Fine-Grain Checkpointing with In-Cache-Line Logging

Nachshon Cohen
Hillel Avni

David T. Aksun
James R. Larus

Previously published in Asplos 2019
Background: Non-Volatile Memory (NVM)
Background: NVM Program Life Cycle
Using NVM with Cache

Caches are volatile!

 Writes can be reordered!!!
Building data structures for NVM

B+ tree

put(key: 10, value 12)

B+ node

<table>
<thead>
<tr>
<th>keys</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
</tr>
</tbody>
</table>

B+ node

key:10

<table>
<thead>
<tr>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
</tr>
</tbody>
</table>
Existing approaches

- **Checkpointing**
  - separate location
  - long intervals

- **Logging**
  - log value during execution
  - explicit persist instructions

ex: x86 `clflush/clflushopt` instructions followed by `sfence/mfence`
A novel approach for library builders

**Goal:** Design a durable data structure with **low durability overhead**

- Fine-grained checkpointing (Periodic persistency)


- **In cache line log** (our novel contribution)

  avoid persist instructions in the critical path of the application
Fine-grained Checkpointing

- Flush entire cache hierarchy
  - checkpoint every 64ms (e.g., x86’s `wbinvd` instruction)
- Restore state to beginning of epoch (use log)

overhead: around % 2
Use undo log for restoring state

**put**(key: 10, value 12)

**put**(key, value):
- entry = log(key, old value)
- persist(entry)
- update(key, value)

<table>
<thead>
<tr>
<th>B+ node</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>key:10</td>
<td>3 7 4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B+ node</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>key:10</td>
<td>3 12 4</td>
</tr>
</tbody>
</table>

Can we do better?
**In cache line log (InCLL)**

- **Put** (key: 10, value: 12)

  - **B+ node**
    - Key: 10
    - Values: 3, 7, 4
    - InCLL

- **Put** (key: 10)
  - **B+ node**
    - Key: 10
    - Values: 3, 7, 4
    - InCLL

- **Put** (key, value): log(key, old value)
  - **B+ node**
    - Key: 10
    - Values: 3, 12, 4
    - InCLL

- **Update** (key, value)

**Put the log inside the same cache line as the modified data**
How does InCLL avoid explicit **persist** instructions?

**Case 1:** Cache line is lost

**Case 2:** Cache line is propagated to NVM
InCLL Limitations

- Limited size

\[
\text{put(key: 10, value 12)} \\
\text{put(key: 12, value 6)} \\
\text{put(key: 8, value 2)}
\]
External Undo Log

- Deal with cases where InCLL is not enough
- Log the entire node
  - the node will be modified within the epoch
  - no need to log the same node more than once within an epoch
  - requires explicit `persist instructions` to NVM

**put**(key: 12, value 6)
**put**(key: 8, value 2)
InCLL and External Undo Log - Per Node Per Epoch

First modification: use InCLL

**Effective** when modifications are *sparse*

- large tree
- uniform key distribution

2+ modifications: use external undo log

**Effective** when modifications are *dense*

- node split
- range of keys

Non-volatile

<table>
<thead>
<tr>
<th>node 1</th>
<th>3</th>
<th>12</th>
<th>4</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>node 1K</td>
<td>8</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>node 1M</td>
<td>23</td>
<td></td>
<td></td>
<td>12</td>
</tr>
</tbody>
</table>
**InCLL and External Undo Log - Per Node Per Epoch**

<table>
<thead>
<tr>
<th>Best case</th>
<th>Worst case</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Single popular key</td>
<td>• Updating two values only once</td>
</tr>
<tr>
<td>• Skewed key distribution</td>
<td>• One persist operation per two modifications</td>
</tr>
</tbody>
</table>

Additional details in [Cohen et al., ASPLOS 2019]
Evaluation

- Modified Masstree [Mao et al., EuroSys 2012] internals
  - make Masstree durable (durable API: insert, get, remove, scan)
  - make allocator durable (prevents dangling pointers, durable memory leaks)
- YCSB Workloads (Uniform and Zipfian Distribution)
  - YCSB A (50% insert, 50% get)
  - YCSB B (5% insert, 95% get)
  - YCSB C (100% get)
  - YCSB E (5% insert, 95% scan)
YCSB Workload on Masstree (Persistent vs Volatile)
Effect of Tree Size on Performance

![Graph showing the effect of tree size on performance. The x-axis represents the number of entries, and the y-axis represents throughput (Ops/sec). Different line colors indicate different configurations, with labels for YCSB A MT+, YCSB A INCLL, YCSB A MT Zipfian, and YCSB A INCLL Zipfian.]
Conclusion

Use **In Cache Line Log** to avoid **persist instructions**

- fine-grained checkpointing
- external log

- **durability with small overhead**

Questions?