A Persistent Lock-Free Queue For Non Volatile Memory

Michal Friedman, Technion
Joint work with: Maurice Herlihy, Virendra Marathe, Erez Petrank
POWER OUTAGE
THIS TALK

Concurrent Lock-Free Queue

Deals With Crashes

- Background & Challenges
- Definitions
- Queue
- Evaluation
BACKGROUND
Upon a crash cache & memory content is lost.
Upon a crash, cache & DRAM content is lost.
DATA FLOW - EXPLICIT

- Write $x = 1$
- Flush $\&x$ + Fence
DATA FLOW - IMPLICIT

- Write $x = 1$
- Flush $\&x$ + Fence
Write $x = 1$

Write $y = x$

Flush & $x$ + Fence

Flush & $y$ + Fence

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

Not linearizable!

$O_2$ may depend on $O_1$, but only $O_2$ is reflected in the memory
Each method call should appear to take effect instantaneously at some moment between its invocation and response.
CORRECTNESS FOR NVMM

Consistent State

Buffered Durable Linearizability < Durable Linearizability < Detectable Execution

[IzraelevitzMendesScott '16] [IzraelevitzMendesScott '16] [FHerlihyMarathePetrunk '18]

Strength
DURABLE LINEARIZABILITY

- [IzraelevitzMendesScott '16]
  - Linearizable after removing crashes
  - Operations completed before crash are recoverable (plus some overlapping operations)
Detectable Execution

- [FHerlihyMarathePetrank '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)
```

Thread 2

```
q.enq(1)
```

Crash

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

Time
BUFFERED DURABLE LINEARIZABILITY

- [IzraelevitzMendesScott ’16]
  - Some prefix of a linearization ordering
    - Support: a “sync” persists all previous operations

Thread 1
q.enq(5)

Thread 2
q.enq(1)

Sync

Crash
q.deq()=5
EXAMPLE 1

- Suppose everything has been written except for one pointer.
- If a crash occurs:

Before:

Head

Tail

After:

Head

Tail

?
EXAMPLE 2

Before:

After:

Head

Data

Head

Data

Data

Data

Tail

Tail
**CHALLENGE I**

**Problem I:** Caches and registers are volatile
- Usually don’t care what’s in the cache/memory
- **Here we care!** Persist data to maintain consistency
- Persisting is costly

**Solution:** Reduce number of flushes and fences
CHALLENGE II

Problem II: Data structure in an intermediate state
- Even when writes order is maintained
- Logging
- Updating a copy
- Costly

Solution: Design **lock-free** data structures
LOCK-FREEDOM VS. LOCKS

- At least one process makes progress
- Slow/halted process may not block others
- Since any thread can crash - it must leave the DS consistent
- Even a partial update is considered consistent

Lock-freedom seems like a good fit!
LOCK-FREE QUEUE
THREE NEW QUEUE DESIGNS

- Three lock-free queues for non-volatile memory
  \cite{HerlihyMarathePetrank18}

  - Relaxed
    A prefix of executed operations is recovered \textit{(Buffered)}

  - Durable
    All operations completed before the crash are recovered \textit{(Durable)}

  - Log
    Durable + can tell if an operation recovered \textit{(Detectable)}

- Based on lock-free queue \cite{MichaelScott96}
MICHAEL AND SCOTT’S QUEUE (BASELINE)

• A **Lock-Free** queue

• The base algorithm for the queue in java.util.concurrent

• A common simple data structure, but

• Complicated enough to demonstrate the challenges
• 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.

---

DURABLE ENQUEUE

<table>
<thead>
<tr>
<th>Volatile</th>
<th>Non-Volatile</th>
</tr>
</thead>
</table>
**DURABLE ENQUEUE - MORE COMPLEX**

- 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 *ptr* point to it
     3.a. Flush *ptr* to memory. *(Completion guideline.)*
  4. Update *tail*.

Fails due to concurrent activity, be careful to maintain durable linearizability…

---

Help: Update tail.

Flush *ptr* to memory. *(Completion guideline.)*
**DURABLE ENQUEUE - MORE COMPLEX**

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

  - Help: Update tail.

- Flush node content to memory (Initialization guideline.)
  - Flush `ptr` to memory (Completion guideline.)

- Head  
  - Tail

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data

- Data
**DURABLE ENQUEUE - MORE COMPLEX**

- Enqueue (data):
  1. Allocate a node with its values.
     1.a. Flush node to memory.
  2. Read `tail` and `tail->next` values.
  3. Insert node to queue - CAS last pointer `ptr` point to it.
  4. Update `tail`.
  
- Complete (and persist) previous operation:
  5. Flush `ptr` to memory.
  6. Update `tail`.

---

Head

- Data

---

Data

Data

Data

---

[[volatile]] non-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 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
  
• Apply to other data structures
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 an `NVMStatus` object $S$; Fetch a number from the global counter.
  2. $S\cdot tail \leftarrow tail$.
  3. Add a special object to block additional enqueues.
  4. $S\cdot head \leftarrow head$.
  5. Remove the special object to allow enqueues.
  6. Make all the nodes between $S\cdot head$ and $S\cdot tail$ 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 SYNC implementation with head and tail pointers and data nodes]
MAKING PERSISTENT

1

Head

Tail

Data → Data → Data → Data

2

Last NVM Head

Tail

Data → Data → Data → Data → Data → Data
RECOVERY

- Recovery ():
  1. Sets the **head** and the **tail** to their saved values in **S (NVMSstatus)**.
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
LOG ENQUEUE

- Enqueue (data, threadID, operationNum):
  1. Allocate a node, a log entry and point on each other.
  2. Flush node and log content to memory. (Initialization guideline.)
  3. Update the logs array for the relevant thread and flush. (Logging guideline.)
  4. Read tail and tail->next values.

- Complete (a 1d persist) previous operation:
  5. Flush ptr to memory. (Dependence guideline.)
  6. Update tail.
EVALUATION
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

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
CONCLUSION

- A variant of durable linearizability: detectable execution
- Lock-free fits the NVM naturally
- Three lock-free queues for NVM: Relaxed, Durable, Log
- Guidelines
- Evaluation:
  - Durability and detectability - similar overhead
  - Buffered durability is less costly

Thank you!