Durability Semantics for Lock-based Multithreaded Programs

Dhruva R. Chakrabarti, Hans-J. Boehm
Hewlett-Packard Laboratories
Do we need a separate durable data representation?

- Conventional durability techniques
  - Separate object and persistent formats
  - Translation code
  - Programmability and performance issues

- In-memory durability
  - Enabled by non-volatile memory or NVRAM (such as memristors, PCM, etc.)
  - In-memory objects are durable throughout
  - Byte-addressability simplifies programmability
  - Low load/store latencies offer high performance
Programming Model

Use persistent regions (PR) instead of flat files

```
pr = find_or_create_persistent_region(name);
persistent_data = get_root_pointer(pr);
if (persistent_data) {
    // restart code
}
else {
    // initialize persistent_data
}
// use persistent_data
```

Data not in a PR is considered transient
Motivating Observations

• Reuse durable data structures after process termination
• Reusable data structures must be consistent across failures
  – Invariants must be preserved
• How are invariants identified in a lock-based multithreaded program?
  – No explicit association between a shared datum and the protecting lock
  – Lock acquires can be nested

Time

\[ C_1, C_2, C_3, C_4, C_5, C_6, C_7 \]

\[ T_1, T_2 \]
Contributions

• Consistency semantics for durable data at intermediate program points
  – In spite of arbitrary lock nesting
  – Largely unchanged code
  – Relationship with transactional semantics
• Optimizations
• Initial idea of overheads
Notion of Consistent Program Points

**Unlocked program points are thread-consistent**

- Critical sections indicators of consistent states
  - If no locks are held, all data structures should be in a consistent state
- Some restrictions:
  - Client provided locks
  - Serial programs
Notion of Failure-Atomic Update Units

Outermost Critical Sections (OCS) are failure-atomic

\[
\begin{align*}
\text{T2} & : \text{OCS} \\
\text{T1} & : \text{Inner critical sections}
\end{align*}
\]
Notion of Durability-related Dependences among OCSes

A completed OCS may depend on an incomplete OCS

- **Cause:** Isolation and durability boundaries may not match
- **Effect:** The durable effects of a completed OCS may have to be undone
  - Happens only with nesting

  x, y are persistent and initially x=y=0

If the program crashes, can the effects of \( o_1 \) be made durable when those of \( o_2 \) are not?

**NO!** \( y=1, x=0 \) is not a consistent state.
OCS-hb relation may be cyclic

Inner critical sections can cause cyclic OCS-hb

All effects in the involved OCSes must appear to be visible in persistent memory at the same time
An Implementation Overview

• All lock operations and writes to persistent memory locations logged
• hb-relations between lock releases and acquires captured in the log
• Logs maintained in non-volatile memory
• Unnecessary log entries periodically pruned
• Some optimizations implemented
• Cache lines flushed at appropriate points
Some Preliminary Experimental Results

- NVRAM-based programs 2-3 orders of magnitude faster than disk-based ones
- But what’s the overhead of adding durability to transient data structures?

1^Runtime comparison of 2 durable applications with the transient version as the baseline

<table>
<thead>
<tr>
<th>Apps</th>
<th>Slowdown (num_threads=4)</th>
<th>Statistics (num_threads=4)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>nvram</td>
<td>nvram-nf</td>
</tr>
<tr>
<td>Dedup</td>
<td>50%</td>
<td>33%</td>
</tr>
<tr>
<td>Memcached</td>
<td>160%</td>
<td>60%</td>
</tr>
</tbody>
</table>

**Dedup**: A deduplication kernel from the PARSEC benchmark suite. The hashtable maintaining unique key-value pairs of chunks of input stream is made durable.

**Memcached**: Starting with the original key-value cache implementation, the cache, LRU lists, and the slab allocator information are made durable.

---

1^DRAM used to simulate NVRAM on a RedHat Linux Intel Xeon x86-64 machine.
Conclusions

• Presented a technique for identifying intermediate application-wide consistent states in a lock-based program

• NVRAM enables an efficient implementation
Backup
Optimizations/Pitfalls

• Is log elision applicable to durable updates outside an OCS?
• Is it mandatory to track every OCS-hb, specifically ones that involve OCSes with updates to transient locations alone?
Optimizing thread-consistent updates

Elide logging outside an OCS, if possible

T0
a1: lock(l1)
a2: t = ready
a3: unlock(l1)
4: if (t)
5: y = x

T1
b11: lock(l2)
b12: p = q
b13: unlock(l2)
14: x = 1
c15: lock(l1)
c16: ready = 1
c17: unlock(l1)

T2
d21: lock(l3)
d22: lock(l2)
d23: q = 1
d24: unlock(l2)
d25: m = 1
d26: unlock(l3)

OCS level hb-relations

Elide logging of line 14 since OCS ‘b’ will not be undone.
Elide logging of line 5 since OCS ‘a’ will not be undone.
Every hb-relation must be captured

- \(x, y, m, p,\) and \(q\) are shared and persistent.
- \(t\) is local, \(ready\) is shared. Both are transient.

Initially \(x = y = m = p = q = t = ready = 0\)

<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1: (\text{lock}(l1))</td>
<td>b11: (\text{lock}(l2))</td>
<td>d21: (\text{lock}(l3))</td>
</tr>
<tr>
<td>a2: (t = ready)</td>
<td>b12: (p = q)</td>
<td>d22: (\text{lock}(l2))</td>
</tr>
<tr>
<td>a3: (\text{unlock}(l1))</td>
<td>b13: (\text{unlock}(l2))</td>
<td>d23: (q = 1)</td>
</tr>
<tr>
<td>4: if ((t))</td>
<td>14: (x = 1)</td>
<td>d24: (\text{unlock}(l2))</td>
</tr>
<tr>
<td>5: (y = x)</td>
<td>c15: (\text{lock}(l1))</td>
<td>d25: (m = 1)</td>
</tr>
<tr>
<td></td>
<td>c16: (\text{ready} = 1)</td>
<td>d26: (\ldots)</td>
</tr>
<tr>
<td></td>
<td>c17: (\text{unlock}(l1))</td>
<td></td>
</tr>
</tbody>
</table>

OCS level hb-relations

- If OCS d fails but all hb-relations are captured, all values are reset to a consistent state.
- If OCS d fails but all hb-relations are not captured, \(y = 1\) while others are 0, an inconsistent state.