Database Systems Meet Non-Volatile Memory

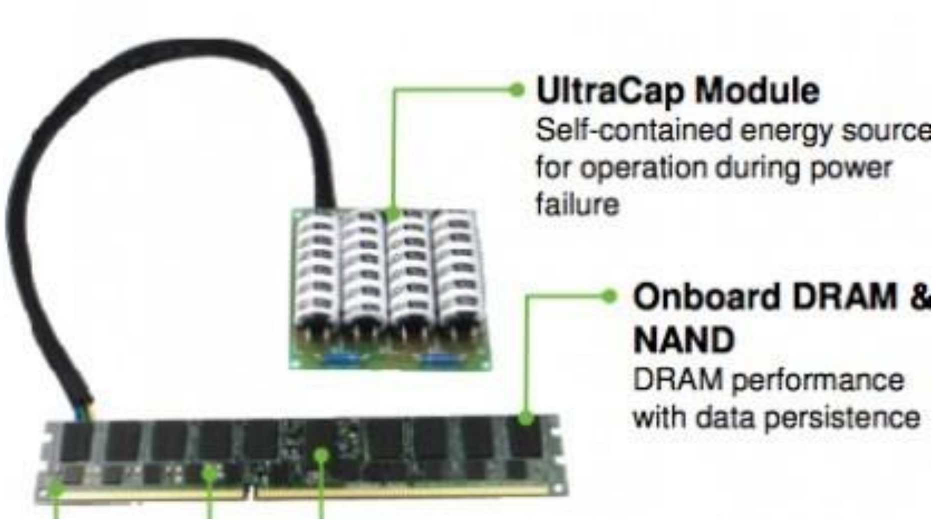
PER-AKE (PAUL) LARSON

Agenda

- NVRAM characteristics and types
- Application access to NVRAM
- NVRAM programming challenges
- And a couple of solutions
- Speeding up logging and replication with NVRAM
- Storing the database in NVRAM

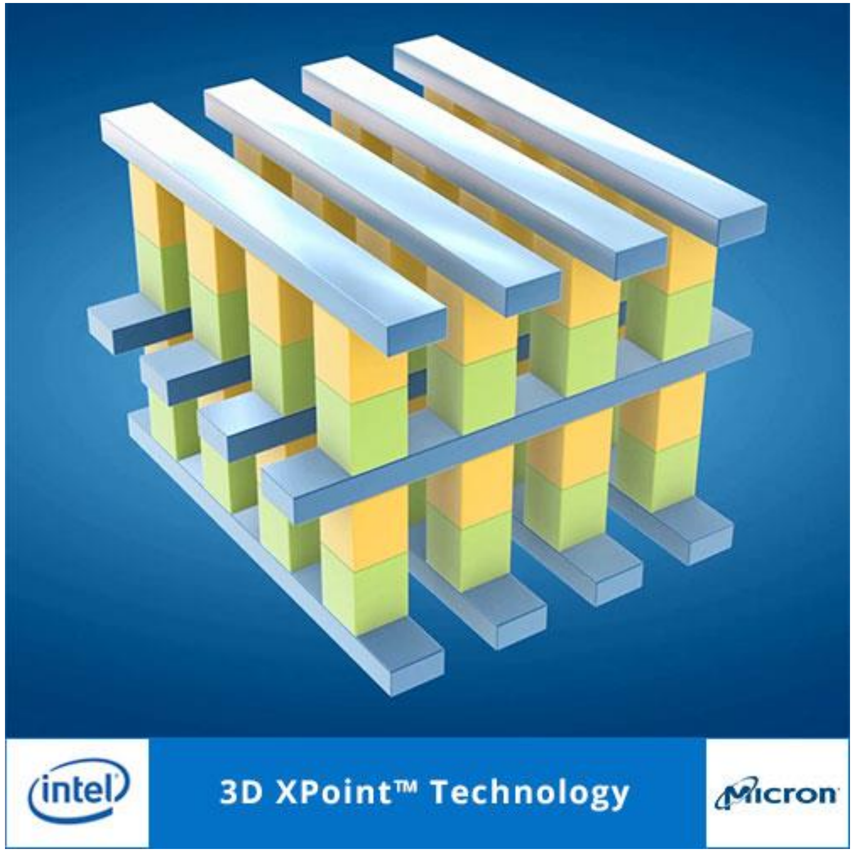
NVDIMM (a.k.a. NVDIMM-N)

- DRAM + flash + power source
- DRAM content
  - saved to flash on power failure
  - restored on power up
- 16 GB DIMMS available now
- ++ normal DRAM speed
- -- reduced memory capacity
  - (smaller DIMMs, space for supercaps)
- -- more expensive than DRAM



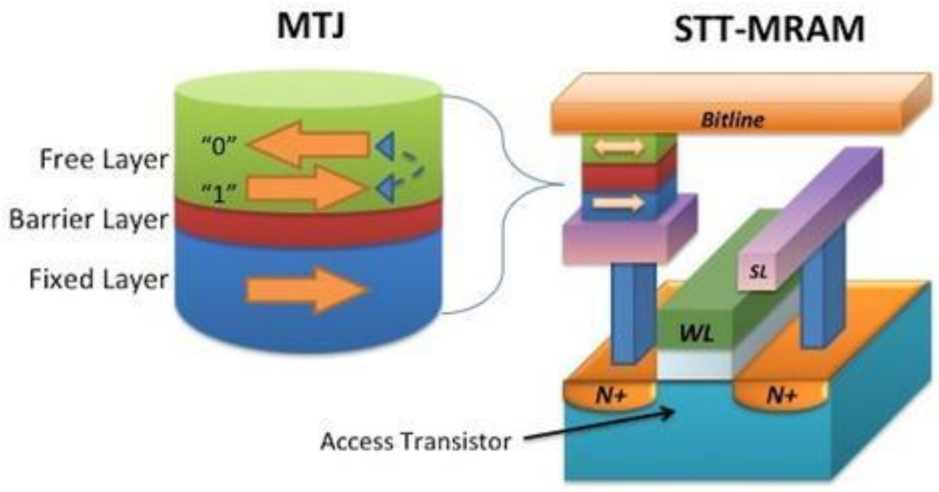
3D XPoint Memory

- Joint development by Intel and Micron
- Announced publicly July 2015
- Physical storage mechanism has not been disclosed
- ++High density, stackable (2 layers initially)
- -- Reads 2-3X slower than DRAM
  - Mitigated by caches – net effect unclear
- Used in Intel's Optane SSDs (end of 2016)
  - 7X more IOPS, 5-10X lower latency than flash-based SSD
- DIMMs to be released in 2017



STT-MRAM (Spin Torque Transfer – Magnetic RAM)

- Resistance depends on polarization of free layer
- Switched by passing a polarized current through the MTJ “layer cake”
- ++ Very fast (SRAM speed), unlimited endurance
- -- Low density (currently)
- 256 Mb DIMMs available now (DRAM at 64 GB DIMMs)
  - Specialty applications for now: satellites, automotive, disk controllers, embedded systems, …
- Ideal NVRAM if they could just up the density…



Agenda

- NVRAM characteristics and types
- Application access to NVRAM
- NVRAM programming challenges
- And a couple of solutions
- Speeding up logging and replication with NVRAM
- Storing the database in NVRAM

Application access to NVRAM

- The physical NVRAM space is modeled and managed much like a disk
  - Divide up space into volumes (partitions)
  - Format volume as either
    - Block addressable – accessed through file read and write commands
    - **Byte addressable (DAX) – accessed by processor load and store instructions**
  - Create files on the volume
- To access data in a DAX file, an application
  - Memory maps the file into its address space
  - Accesses it in the same way as DRAM
- Changes to DAX memory mapped files are persisted immediately
  - Persisting changes to disk-based memory mapped files require a file flush
- Same conceptual model on Linux and Windows

Performance comparison

Test: copying 4K blocks to a file in NVDIMM, single threaded, Windows Server 2016

	IOPS	MB/sec	Latency (ns)
Fast NVMe SSD	14,553	57	66,532
Block-mode NVDIMM	148,553	580	6,418
Memory-mapped NVDIMM	1,112,007	4,344	828

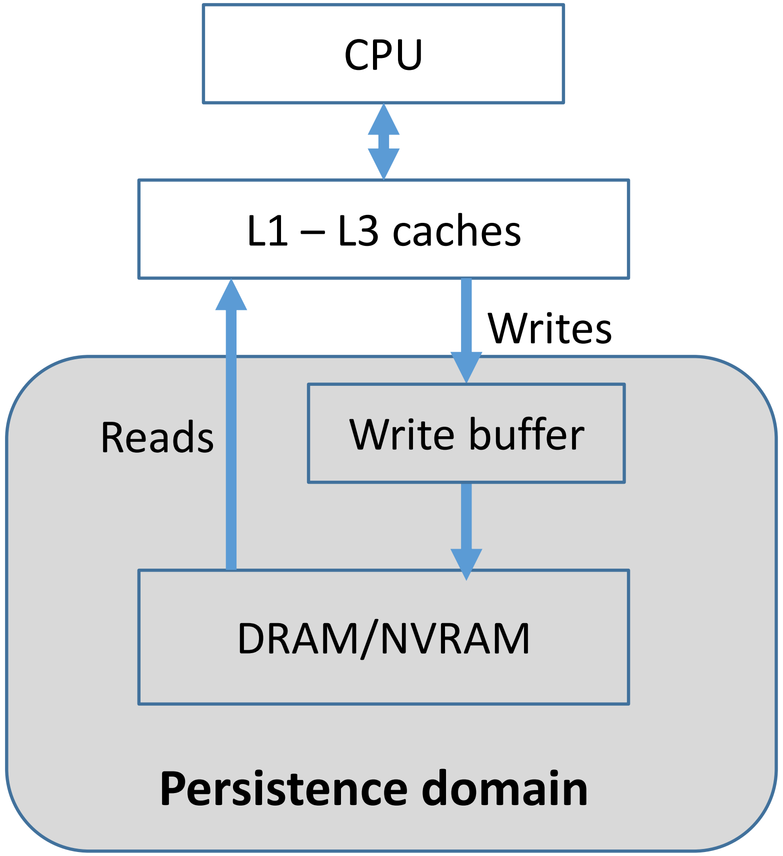
Going through the IO stack slows down block mode by 7.5X

Agenda

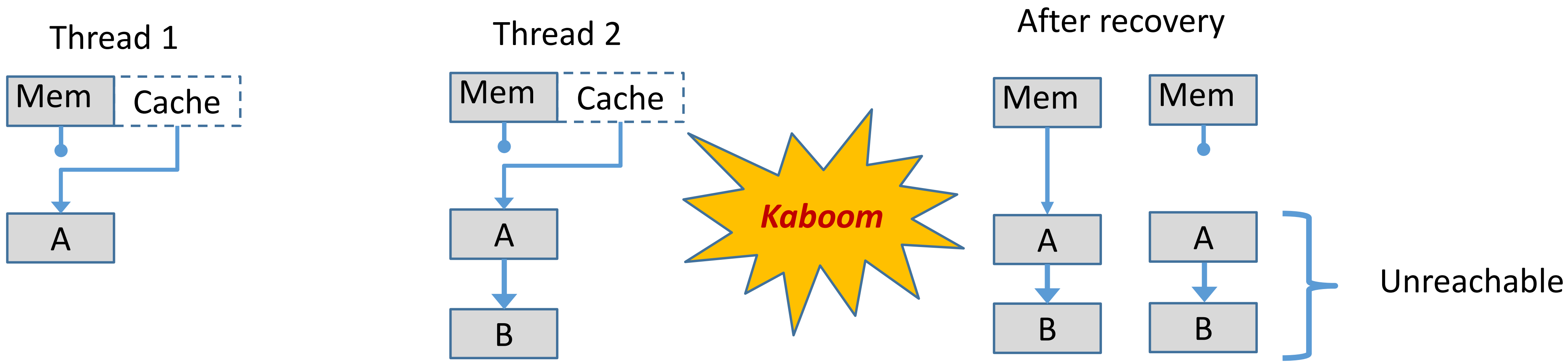
- NVRAM characteristics and types
- Application access to NVRAM
- NVRAM programming challenges
- And a couple of solutions
- Speeding up logging and replication with NVRAM
- Storing the database in NVRAM

Persistence isn't automatic…

- A write only modifies the target in the CPU cache
- Making it persistent requires flushing the cache line
  - Copies line to the memory controller's write buffer
- Code sequence for persisting data
  1. MOV R1, X1
  2. CLWB X1 or CLFLUSH X1
  3. sfence
- CLFLUSH also evicts the line from cache
  - Slows down subsequent access
- CLWB (CL write back) does not evict the line
  - New instruction – big improvement
- Note: cache subsystem can evict a line at any time
  - We don't have full control over when a data item is persisted



And it isn’t atomic!



Correct state

Actual state

Thread 1 inserts A but blocks before persisting the pointer

Thread 2 appends B, persists B and the pointer to it

Must prevent other threads from reading a non-persisted value!

But wait – there's more!!!

- Leaked memory is gone forever
  - Ownership of a persistent memory block must be clear at all times
  - Transfer of ownership needs to be atomic
  - Wear a safety harness whenever possible
    - Being able to determine which blocks are free/in use by scanning a DAX file and/or a allocator's arena
    - Being able to rebuild redundant data structures (indexes, …) from some base data
- Crashes happen – fast recovery a must
  - Recovery is the final defense – this code just has to work
  - And it has to be fast too – it's one of the main selling points of NVRAM
  - Don't ignore cascading crashes (crashes during recovery)

Agenda

- NVRAM characteristics and types
- Application access to NVRAM
- NVRAM programming challenges
- And a couple of solutions
- Speeding up logging and replication with NVRAM
- Storing the database in NVRAM

How to prevent premature reading

- Approach intended for pointers and status-type fields (max 64 bit wide)
- Reserve one bit as a NeedsPersisting flag in each word
- Updates of the word always has the flag set
- Any thread that sees the flag set, resets the flag and flushes the word

```
Int64 ReadPersistentField( Addr)
Begin
    val = *Addr
    while( NeedsPersisting(val) )
        rval = CAS(Addr, val,  ClearNeedsPersistingBit(val) ;
         if( rval == val ) Persist(Addr) ; exitloop; endif
         val = *Addr ;
     endloop
   return val ;
end
```

Persistent multi-word CAS (PMwCAS)

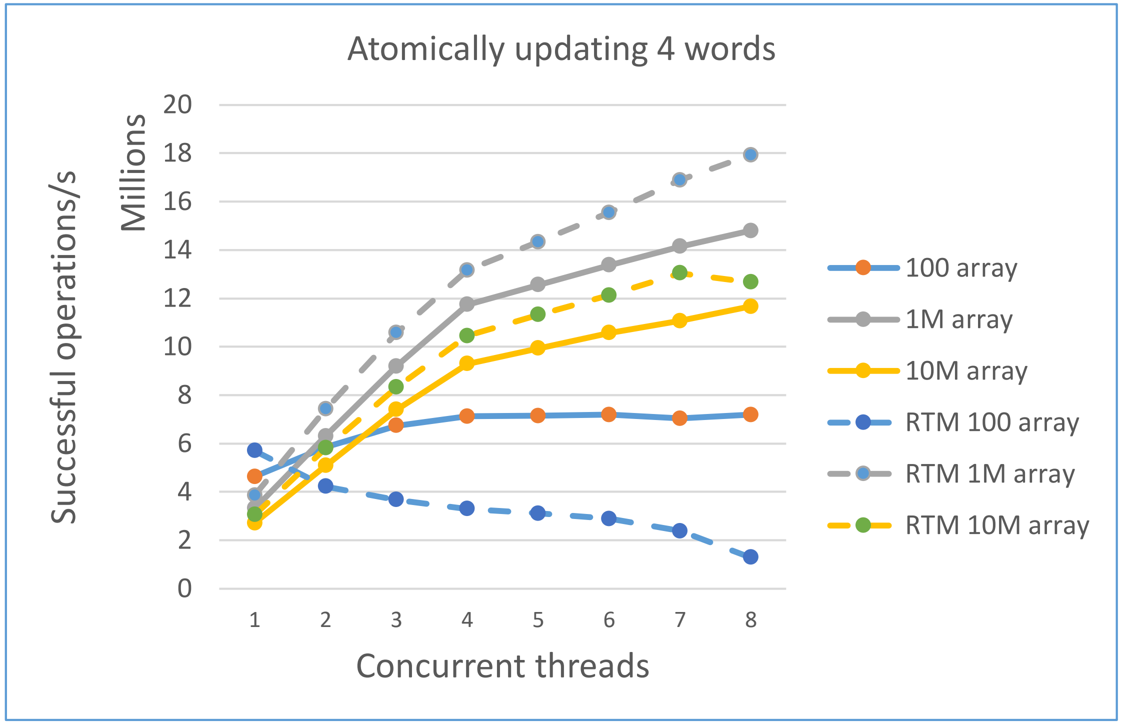
- Need *lock-free* data structures also in NVRAM
  - Doubly linked list, memory allocators, hash tables, B-trees, …
- Lock-free data structures are difficult to implement and potentially slow in NVRAM
- Our approach: implement an efficient lock-free and persistent multi-word CAS operation
  - **Atomically** modifies and persists multiple 64-bit words in NVRAM – basically an ACID transaction
- Based on algorithms by Harris, Fraser and Pratt et al from 2002
  - Our contribution: efficient implementation and persistence
- Classical two-phase algorithm using a descriptor
  - Descriptor specifies what the operation is to do and its current state
  - Non-blocking – threads help each other complete an operation

Harris, Timothy L., Fraser, Keir, Pratt, Ian A., A Practical Multi-word Compare-and-Swap Operation", DISC 2002, 265-279

Algorithm from 30,000 feet

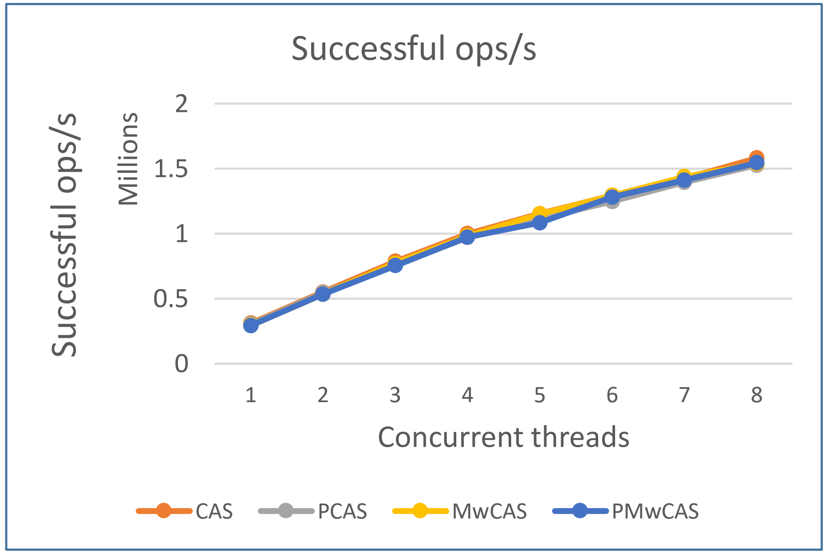
1. Persist descriptor
2. Phase 1:
3. Attempt to swap a pointer to the MwCAS descriptor into every target word but
4. only as long as the descriptor status = UNDECIDED
5.
6. Persist all modified cache lines
7. If all pointer swaps succeeded, set descriptor status to SUCCEEDED else to FAILED
8. Persist status field
9.
10. Phase 2:
11. if status = SUCCEEDED then swap the new value into every target word
12. if status = FAILED then attempt to swap in the old value into every target word
    (Fails if the word no longer contains the original old value but that's OK.)
13. Persist all modified cache lines
14. Set descriptor status to FINISHED
15. Persist status field

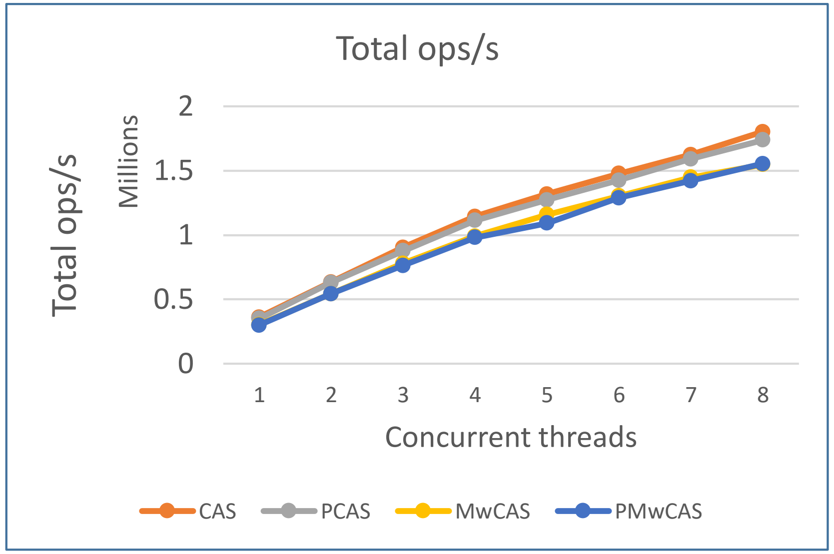
How fast is it?



- Micro-benchmark comparing MwCAS against Intel's HTM implementation (RTM)
- Without persistence because RTM does not guarantee persistence
- Threads atomically update 4 randomly chosen words in an array
- Throughput depends on contention
  - MwCAS wins big under high contention
    - Helping helps!
  - RTM 10-15% faster under low contention

Performance on skip lists





Lock-free skip list (using CAS) vs skip list using MwCAS; doubly linked

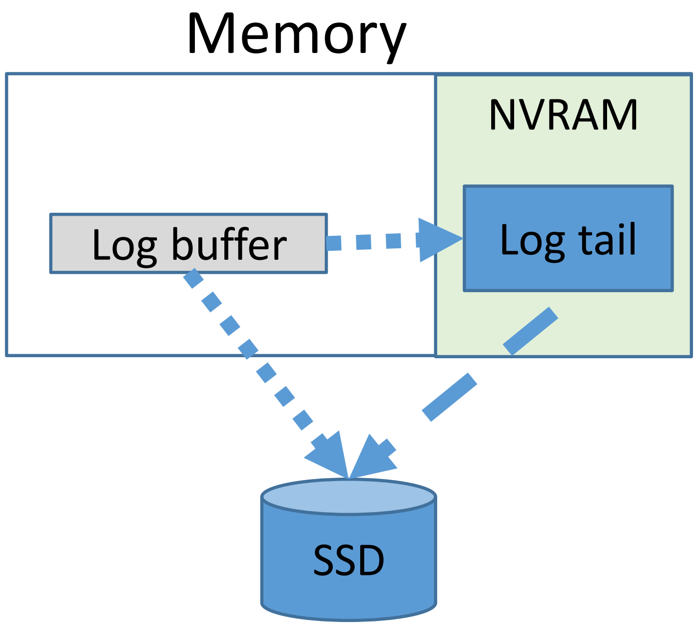
- Initial size 100K, equal proportion of insert, delete, lookup, scan, reverse scan
- Max scan length 100 items

Agenda

- NVRAM characteristics and types
- Application access to NVRAM
- NVRAM programming challenges
- And a couple of solutions
- Speeding up logging and replication with NVRAM
- Storing the database in NVRAM

Speeding up logging with NVRAM

- Need group commit even with SSDs
  - Trading commit latency for higher throughput
- Solution: keep the tail of the log in NVRAM
  - Write log records to NVRAM buffers and commit to NVRAM
  - Flush NVRAM log buffers to storage in large chunks
  - A few tens of MBs is sufficient
- Benefits: fast commit (10-50 microsec), higher log throughput
- Having the tail of the log in NVRAM is sufficient – the whole log is overkill



High Performance Network Characteristics

Remote Direct Memory Access (RDMA)

- Once expensive high-bandwidth network only used in high-performance computing
- Currently becoming cost-competitive
- Bandwidth/latency characteristics improving
- Four dual-port FDR 4x NICs provide roughly the same aggregate bandwidth as DDR3-1600 four-way memory channel
- Kernel and CPU bypass: read and write remote memory directly

Infiniband Type	Latency (us)	Throughput (GB/s)
SDR (2003)	5	1
DDR (2005)	2.5	2
QDR (2007)	1.3	4
FDR (2011)	0.7	6.8
EDR (2014)	0.5	12.1

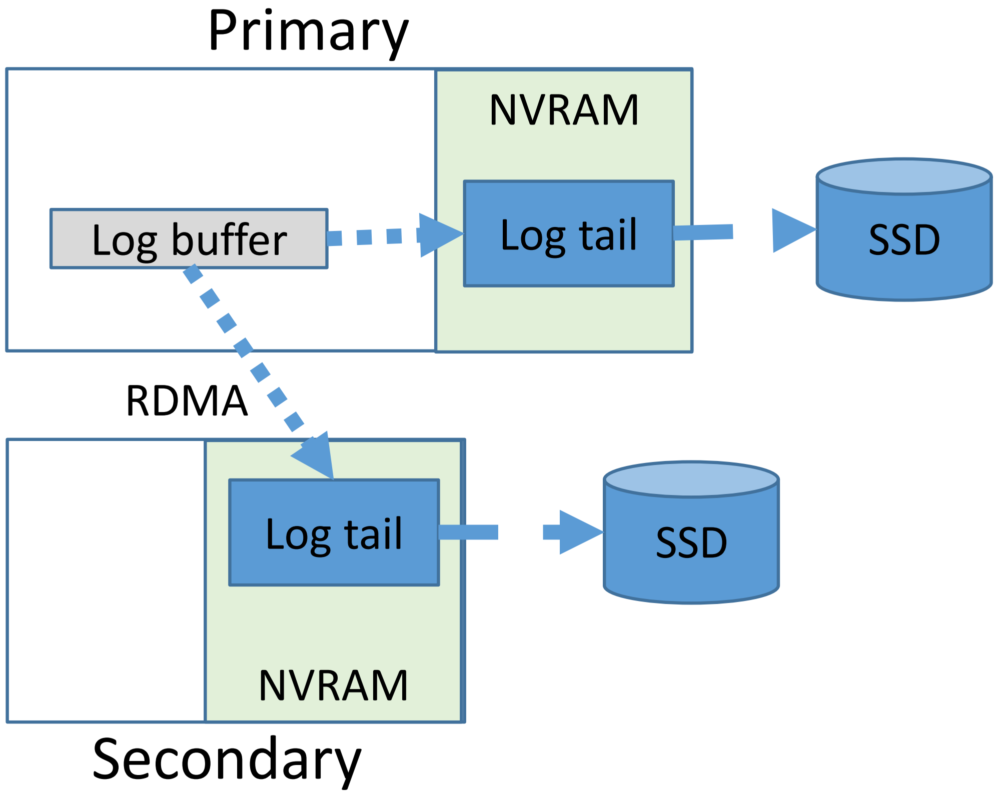
Data Direct I/O

- DMA execution places data directly into CPU L3 cache (if target address is cache-resident)
- Problematic if target address in NVRAM

High-Speed Query Processing over High-Speed Networks
PVLDB 9(4), pp. 228-239, 2015

Speeding up HA (synchronous replication)

- Today: commit a transaction when all synchronous replicas have written its log records to their disks
  - Painfully slow
- NVRAM to the rescue
  - Write log records locally to NVRAM
  - Write them also to NVRAM buffers of all synchronous replicas using RDMA
  - Commit transaction
- No need to wait for disk writes to complete!
- Commit latency of 10-30 microsec possible
  - No more group commit!



Agenda

- NVRAM characteristics and types
- Application access to NVRAM
- NVRAM programming challenges
- And a couple of solutions
- Speeding up logging and replication with NVRAM
- Storing the database in NVRAM

Hekaton in a nutshell

- Main-memory database engine integrated into SQL Server
- Engine uses only lock-free (latch-free) data structures
- Multiversioned records – an update always creates a new version
  - Readers no longer conflicts with writers → higher throughput
  - Each record has two timestamps: begin TS and end TS
- Two index types: hash indexes and range indexes (BW-tree)
- Optimistic concurrency control – no locks, no lock manager

Transaction phases

Txn events	Txn phases	
Begin		Get txn start timestamp, set state to Active
	Normal processing	Perform normal processing ◦ remember read set, scan set, and write set
Precommit		Get txn end timestamp, set state to Validating
	Validation	Validate reads and scans If validation OK, write new versions to redo log
Commit		Set state to Committed
	Post-processing	Fix up version timestamps ◦ Begin TS in new versions, end TS in old versions
Terminate		Set state to Terminated
		Remove from transaction map

Design approach

- Goals: no logging and checkpointing, faster recovery, minimally intrusive design

---

- Records are persisted in NVRAM, indexes are not
- Indexes are rebuilt on recovery
  - Index links still embedded in records but recomputed as part of recovery
- Each transaction acquires a log buffer in NVRAM
  - Short lived: acquired before commit, released when postprocessing is completed
  - Used during recovery to complete postprocessing of committed transactions
  - From a fixed pool of log buffers
  - Stores commit timestamp, pointers to txn's old and new versions plus some additional info
  - Includes a state field: FREE, FILLING, FILLED
- Recovery checks all slots in NVRAM that may contain a record
  - Each table has a separate heap
  - Records stored on "super pages" in fixed-size slots
  - Memory management ensures that we can find all "super pages" owned by the database

Validation phase

1. **Step 1: Validation.**
   1. Validate reads and scans to the extent required by the transaction's isolation level. If validation fails, abort the transaction in the normal way, otherwise continue.
2. **Step 2: Persist database changes and log buffer**
   1. Scan the write set and flush all cache lines modified by the transaction.
   2. Locate a FREE log buffer in the log buffer pool and set its state to FILLING.
   3. Copy the following from the transaction object into the log buffer: transaction ID, commit timestamp, pointers to old versions, and pointers to new versions.
   4. Flush all cache lines covering the log buffer. All changes to the database and the log buffer content are now durable.
3. **Step 3: Commit transaction**
   1. Set the log buffer state to FILLED.
   2. Flush the cache line covering the log buffer state.

Postprocessing phase

1. **Step 1: Finalize timestamps**
    1. Scan the write set and update timestamps of transaction's new and old versions.
    2. Flush all modified cache lines. The timestamp changes are now durable.
2. **Step 2: Free transactions log buffer**
    1. Set the log buffer state to FREE and return it to the log buffer pool.
    2. Flush all cache lines just modified.
3. Terminate the transaction in the normal way.

Database recovery (1/2)

1. **Phase 1: Complete postprocessing of committed transactions.**
    1. For each log buffer lb in state FILLED do.
        1. Scan list of pointers to old versions. For each old version, set the end timestamp to lb's commit timestamp.
        2. Scan the list of pointers to new versions. For each new version, set the begin timestamp to lb's commit timestamp.
        3. Flush all cache lines modified in the previous two steps.
    2. The postprocessing for all transactions that committed before the crash has now been completed and the timestamp changes are durable.
2. **Phase 2: Clean up log buffer pool.**
    1. For each log buffer in state FILLED or FILLING, set its state to FREE.
    2. Flush all cache lines modified in the previous step. All log buffers in the pool are now FREE.

Database recovery (2/2)

1. **Phase 3: Rebuild indexes and free unused record slots**
    1. For each NVRAM page p owned by the database do.
        1. Initialize p's header fields and set it's free list to empty.
        2. For each slot sl on page p do
            1. If sl.BeginTS equals zero, add the slot to p's free list.
            2. if sl.BeginTS contains a transaction ID, the slot contains an uncommitted record so set sl.BeginTS to zero and add the slot to p's free list.
            3. If sl.BeginTS contains a timestamp value, check sl.EndTS
                1. If sl.EndTS equals infinity, it is a current version so determine which table it belongs to and add it to the appropriate indexes.
                2. If sl.EndTS contains a transaction ID, a transaction attempted to delete it but didn't commit so set sl.EndTS to infinity, determine which table it belongs to and add it to the appropriate indexes.
                3. If sl.EndTS contains a timestamp value, the version was deleted by a committed transaction so set sl.BeginTS to zero and add the slot to p's free list.
        3. Flush all cache lines on page p that were modified.
2. End of recovery. Begin normal processing.

Summary

- Small NVRAM is already here and larger ones are coming
  - OS support based on memory-mapped files
- Programming against NVRAM is non-trivial
  - Persistence is neither automatic nor atomic
  - Memory leaks are forever
  - Recovery code required
- Sketched a couple of ways to ensure atomicity
  - Persist-before-read and PMwCAS
- NVRAM can speed commit processing and synchronous replication (HA)
- Explored storing the database in NVRAM