Linearizability of Persistent Memory Objects

Michael L. Scott
Joint work with
Joseph Izraelevitz & Hammurabi Mendes

www.cs.rochester.edu/research/synchronization/
Workshop on the Theory of Transactional Memory
Chicago, IL, July 2016

brief announcement at SPAA’16; full paper to appear at DISC’16
Fast Nonvolatile Memory

- NVM is on its way
  - PCM, STT-MRAM, memristors, ...

- Tempting to put some long-lived data directly in NVM, rather than the file system

- But registers and caches are likely to remain transient, at least on many machines

- Have do we make sure what we get in the wake of a crash is consistent?
Problem: Early Writebacks

- Could assume HW tracks dependences and forces out earlier stuff
  - [Condit et al., Pelley et al., Joshi et al.]
- But real HW not doing that any time soon; have to explicitly force things out in order
  - ARM, Intel ISAs
- Buffering?
  - Can be done in SW now, with shadow memory
    - Likely to be supported in HW eventually
Outline

- Theory review — linearizability of (transient) concurrent object histories
- Extension of theory to persistence — *durable linearizability*
- *Explicit epoch persistency* at the hardware level
  - to explain the behavior of implementations
- Automatic transform to convert a (correct) transient nonblocking object into a (correct) persistent one
- Methodology to prove safety for more general objects
- Future directions
Theory Review

- Focus on *objects* that we can put in libraries
  - data abstractions defined in terms of API (methods)
  - stack, queue, deque, set, mapping, priority queue, ...

- Many possible implementations (data structures)
  - Correctness = *safety* + *liveness*
  - Focus on safety in today’s talk
    (DISC paper also looks at liveness)
Object Histories

- Interleavings of operations (method invocations) performed by a set of threads
  - Concrete history: all the instructions
  - Abstract history: invocations and responses only (calls & returns)

```
I · · I · R · · · I I R · · R I · · I R · R ·
```

- **Sequential** = every invocation followed immediately by its response
- **Well formed** = every thread subhistory is sequential
Safety

- Implementations generate concrete histories
- Implementation is safe if every realizable (single-object) concrete history corresponds to a (well-formed) safe abstract history
- Safe abstract history = \textit{linearizable} (next slide)
- Object semantics defined as a set of abstract sequential histories
  - e.g., a queue is an object with enqueue & dequeue methods, where the \( n \)th dequeue yields the value passed to the \( n \)th enqueue, if there has been one, else \( \perp \)
Linearizability [Herlihy & Wing 1987]

- Standard safety criterion for transient objects
- History $H$ is safe if well-formed and equivalent (same invocations and responses, inc. args) to some sequential history $S$ that respects
  1. object semantics (legal)
  2. “real-time” order ($\text{res}(A) <_H \text{inv}(B) \Rightarrow A <_S B$) (subsumes per-thread program order)
- Programs, of course, generate *multi-object* concrete histories
- Linearizability is nice because it's a *local* property: safety of individual objects implies safety of multi-object programs
  » Follows from respect of real-time order
- Need an extension for persistence
Outline

- Theory review — linearizability of (transient) concurrent object histories
- Extension of theory to persistence — durable linearizability
- Explicit epoch persistency at the hardware level
  - to explain the behavior of implementations
- Automatic transform to convert a (correct) transient nonblocking object into a (correct) persistent one
- Methodology to prove safety for more general objects
- Future directions
Prior Work on Persistency

- **Strict linearizability** [Aguilera and Frølund 2003]
  - At a crash, every pending operation has happened or it hasn't
  - Too restrictive — can't leave anything hanging (e.g., announce array)

- **Persistent atomicity (linearizability)** [Guerraoui et al. 2004]
  - Every pending operation happens before its thread invokes anything post-crash
  - Gives up on locality — have to reason across objects

- **Recoverable linearizability** [Berryhill et al. 2015]
  - Every pending operation happens before its thread invokes anything on the same object post-crash
  - Gives up on program order around a crash — thread can perform an op on some other object before “coming back to” the pending op
Comparing These Conditions

- Where must $T_1$’s pending op linearize?

- Persistent and Recoverable Linearizability are the same if threads don’t survive a crash — and they don’t in real life!

- We use Recoverable Linearizability for the merged condition, under a full-system–crash failure model.
Durable Linearizability

- (Abstract) history $H$ is *durably linearizable* iff
  1. it's well formed (no thread survives a crash) and
  2. $\text{ops}(H)$ is linearizable (elide the crashes)
- But that requires every op to persist before returning
- Want a *buffered* variant
- Say $A$ “happens before” $B$ ($A \prec B$) in an abstract history $H$ if $A.\text{res}$ precedes $B.\text{inv}$ in $H$
- A $\prec$-consistent cut of a crash-free history $H$ is a prefix $P$ of $H$ such that if $V \in P$ and $V' \prec V$ in $H$, then $V' \in P$, with $V' \prec V$ in $P$. 
Buffering

- (Abstract) history $H$ is \textit{buffered durably linearizable} iff for each era $E_i$ we can identify a $\prec$-consistent cut $P_i$ such that $P_0 ... P_{i-1} E_i$ is linearizable $\forall 0 \leq i \leq c$, where $c$ is the number of crashes.

  » That is, we may lose something at each crash, but what's left makes sense. (Again, buffering may be in HW or in SW.)

- NB: Because actual persistence is delayed, and must be controlled across objects, buffered durable linearizability is not a local property.
Outline

- Theory review — linearizability of (transient) concurrent object histories
- Extension of theory to persistence — *durable linearizability*
- *Explicit epoch persistency* at the hardware level
  - to explain the behavior of implementations
- Automatic transform to convert a (correct) transient nonblocking object into a (correct) persistent one
- Methodology to prove safety for more general objects
- Future directions
Memory Model Background

- Sequential consistency: memory acts as if there was a total order on all loads and stores across all threads
  - Conceptually appealing, but only IBM z still supports it
- Relaxed models: separate *ordinary* and *synchronizing* accesses
  - Latter determine cross-thread ordering arcs
  - *Happens-before order* derived from program order and synchronizes with
- Release consistency (ARM v8): each store-release synchronizes with the following load-acquire of the same location
  - Each local access happens after each previous load-acquire and before each subsequent store-release in its thread
- But none of this addresses persistence
Persistence Instructions

- Explicit write back ("pwb"); persistence fence ("pfence"); persistence sync ("psync")

- We assume E1 ≪ E2 if
  - they're in the same thread and
    - E1 = pwb & E2 ∈ {pfence, psync}
    - E1 ∈ {pfence, psync} and E2 ∈ {pwb, st, st_rel}
    - E1, E2 ∈ {st, st_rel, pwb} and access the same location
    - E1 ∈ {ld, ld_acq}, E2 = pwb, and access the same location
    - E1 = ld_acq and E2 ∈ {pfence, psync}
  - they’re in different threads and
    - E1 = st_rel, E2 = ld_acq, and E1 synchronizes with E2
Concrete Histories

- H is well-formed iff
  - abstract(H) is well-formed
  - all instructions are between inv. and res. (or crash or end)
  - loaded values respect the reads-see-writes relation
    - return a most recent or unordered store under happens-before

- NB: Implementations (programs) give us sets of possible histories — possible interleavings.

- A history is data-race-free if conflicting accesses are never adjacent; an implementation is DRF if all of its realizable histories are DRF.
Extensions for Persistence

- H is well-formed iff all previous requirements and
  - for each variable x, at most one store is labeled “persisted” in each era
  - no unpersisted store of x between the persisted store of x and (1) a psync or (2) the persisted store of any other location y
- NB: reads-see-writes augmented to allow returning the persisted store of the previous era in the wake of a crash
- Key problem: you see a write, act on it, and persist what you did, but the original write doesn't persist before we crash.
Outline

● Theory review — linearizability of (transient) concurrent object histories
● Extension of theory to persistence — durable linearizability
● Explicit epoch persistency at the hardware level
  » to explain the behavior of implementations
● Automatic transform to convert a (correct) transient nonblocking object into a (correct) persistent one
● Methodology to prove safety for more general objects
● Future directions
Persistence Transform

- $st \rightarrow st; \ pwb$
  - $st_{rel} \rightarrow pfence; \ st_{rel}; \ pwb$
  - $ld_{acq} \rightarrow ld_{acq}; \ pwb; \ pfence$
  - $cas \rightarrow pfence; \ cas; \ pwb; \ pfence$
  - $ld \rightarrow ld$

- Can prove: if the original code is DRF and linearizable, the transformed code is durably linearizable.
  - Key is the $ld_{acq}$ rule

- But: not all stores have to be persisted
  - elimination/combining, announce arrays for wait freedom

- How do we build a correctness argument for more general, hand-written code?
Outline

● Theory review — linearizability of (transient) concurrent object histories
● Extension of theory to persistence — *durable linearizability*
● *Explicit epoch persistency* at the hardware level
  » to explain the behavior of implementations
● Automatic transform to convert a (correct) transient nonblocking object into a (correct) persistent one
● Methodology to prove safety for more general objects
● Future directions
Linearization Points

● Every operation “appears to happen” at some individual instruction, somewhere between its call and return.

● Proofs commonly leverage this formulation
  » In lock-based code, could be pretty much anywhere
  » In simple nonblocking operations, often at a distinguished CAS

● In general, linearization points
  » may be statically known
  » may be determined by each operation dynamically
  » may be reasoned in retrospect to have happened
  » (may be executed by another thread!)
Persist Points

● Proof-writing strategy (again, challenge is make sure nothing new persists before something old on which it depends)

● Implementation is (buffered) durably linearizable if
  1. somewhere between linearization point and response, all stores needed to "capture" the operation have been pwb-ed and pfence-d;
  2. whenever M1 & M2 overlap, linearization points can be chosen s.t. either M1’s persist point precedes M2’s linearization point, or M2’s linearization point precedes M1’s linearization point.

● NB: nonblocking persistent objects need helping: if an op has linearized but not yet persisted, its successor in linearization order must be prepared to push it through to persistence.
Objects from the Literature

- Many strictly linearizable
  - trees [Chen & Jin, VLDB’15; Venkataraman et al., FAST’11; Yang et al., FAST’15]
  - hash maps [Schwalb et al., IMDM’15]
  - file system metadata [Condit et al., SOSP’09]
- Also a few buffered strictly linearizable [Moraru et al., TRIOS’13]
- And a few durably (but not strictly) linearizable
  - JustDo Logging [Izraelevitz et al., ASPLOS’16]
  - Hope to see more!
Outline

- Theory review — linearizability of (transient) concurrent object histories
- Extension of theory to persistence — *durable linearizability*
- *Explicit epoch persistency* at the hardware level
  - to explain the behavior of implementations
- Automatic transform to convert a (correct) transient nonblocking object into a (correct) persistent one
- Methodology to prove safety for more general objects
- Future directions
Ongoing Work

- More optimized, nonblocking persistent objects
- Integrity in the face of buggy (Byzantine) user threads
  - File system no longer protects metadata!
- Integration w/ transactions
- Suggestions welcome!
Liveness

- **Nonblocking sync** — bounded # of own steps
- **Bounded completion** — in the wake of a crash, for each operation m that was pending on object O, \( \exists k \) s.t. if some post-crash era has at least k instructions executed in O, m has completed or it never will.