# Data Structures for Persistent Memory

CSC 2/458

April 2019

Joint work with Joseph Izraelevitz, Hammurabi Mendes,
Faisal Nawab, Terrence Kelly, Charles Morrey, Dhruva Chakrabarti,
Virendra Marathe, Qingrui Liu, Se Kwon Lee, Sam Noh, and
Changhee Jung

# Fast Nonvolatile Memory

- NVM is on its way: PCM, ReRAM, STT-MRAM, ...
  - Could just treat it as dense, low-power DRAM replacement
  - 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 (power failure) is consistent?
  - Implications for algorithm design & for compilation.

- (Could also consider full-system persistence — not the topic of this talk.)

# Problem: Early Writes-back

- 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 — writes-back can happen in any order
  - Danger that B will perform — and persist — updates based on actions taken but not yet persisted by A
  - Have to explicitly force things out in order (ARM, Intel ISAs)

# Outline

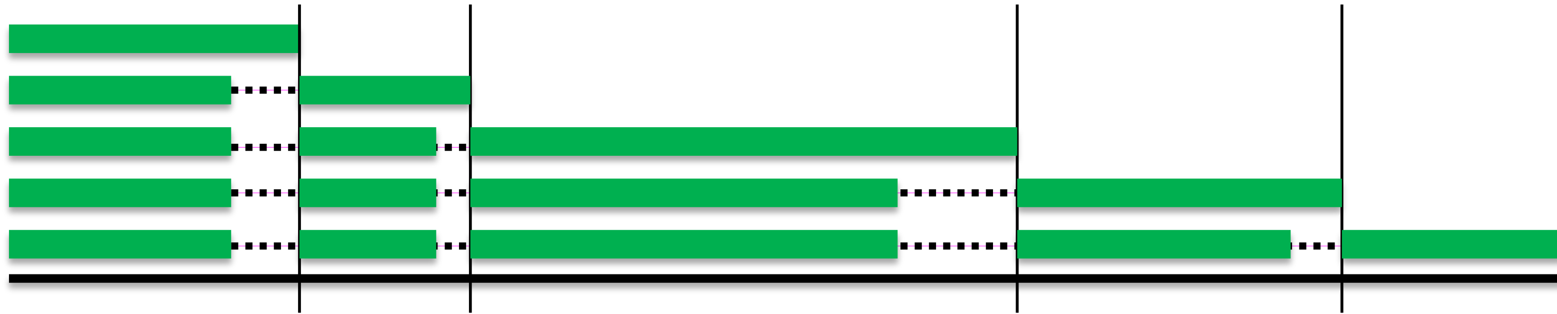
- Formal framework for persistency [DISC'16]
  - High level semantics – *durable linearizability*
  - Hardware memory model – *explicit epoch persistency*
- Incremental persistence
  - Mechanical conversion of (correct) transient nonblocking object into a (correct) persistent one
  - Methodology to prove safety for more general objects
- Reducing the frequency of fences
  - JUSTODO [ASPLOS'16] and iDO logging [MICRO'18]
- Ensuring (meta)data integrity
  - ~~Janus~~ [ATC'19]
    Hodor

# Linearizability [Herlihy & Wing 1987]

- Standard safety criterion for transient objects
- Concurrent execution H guaranteed to be equivalent (same invocations and responses, inc. args) to some sequential execution S that respects
  1. object semantics (*legal*)
  2. "real-time" order (res(A) $<_H$ inv(B) $\Rightarrow$ A $<_S$ B)
     (subsumes per-thread program order)
- Need an extension for persistence

# Durable Linearizability
## [Izraelevitz et al., DISC'16]

- Execution history H is *durably linearizable* iff
  1. It's well formed (no thread survives a crash) and
  2. It's linearizable if you elide the crashes
- But that requires every op to persist before returning
- Want a *buffered* variant
- H is *buffered durably linearizable* iff for each inter-crash *era* $E_i$ we can identify a *consistent cut* $P_i$ of $E_i$'s real-time order such that $P_0 \ldots P_{i-1}\ E_i$ 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.)



# Proving Code Correct

- Need to show that all realizable instruction histories are equivalent to legal abstract (operation-level) histories.
- For this we need to understand the hardware *memory model,* which determines which writes may be seen by which reads.
- And that model needs extension for persistence.

# Memory Model Background

- Sequential consistency: memory acts as if there were 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.
  - Within a thread, ordinary accesses ordered wrt synchronizing accesses.
  - Synchronizing accesses ordered across threads.
  - Transitive closure defines *happens-before* relationship.
  - A read will see the most recent write on a happens-before path, or a write that is not ordered by happens-before.
- None of this addresses persistence.

# Persistence Instructions

- Explicit write back (“pwb”); persistence fence (“pfence”); persistence sync (“psync”) — idealized.
- 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.

# Explicit Epoch Persistency

- With persistence, the *reads-see-writes* relationship must be augmented to allow returning a value persisted prior to a recent crash.
  - In an era ending with a crash, at most one write of each location will be "the" persisted write. HW guarantees that these represent a consistent cut of the *persists-before* order. All are said to happen before everything in the next era.
  - Then, as usual, a read will see the most recent write on a happens-before path, or a current-era write that is not ordered by happens-before.
- How do we ensure that a structure is consistent after a crash?

# Post-crash Usability

- Sufficient but not necessary condition:
  - If we can guarantee that persists-before is consistent with happens-before, then a nonblocking structure will always be usable.
  - Also, a blocking structure will be usable if undo or redo logging allows us to roll back or forward to a critical section boundary.

# Incremental Persistence

- Mechanical transform:

  | | |
  |---|---|
  | st | → st; pwb |
  | st_rel | → pfence; st_rel; pwb |
  | ld_acq | → ld_acq; pwb; pfence |
  | cas | → pfence; cas; pwb; pfence |
  | ld | → ld |

- Can prove: if the original code is DRF and linearizable, the transformed code is durably linearizable.
  - Key is the ld_acq rule.
- If original code is nonblocking, recovery process is null.
- But not all stores *have* to be persisted!
  - Elimination/combining, announce arrays for wait freedom, ...
  - (This is the “but not necessary” part.)

# 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

- (Sufficient, weaker, but still not necessary) proof-writing strategy.
- 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.

# Fewer Fences

- Writes-back aren’t expensive: waiting for them is.
- Want to do a bunch of writes between fences.
- iDO logging: leverage idempotent regions.
- Periodic persistence: leverage functional persistence (history preserving updates).

# JUSTDO Logging
## [Izraelevitz et al, ASPLOS'16]

- Designed for a machine with *nonvolatile caches.*
- Goal is to assure the atomicity of (lock-based) *failure-atomic sections* (FASEs).
- Prior to every write, log (to that cache) the PC and the live registers.
- In the wake of a crash, *execute* the remainder of any interrupted FASE.

# iDO Logging

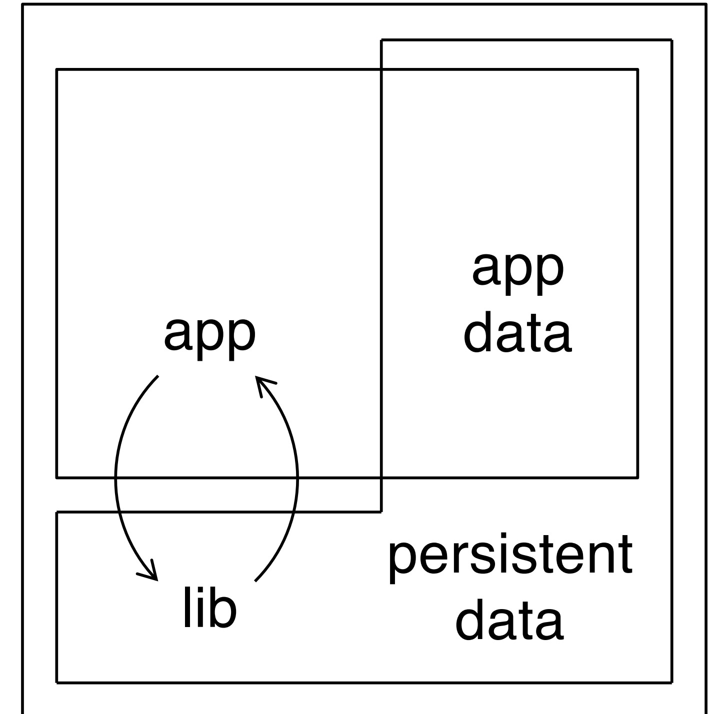
## [Joint work w/ Qingrui Liu, Se Kwon Lee, Sam Noh, and Changhee Jung at Virginia Tech]

- JUSTDO logging is (perhaps) fast enough to use with nonvolatile caches (less than an OOM slowdown of FASEs), but not w/ volatile caches (2 orders of magnitude).
- Key observation: programs have *idempotent regions* that are 10s or 100s or instructions.
- Key idea: do JUSTDO logging at i-region boundaries
- On recovery, complete each interrupted FASE, starting at beginning of interrupted i-region.

# Hodor
# ~~Janus~~: Protected Libraries

- Traditional file system protects metadata.
- Mmap-ed persistent (meta)data opens new vulnerabilities
  - Buggy programs lead to Byzantine faults.
  - (Even in the absence of a malicious adversary.)
- Division between data and metadata also fuzzy
  - Consider integrity of hash chains in memcached.

# Ensuring (meta) data integrity




- Want to allow only trusted library to access protected (persistent) data.
- ~~Janus~~ Hodor system [Usenix ATC'19]:
  - Leverage Intel PKU mechanism
  - Change protections when crossing into/out of library
  - Prevent spurious use of WRPKRU instruction via compiler help, binary scanning/rewriting, and/or use of debug registers
- Future work:
  - Killer apps: high throughput devices, in-core databases, window system – cf. work on microkernels
  - Tolerance of/recovery from independent failues

# Other Ongoing Work

- More optimized, nonblocking persistent objects.
- Integration of persistence and transactional memory.
- Nonblocking persistent heap management.
- "Systems" issues — replacing (some) files with persistent segments.
  - What are (cross-file) pointers?
- Integration w/ distribution (is this even desirable?)

UNIVERSITY of ROCHESTER

www.cs.rochester.edu/research/synchronization/

www.cs.rochester.edu/u/scott/