Concurrent Data Structures for Non Volatile Memory

MICHAL FRIEDMAN

Joint work with: Nachshon Cohen, Maurice Herlihy, Virendra Marathe, Erez Petrank, Gali Sheffi, Yoav Zuriel
THIS TALK

- Lock-Free Concurrent Data Structures
- Non-Volatile Byte-Addressable Memory

- Platform & Challenge
- Definitions
- Queue
- Sets
Upon a crash Cache & Memory content is lost
PLATFORM - AFTER

Upon a crash Cache content is lost
**NON-VOLATILE MEMORY**

A type of computer memory that can retrieve stored information even after a power failure.

- NVDIMM’s (super capacitors)
- New technologies: PCM, Memristors (RRAM), STT-RAM(MRAM), etc
- Available on the open market: 3DXPoint, 2019.
# NVM TECHNOLOGIES

<table>
<thead>
<tr>
<th></th>
<th>DRAM</th>
<th>PCM</th>
<th>RRAM</th>
<th>MRAM</th>
<th>SSD</th>
<th>HDD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read latency</td>
<td>60 ns</td>
<td>50 ns</td>
<td>100 ns</td>
<td>20 ns</td>
<td>25 $\mu$s</td>
<td>10 ms</td>
</tr>
<tr>
<td>Write latency</td>
<td>60 ns</td>
<td>150 ns</td>
<td>100 ns</td>
<td>20 ns</td>
<td>300 $\mu$s</td>
<td>10 ms</td>
</tr>
<tr>
<td>Addressability</td>
<td>Byte</td>
<td>Byte</td>
<td>Byte</td>
<td>Byte</td>
<td>Block</td>
<td>Block</td>
</tr>
<tr>
<td>Persistent</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Endurance</td>
<td>$&gt;10^{16}$</td>
<td>$10^{10}$</td>
<td>$10^8$</td>
<td>$10^{15}$</td>
<td>$10^5$</td>
<td>$&gt;10^{16}$</td>
</tr>
</tbody>
</table>
- **NVRAM vs. SSD** - byte addressable, faster, more durable, no erase before rewrite
  - Use original DS
  - No need to use temp DS in main memory
DATA FLOW - EXPLICIT

- Write $x = 1$
- Flush $&x$

```
Cache
  x=1

NVRAM
  x=1
```
DATA FLOW - IMPLICIT

- Write $x = 1$
- Flush &x

![Diagram showing data flow from cache to NVRAM with x=1]
MAJOR PROBLEM: ORDERING NOT MAINTAINED

- Write $x = 1$
- Write $y = 1$
- Flush & $x$
- Flush & $y$

Due to implicit eviction:
Upon a crash, memory may contain $y = 1$ and $x = 0$.

$O_2$ can follow $O_1$, but only $O_2$ is reflected in the memory.
EXAMPLE 1

Suppose everything has been written except for this one pointer.
If a crash occurs, the memory will contain:

Head

Tail

Head

Tail
EXAMPLE 2

If a crash occurs:

Head

Tail

Data

Data

?
**Problem:** Caches and registers are volatile.

- Usually don’t care what’s in the cache/memory
- Here we care! Flush some data to maintain consistency in memory
- Flushing is costly

**Challenge:** make data persistent at minimal cost
HIGH INTEREST

Programming Languages

Parallel
Distributed

Transactions
Allocators
Heaps
Locks

File Systems

Storage

B-trees
Radix trees

Data Structures

Databases

Memory Models
Correctness Conditions
THE MODEL

- Main memory is non-volatile
- Caches and registers are volatile
- All threads crash together
  - New threads are created to continue the execution
OUTLINE

- Definitions
- Lock-free queue
- Lock-free sets
OUTLINE

- Definitions
  - Correctness conditions: [IzraelevitzMendesScott ’16]
  - More conditions: [FHerlihyMarathePetrank ’18]

- Lock-free queue

- Lock-free sets
LINEARIZABILITY

- [HerlihyWing ‘90]

  - Each method call should appear to take effect instantaneously at some moment between its invocation and response.

Thread 1

- `q.enq(5)`

Thread 2

- `q.enq(1)`

```
Thread 1
q.enq(5)
```

```
Thread 2
q.enq(1)
```

```
Thread 1
q.deq()=5
```

```
Thread 2
q.deq()=1
```
CORRECTNESS FOR NVM

Consistent state

1. Buffered Durable Linearizability
   [IzraelevitzMendesScott '16]

2. Durable Linearizability
   [IzraelevitzMendesScott '16]

3. Detectable Execution
   [FHerlihyMarathePetrack '18]

Strength
[IzraelevitzMendesScott ’16]

- Linearizable after removing crashes
  - Operations completed before the crash are recoverable (plus some overlapping operations)
  - Prefix of linearization order
Detectable Execution

- [F. Herlihy, M. Marathe, P. Petrark '18]
  - Durable-linearizability - no ability to determine completion
  - **Detectable execution** extends durable linearizability:
    - Provide a mechanism to check if operation completed
    - Implementation example: a persistent log

Thread 1
- `q.enq(5)`
- `q.deq=(1)`
- `q.deq=(1)`
- `q.deq=(5)`

Thread 2
- `q.enq(1)`
- `q.deq=(5)`
[IzraelevitzMendesScott ’16]

- Some prefix of a linearization ordering
- Support: a “sync” persists all previous operations
OUTLINE

- Definitions
- Lock-free queue
  - Queue [FHerlihyMarathePetrank ’18]
- Lock-free sets
THREE NEW QUEUE DESIGNS

- Three lock-free queues for non-volatile memory [FHerlihyMarathePetrank ’18]

- Based on lock-free queue [MichaelScott ’96]

- Design
- Evaluation
MICHAEL AND SCOTT’S QUEUE (BASELINE)

- A **Lock-Free** queue
- The base algorithm
- A common simple data structure
- Complicated enough to demonstrate the challenges

Since any thread can crash - it must leave the DS consistent.
If writes persisted in the same order they are written, the DS will remain consistent.
Enqueue (data):

1. Allocate a node with its values.
   1.a. Flush node content to memory. (*Initialization* guideline.)
2. Read *tail* and *tail->next* values.
   2.a. Help: Update *tail*.
3. Insert node to queue - CAS last pointer *ptr* point to it.
   3.a. Flush *ptr* to memory. (*Completion* guideline.)
4. Update *tail*.
**DURABLE ENQUEUE - MORE COMPLEX**

- Enqueue (data):
  1. Allocate a node with its values.
     1.a. Flush node content to memory. *(Initialization guideline.)*
  2. Read `tail` and `tail->next` values.
  3. Insert node to queue - *CAS last pointer `ptr` point to it.*
     3.a. Flush `ptr` to memory. *(Completion guideline.)*
  4. Update `tail`.

For example, if this CAS fails due to concurrent activity, we need to be careful to maintain durable linearizability...
Enqueue (data):

1. Allocate a node with its values.
   1.a. Flush node content to memory.
2. Read tail and tail->next values.
3. Insert node to queue - CAS last pointer.
   3.a. Flush pointer to memory.
4. Update tail.

Data

Head

Tail

Tail

Data

Data

Data

Data

volatile

non-volatile

non-volatile

non-volatile

non-volatile
**DURABLE ENQUEUE - MORE COMPLEX**

- **Enqueue (data):**
  1. Allocate a node with its values.
  2. Read \( \text{tail} \) and \( \text{tail->next} \) values.
  3. Insert node to queue - CAS \( \text{ptr} \) point to it.
  4. Update \( \text{tail} \).

- **Complete (and persist) previous operation:**
  5. Flush \( \text{ptr} \) to memory. (Dependence guideline.)
  6. Update \( \text{tail} \).

---

**Head**

```
Data --> Data --> Data
```

**non-volatile**

```
----------- volatile
```

**Head**

```
Data
```

---

**Tail**

```
----------- volatile
```

---
RECOVERY

- Recovery ():
  1. Start at head.
  2. Advance in list until reaching a node $N$ with $N.next.next == NULL$.
  3. Flush next pointer.
  4. Set tail = N.next.

- General - run in every model
GENERAL DESIGN GUIDELINES

- For Durable linearizability:
  - **Initialization** - Flush data before node becomes globally reachable
  - **Dependence** - Persist “previous” operations before persisting current one
  - **Completion** - Persist an operation before it is complete

- For Detectable Execution:
  - **Logging** - Persist a log with description before the operation is executed

- Apply to other data structures
**LOG QUEUE**

- **Properties:**
  - Durable linearizable
  - Detectable execution

- **Implementation:**
  - Use log per thread to register operations

- **Challenges encountered:**
  - Maintaining the logs
  - More complicated operations and dependencies
  - More complicated recovery
RELAXED QUEUE

- **Properties:**
  - Buffered Durable linearizable
  - No flushes until a sync() operation is called

- **Challenges encountered:**
  - Obtain snapshot at sync() time
  - Making sync() concurrent
**SYNC IMPLEMENTATION**

- **Sync ()**: 
  1. Allocate a `NVMStatus` object $S$; Fetch a number from the global counter.
  2. $S.tail \leftarrow \text{tail}$.
  3. Add a special object to block additional enqueues.
  4. $S.head \leftarrow \text{head}$.
  5. Remove the special object to allow enqueues.
  6. Make all the nodes between $S.head$ and $S.tail$ persistent.
  7. Make $S$ persistent.
  8. Update the `status` in the queue with $S$ (if the counter is bigger).
  9. Flush the `status`.

![Diagram of a sync implementation with head, tail, and data nodes connected by arrows. A label indicates `status: head tail sync no. 5 6` and a global counter shows `Counter: 5`.]
MAKING PERSISTENT

1. Head
   - Data
   - Data
   - Data
   - Data

2. Last NVM Head
   - Data
   - Data
   - Data
   - Data
   - Data
   - Data

Head
Tail
RECOVERY

- Recovery ():
  1. Sets the head and the tail to their saved values in S (NVMStatus).
EVALUATION

- Compare the three queues: durable, relaxed, log and Michael and Scott’s queue
- Platform: 4 AMD Opteron(TM) 6376 2.3GHz processors, 64 cores in total, Ubuntu 14.04.
- Workload: threads run enqueue-dequeue pairs concurrently
EVALUATION - THROUGHPUT

Operations/Sec [Millions]

Not persistent

Persistent

Operations/Sec [Millions]

Num of Threads

Implementation details:

- Frequent sync: every 10 ops/thread
- Infrequent sync: every 1000 ops/thread
- Queue initial size: 1 M

Buffered durability less costly

Durability & detectable costly. Similar overhead
A variant of durable linearizability: detectable execution

Three lock-free queues for NVM: Relaxed, Durable, Log

Guidelines

Evaluation

• Durability and detectability - similar overhead
• Buffered durability is less costly
Log-Free Concurrent Data Structures
[David Dragojevic Guerraoui Zabltochi ’18]

- lock-free list
  - follows our guidelines
  - skip-list, hash-table, BST, Memcached
  - durable linearizable
  - no logging
OUTLINE

- Definitions
- Lock-free queue

- Lock-free sets
  - Efficient Lock-Free Durable Sets [ZurielFSheffiCohenPetrank '19] 🏆
Two lock-free sets designs for non-volatile memory
[ZurielFSheffiCohenPetrank '19]

• List
  • Based on [Harris ‘01]
• Hash-table
• BST
• Skip-list
Insert (key, value):

1. Find pred and succ nodes.
2. Allocate a node with its values.
   2.a. Flush node content to memory. (Initialization guideline.)
3. Insert node to list - CAS pred pointer ptr point to it.
   3.a. Flush ptr to memory. (Completion guideline.)
Insert (key, value):

1. Find \textit{pred} and \textit{succ} nodes.
2. Allocate a node with its values.
   
   2.a. Flush node content to memory. (\textit{Initialization} guideline.)
3. Insert node to list - CAS \textit{pred} pointer \textit{ptr} point to it.
   
   3.a. Flush \textit{ptr} to memory. (\textit{Completion} guideline.)
Order can be reconstructed - this is what matters!
validity → not-deleted

Head

Data

98

Data

76

Data

62

Data

40

Data

18

Data

7

Data

8

Data

9

Data

3

Data

5

Data

20

Data

35

Data

TAKE AWAY: DURABLE SETS

- Two lock-free durable sets designs for NVM: Link-Free and SOFT
  - list, hash-table, skip-list, BST
- Important observation: links don’t matter
- Achieve theoretical bound
- Evaluation
  - 3.3x over state-of-the-art with 32 threads
CONCLUSION

- A variant of durable linearizability: **detectable** execution
- **Lock-free** fits the NVM naturally
- **Three lock-free queues** for NVM: Relaxed, Durable, Log
- **Two lock-free durable sets designs** for NVM: Link-Free and SOFT
- Achieve theoretical bound

Thank you!