Log-free concurrent data structures

Tudor David  (IBM Research, Zurich)
Aleksandar Dragojevic  (Microsoft Research, Cambridge)
Rachid Guerraoui  (EPFL)
Igor Zablotchi  (EPFL)
In-memory data structures – at the core of many systems

Lists

Trees

Hash tables

Skip lists

Interface:
• insert(k,v);
• remove(k);
• find(k);

Data structure performance – essential to system behavior
An ideal data structure:
• fast
• scalable
• durable

NV-RAM – expected to become ubiquitous

How can we ensure performance + consistent durable state?
Upcoming technology: Non-volatile RAM

NV-RAM:
• Durable
• Byte-addressable
• Latencies – comparable with DRAM
• Programming model - map to area of virtual memory

Latencies:

<table>
<thead>
<tr>
<th></th>
<th>L1 cache</th>
<th>L2 cache</th>
<th>L3 cache</th>
<th>DRAM</th>
<th>FeRAM</th>
<th>PCM</th>
<th>MRAM</th>
<th>STT-RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>1 ns</td>
<td>3 ns</td>
<td>10 ns</td>
<td>60 ns</td>
<td>100 ns</td>
<td>70 ns</td>
<td>62 ns</td>
<td>60 ns</td>
</tr>
<tr>
<td>Write</td>
<td>1 ns</td>
<td>3 ns</td>
<td>10 ns</td>
<td>60 ns</td>
<td>125 ns</td>
<td>150 ns</td>
<td>62 ns</td>
<td>60 ns</td>
</tr>
</tbody>
</table>

Source: "nv_malloc: Memory Allocation for NVRAM", Schwalb et al., 2015
Persistent, fast and correct software – not trivial

1: mark memory as allocated
2: **persist allocation**
3: change link of node 1
4: **persist memory content**
5: change link of node 1
6: **persist new link**
7: change link of node 2
8: **persist modified link**
9: done = 1

---

**Write-back cache:**
1: mark allocation
2: initialize mem
3: change link 1
4: change link 2
5: done = 1

---

**NV memory:**
3: change link 1
5: done = 1

---

**Upon restart: incorrect state**

---

Batching write-backs: beneficial for performance
Persistent, fast and correct software – not trivial

Write-back cache:
1: mark allocation
2: initialize mem
3: change link 1
4: change link 2
5: done = 1

Frequent waiting for data to be persisted

NV memory:
1: mark allocation
2: initialize mem
3: change link 1

Upon restart:
undo incomplete operations

1: log[0] = starting transaction X
2: persist log[0]
3: log[1] = allocating a node at address A
4: persist memory content
5: mark memory as allocated
6: persist new link
7: change link of node 1
8: persist modified link
9: log[2] ≠ previous value of link
10: persist log[2]
11: change link 1
12: persist modified link
13: log[3] = previous value of link
14: persist log[3]
15: change link 2
16: persist modified link
17: done = 1
18: persist done
19: mark transaction X as finished

crash

Undo incomplete operations
Persist latency >> cache latency

For performance – minimize time waiting for data to be persisted;

Logging – problematic:
• We need to issue write-back to the log before each update;
• The log entry must be persisted before we perform the update;

Our work: minimize logging and write backs in data structures
Our techniques for fast durable data structures

1. Link-and-persist:
Use lock-free algorithms: they never leave the structure in an inconsistent state
    \[\Rightarrow \text{no logging in the data structure algorithm}\]
Correctness: \textit{link-and-persist}

2. Link cache:
Buffer updates;
When one must be persisted, batch write-backs;

3. NV-epochs: coarse-grain memory management:
Track active memory areas, don’t log individual allocations;
Avoid work at run-time if it can be delayed for recovery time;

\textbf{Log-free concurrent data structures}
Use lock-free algorithms: they never leave the structure in an inconsistent state

⇒ no logging in the data structure algorithm

**Correct durable implementations?**

**Correctness condition: durable linearizability**

After a restart, the structure reflects:
• all operations completed (linearized) before the crash;
• (potentially) some operations that were ongoing when the crash occurred;

1. Persistently allocate and initialize node
2. Add link to new node
3. Persist link to new node

If crash between steps 2 and 3, violation of durable linearizability
Link-and-persist: achieving correctness

1. Persistently allocate and initialize node
2. Add marked link to new node
3. Persist link to new node
4. Remove mark

Other threads - persist marked link if needed

**Link-and-persist**: atomic “modify” and “persist” link
Going further: defer persisting links

A link only needs to be persisted when an operation depends on it

Store all un-persisted links in a fast concurrent cache

When an operation directly depends on a link in the cache:
batch write-backs of all links in the cache (and empty the cache)

Insert(X)

... write-back all links

Read(X)

The link cache: batch write-backs of links
Memory allocation and reclamation

Insertion
1: log alloc_and_link
2: allocate and initialize new node;
3: link into data structure;

Deletion
1: log unlink_and_free
2: unlink node;
3: free the node;

alloc 0xa010
alloc 0xa110
alloc 0xa210

Locality of allocations/deallocations

Instead of logging each allocation, keep track of recently used memory areas
At recovery, traverse nodes in active areas at the moment of the crash/restart
Experimental evaluation

• No fast byte-addressable NV-RAM for now;
• Inject latencies of flushes and write-backs to NV-RAM based on published values;
  • Shown experiments: 125 ns;
• Comparison with redo-log-based implementations
  • Also provide durable linearizability;
Consistently faster than log-based data structures

Particularly suited to small and medium sized data structures
NV-Memcached

1. Lock-free Memcached: Memcached-clht
2. Replace hash table & slab allocator with durable versions

Similar throughput, much lower recovery time
Log-free concurrent data structures

1. **Link-and-persist:**
   Use lock-free algorithms: they never leave the structure in an inconsistent state
   \[\Rightarrow\] no logging in the data structure algorithm

   Correctness: **link-and-persist**

2. **Link cache:**
   Buffer updates;
   When one must be persisted, batch write-backs;

3. **NV-epochs:** coarse-grain memory management:
   Track active memory areas, don’t log individual allocations;
   Avoid work at run-time if it can be delayed for recovery time;

   Up to several times throughput improvement

Thank you!