values) as prescribed by the consistency model. *Persist memory order* (PMO) comprises the same events, however, events are instead ordered by the constraints imposed by the persistency model. We denote these ordering relations as:

- $A \leq_v B$: A occurs no later than B in VMO
- $A \leq_p B$: A occurs no later than B in PMO

An ordering relation between stores in PMO implies the corresponding persist actions are ordered; that is, $A \leq_p B \rightarrow B$ may not persist before A.

Strong persist atomicity: Memory consistency models often guarantee that stores to a single address are serialized (*store atomicity*). Persistency models could guarantee *persist atomicity*, persists to the same address are serialized. Pelley argues persistency models should provide *strong persist atomicity* (SPA), to preclude non-intuitive behavior (e.g., recovery to states unreachable under fault-free execution). SPA requires that conflicting accesses (accesses to the same address, at least one being a store) ordered in VMO are also ordered in PMO.

$$\begin{aligned} S_a^i \leq_v M_a^j \rightarrow S_a^i \leq_p M_a^j \\ M_a^i \leq_v S_a^j \rightarrow M_a^i \leq_p S_a^j \end{aligned} \tag{1}$$

SPA is guaranteed by *all* the persistency models described below. SPA is critical to constructing precise ordering dependences across threads (see section 2).

Strict persistency: Under strict persistency, the consistency model governs both VMO and PMO; that is, the two are identical. So, for any two stores ordered by the consistency model, the corresponding persists are also ordered. Under strict persistency:

$$M_a^i \leq_v M_b^j \leftrightarrow M_a^i \leq_p M_b^j \tag{2}$$

While strict persistency is the most intuitive of the persistency models, it is not the best performing. By ordering persists per VMO, strict persistency enforces orderings typically not required for recovery correctness [3].

Relaxed persistency: Relaxed persistency models govern PMO independent of the memory consistency model. These models provide programmers with additional memory events, to prescribe only those ordering constraints in PMO required to ensure correct recovery.

Epoch persistency: The *epoch persistency model*, a relaxed persistency model, introduces a new memory event, the "persist barrier" (different from memory consistency barriers). We denote persist barriers issued by thread i as PB^i. Under epoch persistency, any two memory accesses on the same thread separated by a persist barrier are ordered in PMO.

$$M_a^i \leq_v PB^i \leq_v M_b^i \rightarrow M_a^i \leq_p M_b^i \tag{3}$$

Persist barriers separate a thread's execution into ordered epochs (persists from an epoch are concurrent). Epoch persistency is similar to the model described in [2], though Pelley introduces some subtle differences.

```
Thread 1     Thread 2
lock L
store A
PB
unlock L
             lock L
             PB
             store B
             unlock L
```

(a) Lock

```
Thread 1     Thread 2
lock L
store A
unlock L
             lock L
             load A
             PB
             store B
             unlock L
```

(b) Observe

Figure 1: Ordering persists across threads/strands.

Strand persistency: Strand persistency divides program execution into *strands.* Strands are logically independent segments of execution that happen to execute in the same thread. Strands are separated by the *strand barrier* (SB) memory event. Strand barriers from thread i are denoted as SB^i. The strand barrier clears all prior PMO constraints from prior instructions, effectively making each strand a separate thread (with respect to persistency). Memory accesses within a strand are ordered using persist barriers (eq 3). Under strand persistency, any two memory accesses on the same thread separated by a persist barrier without an intervening strand barrier are ordered in PMO.

$$(M_a^i \leq_v PB^i \leq_v M_b^i) \wedge (\nexists SB^i : M_a^i \leq_v SB^i \leq_v M_b^i) \rightarrow M_a^i \leq_p M_b^i \qquad (4)$$

2. Coding Patterns to Order Persists

We next discuss coding patterns that can be used to enforce desired persist ordering constraints across threads/strands. Persists on a single thread (or strand under strand persistency) are ordered either by VMO (strict persistency) or by persist barriers (epoch and strand persistency). Enforcing persist order across threads is more complex; as with VMO, these orderings must be established using conflicting accesses. For the sake of brevity, from here on we assume that the underlying memory consistency model is sequential consistency (SC).

Figure 1 illustrates coding patterns to establish order for a simple scenario under each model. Consider two stores to addresses A and B, executed on different threads (or strands), which are protected by a single lock L, we assume thread 1 wins: $S_A^1 \leq_v S_B^2$. Our objective is to use the persistency models to ensure that the persists of A and B follow the same order: $S_A^1 \leq_p S_B^2$.

The definition of strict persistency (eq 2) ensures the desired order of persists. Below, we describe two techniques *lock* and *observe*, employed under relaxed persistency models to achieve the desired persist ordering.

Lock: The central intuition is to leverage the conflicting accesses of the concurrency control mechanism (i.e., locks), which establish required constraints (e.g., mutual exclusion) in VMO, to also establish the required ordering constraints in PMO. Figure 1a shows how to order persists to A and B under epoch persistency using persist barriers PB^1 and PB^2. We denote the unlock operation on thread 1 as S_L^1 and the lock operation on thread 2 as S_L^2. The program orders of thread 1, thread 2 and the ordering property of persist barriers (eq. 3) ensures that:

$$S_A^1 \leq_v PB^1 \leq_v S_L^1 \rightarrow S_A^1 \leq_p S_L^1 \qquad (5)$$

$$S_L^2 \leq_v PB^2 \leq_v S_B^2 \rightarrow S_L^2 \leq_p S_B^2 \qquad (6)$$

From conflicting accesses to lock L and SPA (eq 1)

$$S_L^1 \leq_v S_L^2 \rightarrow S_L^1 \leq_p S_L^2 \qquad (7)$$

By transitivity and eqs. 5-7, we ensure that $S_A^1 \leq_p S_B^2$. This same reasoning extends to strands (instead of threads) under strand persistency.

Observe: Instead of relying on lock L for conflicting accesses, we can explicitly *observe* (using loads) the specific addresses after which subsequent persists should be ordered, and then issue a persist barrier. Figure 1b illustrates this pattern. S_A^1's persist is unordered with respect to any other persist on Thread 1 or (absent L_A^2 and PB^2 we have included) Thread 2. Note that the lock L still ensures mutual exclusion and ordering of the (volatile) execution of the critical sections but, by itself, will not order the persists of A and B. Since thread 1 acquires lock L first, from VMO and SPA (eq 1), we have:

$$S_A^1 \leq_v L_A^2 \rightarrow S_A^1 \leq_p L_A^2 \qquad (8)$$

From the program order of Thread 2 and the ordering property of persist barriers (eq. 3), we have:

$$(L_A^2 \leq_v PB^2 \leq_v S_B^2) \rightarrow L_A^2 \leq_p S_B^2 \qquad (9)$$

By transitivity and eqs. 8 and 9, we have $S_A^1 \leq_p S_B^2$. Again, the above reasoning extends to strands as well. In fact, by placing all persists on their own strands and using the observe technique, it is possible to enforce only the required ordering constraints, even under SC.

To conclude, we introduced precise definitions for persistency models and used the definitions to detail generic approaches to enforce the desired order of persists.

References

[1] J. Coburn, A. Caulfield, A. Akel, L. Grupp, R. Gupta, R. Jhala, and S. Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of ASPLOS*, 2011.

[2] J. Condit, E. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of SOSP*, 2009.

[3] S. Pelley, P. Chen, and T. Wenisch. Memory persistency. In *Proceedings of ISCA*, 2014.