Programming for Persistent Memory

Michael L. Scott

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

Summer School on Practice & Theory of Distributed Computing
July 2020

The University of Rochester

- Small private research university
- 6800 undergraduates
- 5000 graduate students
- Set on the Genesee River in Western New York State, near the south shore of Lake Ontario
- 250km by road from Toronto; 590km from New York City

The Computer Science Dept.

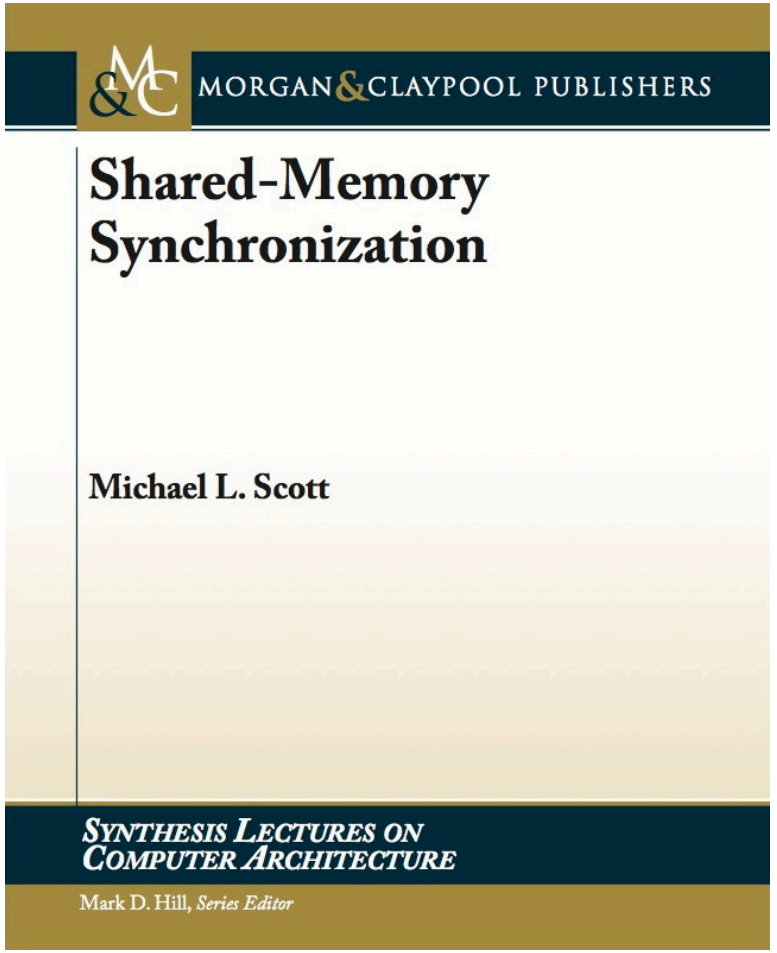
- Founded in 1974
- 20 tenure-track faculty; 70 Ph.D. students
- Specializing in AI, theory, HCI, and parallel and distributed systems
- Among the best small departments in the US

Introduction

- Will assume some familiarity with multithreaded programming (ideally in C/C++), ideally including nonblocking data structures and synchronization techniques
- Have a total of 3 hours; will survey material that was the subject of an entire class this past semester (!)
- Will emphasize cross-cutting issues and techniques
  - » You can look up the details of individual structures, concrete performance results
- *Please interrupt with questions!*

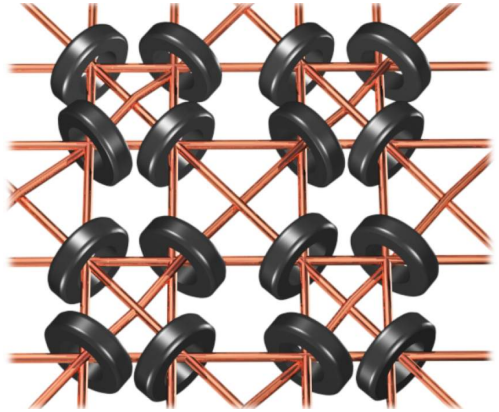
For parallel programming background

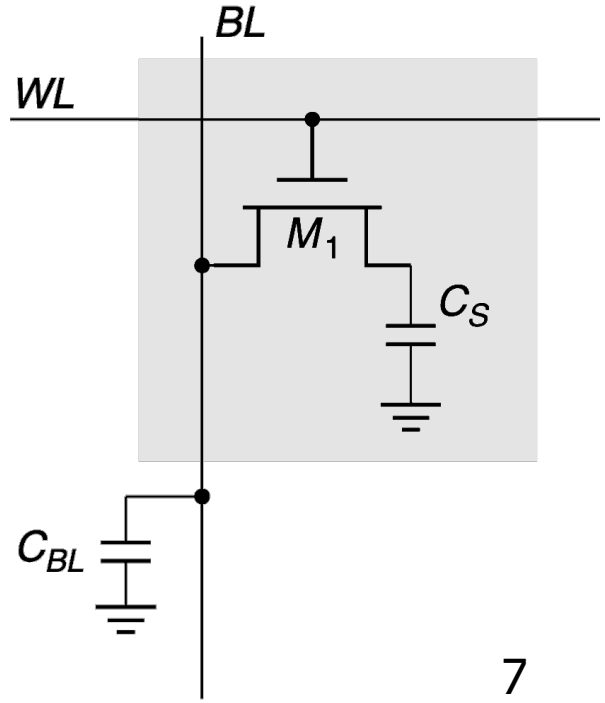
- See the Morgan & Claypool monograph, © 2013
  - Especially chapters 1, 2, 3, & 8
  - Likely in your library
  - 2nd edition planned



The very long reign of DRAM

- Main memory in the 1960s was nonvolatile—magnetic core. DRAM took over in the 1970s.
- But DRAM density has stalled.
  - » Grew from ~1Kb in 1970 to ~16Gb in 2018 — 7 orders of magnitude in 45 years.
  - » But significant further increases are not expected: smaller capacitors leak too much.
- DRAM is also power hungry.
  - » Requires refresh at ~1KHz.
  - » More than 25% of data center energy consumption.





Enter nonvolatile memory (NVM)

- Several new technologies available or in the lab
  - » Designed for higher density and lower energy consumption.
  - » As a side effect, *nonvolatile* — content is retrained on power loss.

1. Could just use the same way we've always used DRAM
   - » The distinction between transient memory and persistent storage may just be a good idea!
2. Or we could rethink the storage hierarchy
   - » Keep data "in memory" across program runs and even system crashes

★ This class explores the 2nd option

Class outline

- NVM technology and hardware architecture
- Challenges of using NVM
- Correctness criteria for persistent “in memory” data

- Persistent data structures
- Persistent run-time systems
- Persistent memory allocation
- Other issues

Class outline

- NVM technology and hardware architecture
- Challenges of using NVM
- Correctness criteria for persistent “in memory” data

- Persistent data structures
- Persistent run-time systems
- Persistent memory allocation
- Other issues

NVM varieties

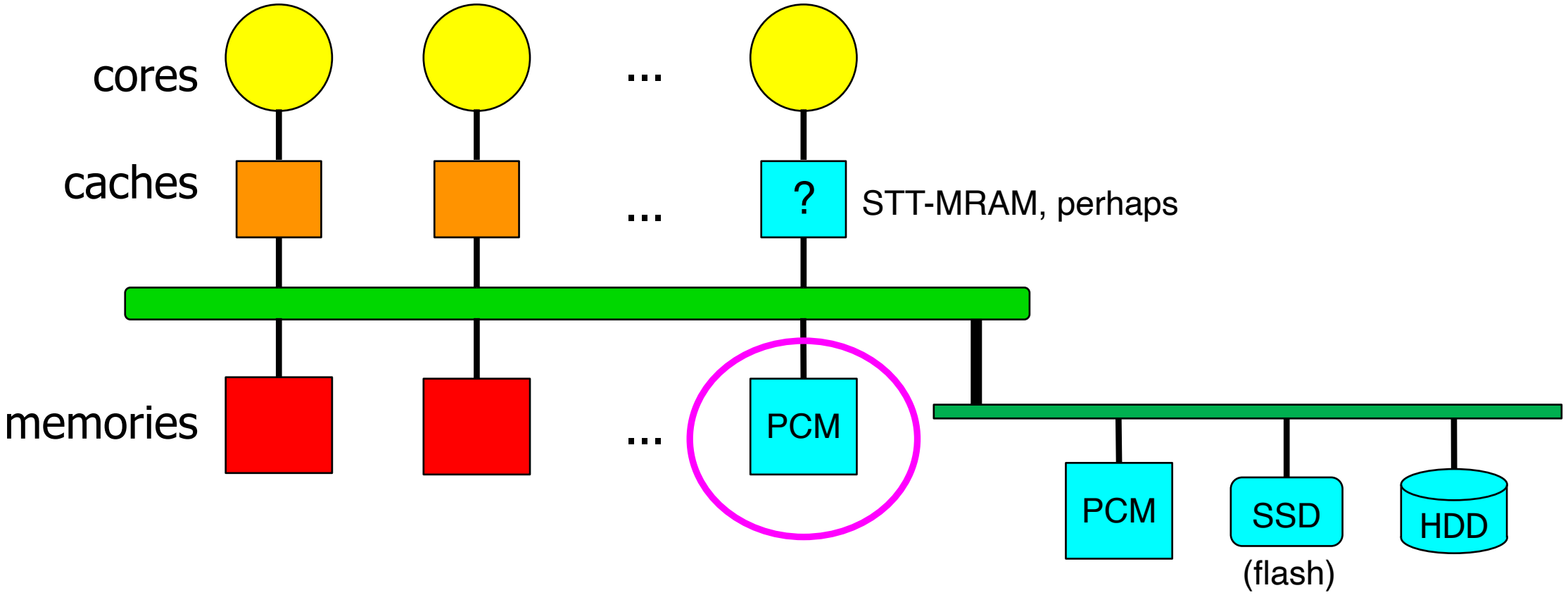
- Phase-change memory (PCM) — e.g., Intel Optane
  - » Amorphous vs crystalline chalcogenide — Cf. rewritable CDs
- Resistive RAM (ReRAM) — a.k.a. memristors
  - » Ion migration in a metal oxide to make or break conductive filaments
- Spin-transfer torque magnetic memory (STT-MRAM)
  - » Electron spin alignment; process-compatible with CMOS (for caches?)
- Ferro-magnetic RAM (FeRAM)
  - » Used in IoT for many years; speed and density slightly worse than DRAM

Technology tradeoffs

	ReRAM	PCM	STT-MRAM	SRAM	DRAM	Flash (NAND)	HDD
Cell size (F^2)	<4	4–16	20–60	140	6–12	1–4	2/3
Energy per bit (pJ)	0.1–3	2–25	0.1–2.5	0.0005	0.05	0.00002	$1–10\times10^9$
Read time (ns)	<10	10–50	10–35	0.1–0.3	10	100,000	$5–8\times10^6$
Write time (ns)	~10	50–500	10–90	0.1–0.3	10	100,000	$5–8\times10^6$
Retention	years	years	years	While voltage is applied	$\ll$ 1s	years	years
Endurance (cycles)	10^{12}	10^9	10^{15}	10^{16}	10^{16}	10^4	10^4

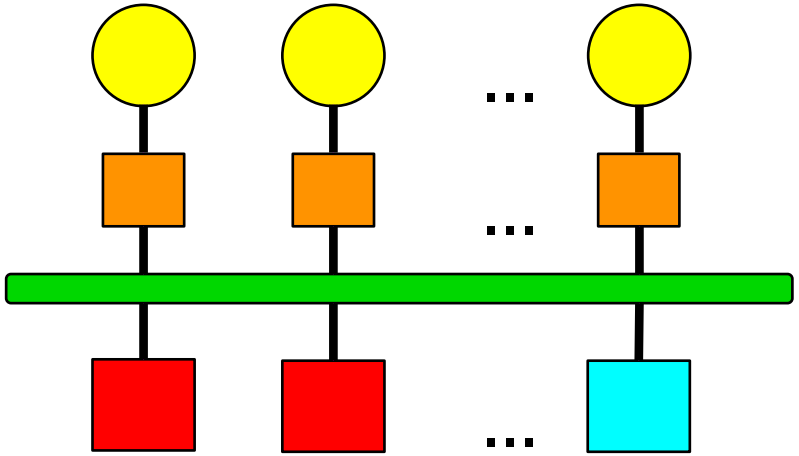
https://www.elinfor.com/knowledge/comparison-between-different-memoriespcm-stt-ram-sram-dram-flash-nand-hdd-p-10908

Where to attach NVM



How to employ NVM

- "Below" DRAM, which then serves as a cache
  - » Intel "memory mode" — transparent to apps
- "Alongside" DRAM, as additional memory
  - » Intel "app direct mode"— explicitly allocated
  - » Needs direct access (DAX) support in the OS
    - – Persistent regions embedded in the file system name space
    - – Opened with special mmap that bypasses the buffer cache — direct access with load and store instructions



How to use DAX NVM

- As a big slow memory
  - ~1/8 the price of DRAM at present: < US $5K/TB
- As a hierarchy level between DRAM and flash
  - Managed by the OS, compiler, or runtime
- As a tool to optimize file systems and databases
  - With or without SSD/HDD below it

★ As explicitly persistent memory — a new abstraction

  - Needs to be kept consistent on a crash

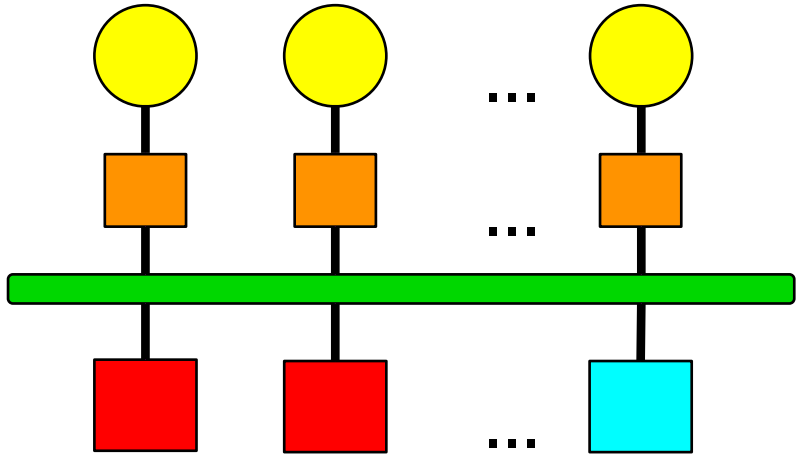
Class outline

- NVM technology and hardware architecture
- Challenges of using NVM
- Correctness criteria for persistent “in memory” data

- Persistent data structures
- Persistent run-time systems
- Persistent memory allocation
- Other issues

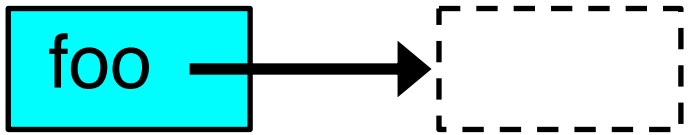
(1) The writeback problem

- Consistency is provided by the cache coherence protocol
- NVM lives below the cache
- Caches write back lines in arbitrary order — typically *not* in program order
- Unless we do something special, memory may be inconsistent in the wake of a system crash



A simple example

```
p = new Node();
foo.next = p;
// cache writes foo.next back to memory, but not *p; system crashes
```



- If B depends on A, you generally need to persist A (write it back and fence it) before even performing B

Forcing persistence on Intel machines

- clflush — evicts line from cache system and waits for persistence
- clflushopt — evicts line but doesn't wait
- clwb — writes line back and doesn't wait
  - also evicts on current machines — alias for clflushopt
- sfence — waits for previous writes-back issued by this HW thread

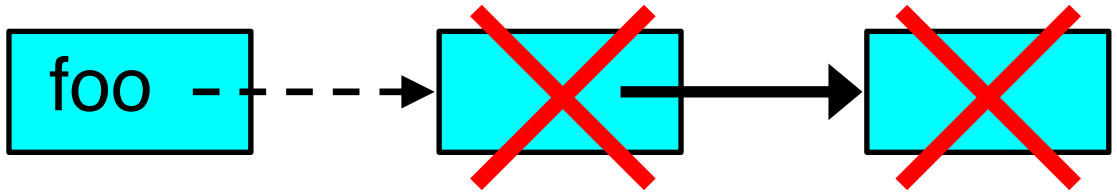
- These aren't cheap: ~95ns to flush and/or fence
  - Handshakes with the on-chip memory controller ("ADR domain")
  - For good performance, we need to minimize fences in particular: consecutive flushes will pipeline; flush-fence pairs will not

A slightly more complex example

```
T1: p = new Node();
    persist *p;  // wb & fence
    foo.next = p;
    persist foo.next;
```

```
T2: q = new Node();
    persist *q;  // wb & fence
    foo.next.next = q;
```

// cache writes back everything but foo.next; system crashes; memory leaks



- Writing back your own updates does not suffice: you often have to write back what you *read*, to make sure it persists before your subsequent updates

(2) The failure atomicity problem

- In general, intermediate states of a typical data structure operation are not consistent, even in transient memory
- Each complete operation transforms a structure from one consistent state to another; we depend on forward progress to get us to the end of each operation
- If a crash occurs in the middle, how do we reach consistency?
  - » Cf: the related problem in file systems and databases
  - » Typically solved with some sort of logging (more on this later)

Other issues (glossed over here)

- Read-write asymmetry
  - » Optane reads are about 3X the latency of DRAM; writes are 10X
  - » May want to design software to minimize writes
- Endurance
  - » PCM lasts for ~10^9 writes (Cf. flash for ~10^4, DRAM for ~10^{16})
  - » Wear leveling and write reduction may be needed
- Full-system recovery (resume active processes)
- Security
  - » We encrypt disks to protect against theft
  - » We need to encrypt NVM as well (built-in on Intel machines)

Key questions for this class

- What does it mean for a persistent data structure to be correct?
  - When and what must we write back and fence to ensure consistency and failure atomicity after a crash?
  - What must we do during recovery itself?
  - How do we prove we've done it right?
- How do we maintain correctness while minimizing costs?
  - For specific data structures
  - In general-purpose *systems* or *methodologies*

Class outline

- NVM technology and hardware architecture
- Challenges of using NVM
- Correctness criteria for persistent “in memory” data

- Persistent data structures
- Persistent run-time systems
- Persistent memory allocation
- Other issues

Linearizability [Herlihy & Wing 1987]

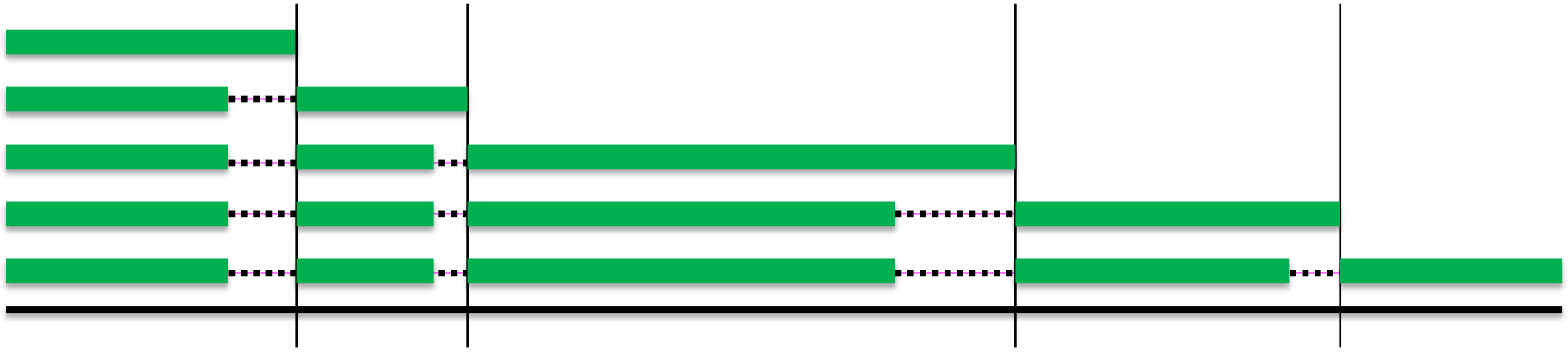
- Standard safety criterion for transient objects
- Concurrent execution H is guaranteed to be equivalent (same invocations and responses, inc. arguments) to some single-threaded execution S that respects
  1. object semantics (*legal*)
  2. program order within each thread
  3. "real-time" order across threads ($res(A) <_H inv(B) \Rightarrow A <_S B$)
- Program is linearizable iff all of its executions are
- 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
- Friedman et al. [PPoPP'18] suggest an equivalent definition: H is durably linearizable iff
  1. It's well formed
  2. Every operation persists before returning
  3. The persist order matches the linearization order
- Note, however, that persisting is generally expensive; doing it before returning slows each operation down a lot

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. Note that buffering may be in hardware or in software.



Ensuring (B)DL

- Recall that the *memory model* determines, for a given architecture, which instructions are guaranteed to occur in order, and which writes may be "seen" by which reads
- x86 machines, for example, have *total store order* (TSO)
  - » Synchronizing instructions (e.g., mfence or CAS [cmpxchg]) are totally ordered wrt each other and wrt previous and subsequent instructions of the current thread
  - » Other instructions obey RW, RR, and WW order, but not necessarily WR
- How does persistence fit into this scheme?

Memory model extensions

- Formalized by Izraelevitz et al. [DISC'16]
  - » release consistency; pwb, pfence, and psync instructions
- Subsumes x86, on which (simplifying a bit)
  - » {store, clflush, clflushopt, clwb} < {sfence, mfence, lock}
  - » {lfence, mfence, lock} < {load}
- A load can see
  1. the most recent store on some backward ordered path,
  2. an *unordered* (racing) store, or
  3. the value persisted before the most recent crash, if there is no ordered intervening write

Incremental mechanical persistence

```
st      → st; pwb
st_rel  → pfence; st_rel; pwb
ld_acq  → ld_acq; pwb; pfence          // this is the tricky one
cas     → pfence; cas; pwb; pfence
ld      → ld
```

- If the original code is data race free and linearizable, the transformed code is durably linearizable
- If the original code is nonblocking, the recovery process is null
- But:
  1. Extensions are required for failure atomicity, which enables forward progress in lock-based code
  2. Optimizations are needed to eliminate unnecessary fences

Ensuring failure atomicity

- Undo logging
  - » Log each previous value, so incomplete operations can be rolled back
  - » Requires a fence after every logging operation
- Redo logging
  - » Log new values as you go, so complete but uncommitted ops can roll forward
  - » Requires special care to read one's own writes
- JustDo logging
  - » *Execute* operations to completion during recovery
- Periodic persistence (*buffered* DL)
  - » Checkpoint now and then (and work on a separate copy), or
  - » Preserve history and arrange for recovery to ignore the recent updates

Reducing the number of fences

- Writes-back aren't expensive (they pipeline): waiting for them is
  - Want to perform multiple writes between fences
- Possible strategies
  - Prefer redo over undo logging
  - When initializing an object, write back *all* lines before fencing
  - Avoid persisting metadata that can be reconstructed during recovery
  - Consider *buffered* DL structures and systems, rather than strictly DL
- Each strategy requires a correctness proof

Linearization points (review)

- Every operation “appears to happen” at some individual instruction, somewhere between its call and return
- Proofs commonly leverage this formulation
  - In lock-based code, L.P. 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 (new)

- (Sufficient but not mandatory) proof-writing strategy
- Implementation is durably linearizable if
  1. somewhere between linearization point and response, all stores needed to “capture” the operation have been written back and fenced
  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
- As in the mechanical transform, nonblocking persistent objects need helping: if an op has linearized but not yet persisted, its successor must be prepared to push it through to persistence

Class outline

- NVM technology and hardware architecture
- Challenges of using NVM
- Correctness criteria for persistent “in memory” data

- Persistent data structures
- Persistent run-time systems
- Persistent memory allocation
- Other issues

Individual data structures

- Lots of trees
  - CDDS B-tree [Venkataraman FAST 2011]
  - wB+-tree [Chen VLDB 2015]
  - NV-Tree [Yang FAST 2015]
  - FPTree [Oukid SIGMOD 2016]
  - WORT [Lee FAST 2017]
  - clfB-tree [Kim TOS 2018]
  - F&F B+-tree [Hwang FAST 2018]
  - NV RB-tree [Wang TOS 2018]
  - RNTree [Liu ICPP 2019]
  - B3-tree [Nam TOS 2020]
- Hash tables
  - NVC-hashmap [Schwalb IMDM 2015]
  - Dalí [Nawab DISC 2017]
- Queue [Friedman PPoPP 2018]

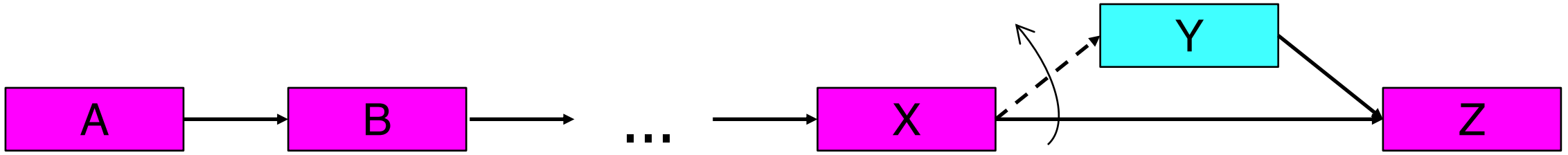
This just scratches the surface

Key ideas (1)

- Use nonblocking structures to get failure atomicity without logs
- Use hardware transactional memory to reduce reliance on locks
  - » In particular, to update an entire cache line atomically
- Track pointers that may not have persisted [David ATC 2018]
  - » *Link and persist* — mark each newly created pointer; persist it before using it to create anything else that might need to be persisted
  - » *Link caching* — add non-persistent pointers to a set; when one needs to be persisted, write them all back and issue a single fence

Key ideas (2)

- Don't persist metadata that is easily reconstructed
  - » Tradeoff between recovery time and overhead during crash-free operation
- Don't persist things that are read during a (possibly lengthy) *traversal* phase [Friedman PLDI 2020]
  - » Validate (and persist) final location; then perform operation
  - » Consider persistent version of Harris & Michael lock-free sorted list:

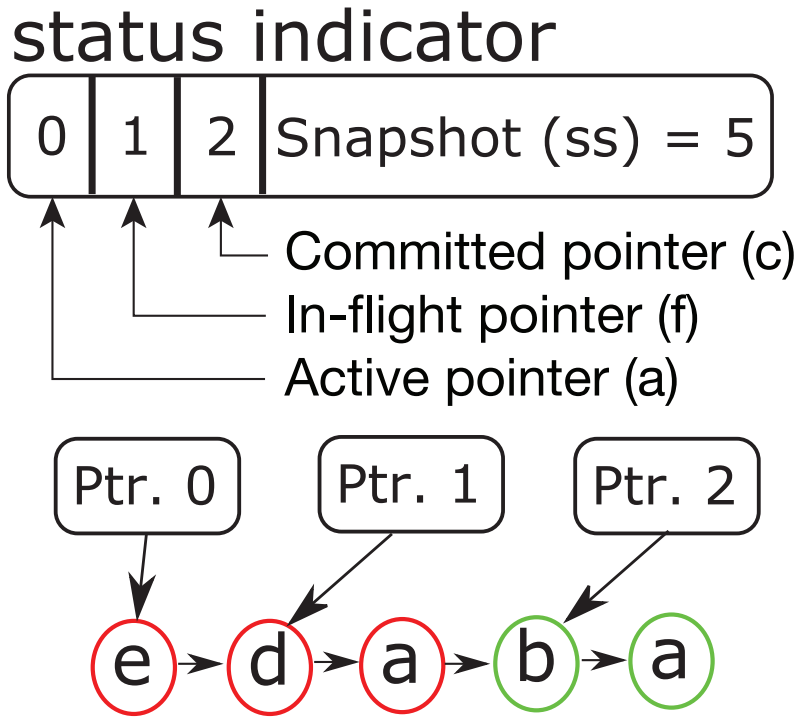


Key ideas (3)

- Leverage “persistent” (history-preserving) structures from functional programming
  - Dalí hashmap [Nawab DISC 2017]
    - “Periodic persistence” for buffered durable linearizability
    - Periodically flush accumulated changes (or maybe everything) — notion of an *epoch*
    - Design structure so recovery can ignore everything changed in recent epochs (tricky!)
  - MOD library [Haria ASPLOS 2020]
    - Create & write back new nodes; fence; install update w/ a single CAS; write-back & fence

Dalí hashmap [Nawab DISC 2017]

- Prepend-only buckets (with occasional garbage collection)
- Updating thread
  - locks bucket
  - identifies old and new epoch number, and old and new pointer roles
  - creates new prepended record
  - retargets new head to new node
  - updates status indicator, atomically, to “rotate” pointers (if necessary) and update snapshot number
  - unlocks bucket



Class outline

- NVM technology and hardware architecture
- Challenges of using NVM
- Correctness criteria for persistent “in memory” data

- Persistent data structures
- Persistent run-time systems
- Persistent memory allocation
- Other issues

Locking — the Atlas system
[Chakrabarti OOPSLA 2014]

- Lock-based failure-atomic sections (FASEs)
- Undo logging
- Compiler support to instrument FASE boundaries & accesses
- Programmer labels persistent data; uses special malloc+free
- Non-isolated critical sections and “naked” accesses mean a completed FASE may see an incomplete FASE
  - System tracks dependences to embed happens-before in log
  - Recovery process replays undo log in reverse order to undo both incomplete and inconsistent FASEs
- Program must be data-race-free, and must support GC

JUSTDO Logging [Izraelevitz ASPLOS'16]

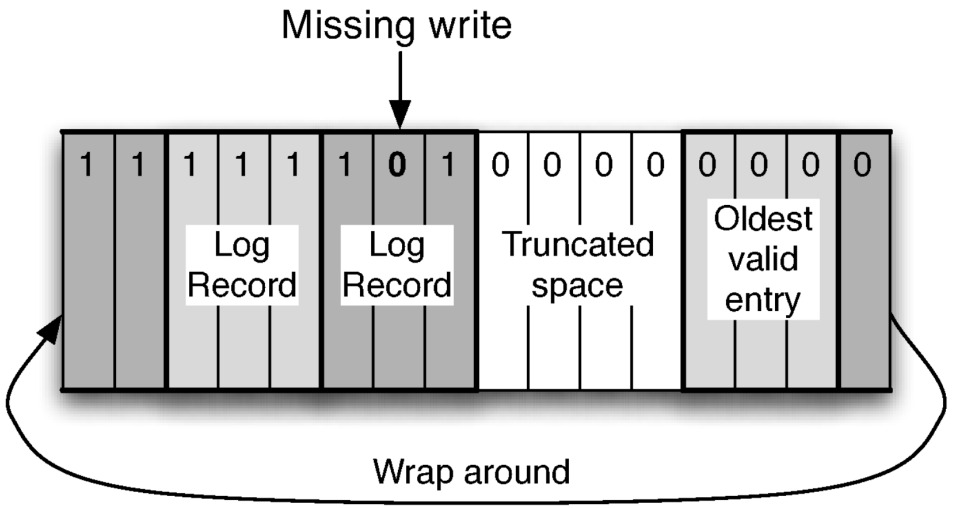
- Designed for a machine with *nonvolatile caches*
- Goal is to assure the atomicity of lock-based FASEs (cf. Atlas)
- Prior to every write, log (to cache) the PC and the live registers
  - » Note that log is small and bounded, and requires no cleanup
- In the wake of a crash, *execute* the remainder of any interrupted FASE
- Less than an order of magnitude slowdown for FASEs (3x faster than Atlas), but not w/ volatile caches (2 orders of magnitude)

iDO Logging [Liu MICRO'19]

- 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
- Makes JUSTDO practical even with volatile caches

Transactions — Mnemosyne
[Volos ASPLOS 2011]

- Based on TinySTM [Felber PPoPP 2008]
  - » Speculative concurrency during crash-free operation
- Uses Intel's STM compiler to instrument code
- Per-thread redo logs, with global timestamps for recovery ordering
- Novel "torn bit" in each word of log so recovery can tell which entries are complete (i.e., valid)



Other transaction systems

- NV-Heaps [Coburn ASPLOS 2011]
  - Object-based; undo logs
- REWIND [Chatzistergiou VLDB 2015]
  - Write-ahead (redo + undo) logging
- SoftWrAP [Giles MSST 2015]
  - DRAM as shadow memory; NVM writes in background
- Romulus [Correia SPAA 2018]
  - 2 copies of everything in NVM; only 4 fences per transaction
- OO Recovery [Cohen OOPSLA 2018]
  - Language-based; *reconstructors* can do arbitrary work at recovery time
- Breeze [Memaripour ICCD 2018]
  - Emphasis on legacy SW; LLVM-based undo logging
- OneFile [Ramalhete DSN 2019]
  - Non-scalable, but wait-free; 2x space overhead
- Pisces [Gu ATC 2019]
  - Object-based; snapshot isolation for fast reads; 2 copies of everything
- QSTM [Beadle 2020 submission]
  - Lock-free; less intrusive than OneFile

And there are more!

Crafty [Genc PLDI 2020]

- First persistence system to leverage full generality of hardware TM
- Combines undo and redo logging
  - » Run a transaction in HTM, building an undo log
  - » Before committing, traverse the undo log, undo the “real” writes, build a (transient) redo log, and record a Lamport timestamp in the undo log
  - » After committing, persist the undo log and then execute the redo log (perhaps with HTM)
  - » If the redo fails due to intervening action in another thread (detected via timestamp), fall back and retry using a global lock (also fall back if HTM fails)
  - » Persist the modified data, then mark the undo log as completed

Other general-purpose models

- Failure-atomic msync [Kelly Queue 2019]
  - » DRAM as shadow copy; all updates persist atomically on sync (and not before!)
- Pronto [Memaripour ASPLOS 2020]
  - » Periodic checkpointing + *high level* logging
- Montage [Wen 2020 (work in progress)]
  - » Buffered durable linearizability; less than one fence per operation
  - » Builds on the Ralloc allocator (more on this in a minute)

Intel PMDK

- Default starting point for most programmers
- Suite of tools from which to pick and choose
  - Log implementation
  - Failure-atomic (but not isolated) transactions
  - Memory allocation
  - Key-value store
  - Testing
  - Remote access via RDMA
  - and more (and yet more under development)

Class outline

- NVM technology and hardware architecture
- Challenges of using NVM
- Correctness criteria for persistent “in memory” data

- Persistent data structures
- Persistent run-time systems
- Persistent memory allocation
- Other issues

malloc / free

- Could simply roll into transactions
  - » Gives up on optimization
  - » Stronger semantics than required (return value of malloc doesn't matter!)
- Preferable for allocator to be a separate abstraction
  - » Introduces problem of pre- and post-attachment leaks:

CRASH →

```
node *t = malloc(size)
t->init(...)
t->next = head

head = t
```

CRASH →

```
node *t = head
head = t->next

free(t)
```

malloc-to / free-from

- Found in nvm_malloc [Schwalb ADMS 2015], PMDK
- Allocate and attach, or detach and deallocate, atomically

```
persistent node *p
malloc-to(size, p)
p->init()
p->next = head
head = p
```

```
p = head
head = head->next
free-from(p)
```

- Non-standard API; hard to retrofit into existing code
- Requires write-back and fence inside each allocator call

GC as an alternative

- Makalu [Bhandari OOPSLA 2016] provides standard API, but does garbage collection after a crash to recover leaked blocks
  - » Still requires write-back and fence in every operation
- Ralloc [Cai ISMM 2020] expands on this idea
  - » Nonblocking (based on LRMalloc [Leite VECPAR 2018])
  - » No write-back or fence required in most operations
  - » Position-independent data
  - » *Filter functions* to assist GC
  - » Exceptionally fast — rivals purely persistent allocators like JEMalloc

Class outline

- NVM technology and hardware architecture
- Challenges of using NVM
- Correctness criteria for persistent “in memory” data

- Persistent data structures
- Persistent run-time systems
- Persistent memory allocation
- Other issues

Testing

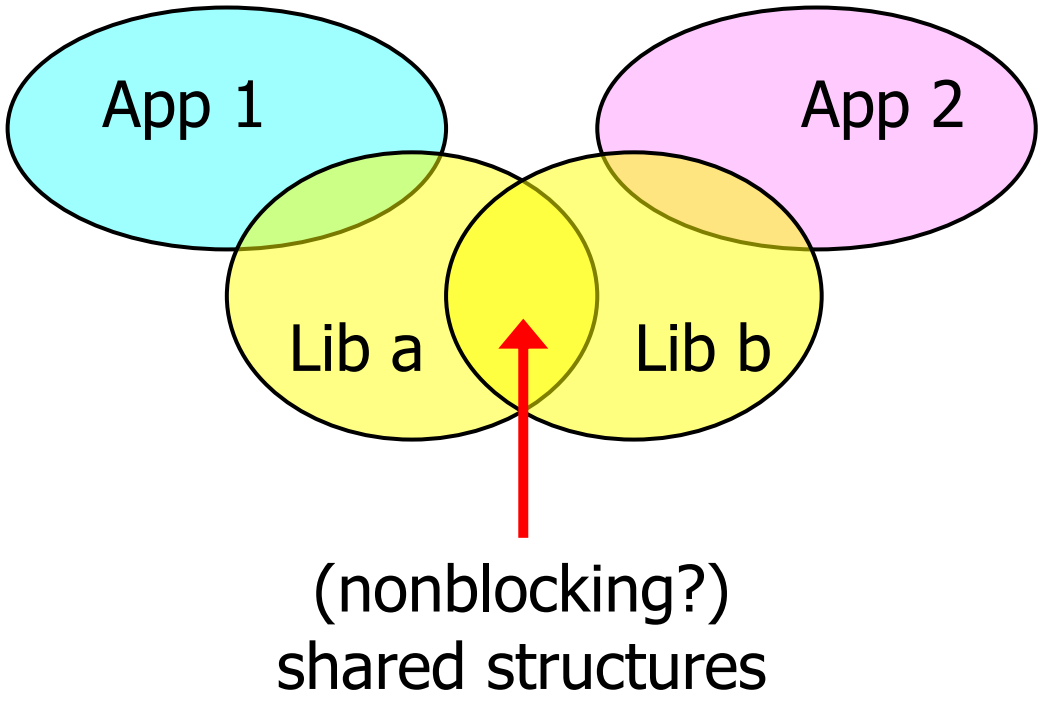
- Real crashes are slow, and hard on real machines
- Several tools catch problematic idioms (e.g., write not followed by persist, or read & use without persist in-between)
- PMDK will enumerate possible write-back interleavings and apply a user-provided validation function to every memory image
  - » Exceptionally slow
- We're developing a tool (PMAT) that intelligently samples from the space of memory images

Other research topics

- Data structures that minimize writes
  - For both performance and endurance
- (Non-persistent) systems that choose what to put in NVM
- File systems and databases with a traditional API, but optimized for implementation on NVM
- Data center architectures with *disaggregated* memory
  - Can you satisfy a cache miss in hardware across the data center network?
- Management of persistent segments
- . . .

Cross-application sharing

- Hodor project [Hedayati ATC 2019] shows how to
  1. Create fast protected libraries that can access memory not visible to the main application
  2. Use these libraries to safely share state between applications
- But threads of different applications can fail independently
  - » Ideal use case for nonblocking structures as originally envisioned by members of the theory community
- See tomorrow's talk at Hydra!



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

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