Pessimistic Software Lock-Elision

Yehuda Afek
Alexander Matveev
Nir Shavit
Read-Write Locks

• One of the most prevalent lock forms in concurrent applications
  – 80/20 rule applies to reading vs writing of data
• Writes are serialized and cannot proceed with the reads
• Allow reads can proceed in parallel with one another
Coming Next Year:
HTM and Hardware Lock Elision
Speculative Lock Elision

- Rajwar and Goodman: speculative execution of locks by optimistic hardware transactions (Haswell)

  \[\textbf{Thread 1} \quad \begin{array}{c}
  \text{Lock Acquire} \\
  \text{Lock Elided} \\
  \text{Lock Release}
  \end{array} \quad \begin{array}{c}
  \text{Lock Acquire} \\
  \text{Lock Elided} \\
  \text{Lock Release}
  \end{array} \quad \textbf{Thread 2} \]

  - \textit{Speculate}: try to execute the critical sections concurrently (using transactions)
  - \textit{On failure}: revert back to the lock

- Roy, Hand, and Harris: \textbf{software implementation of SLE}, transactions executed speculatively in software.
**SLE: Good and Bad**

- **Advantages:** Concurrency among writes and among reads and writes -- as long as they do not share/contend for memory

- **Disadvantages:**
  - Contention implies default locking
    - Reads delayed
  - System calls and I/O cannot be used
  - Debugging hard due to the speculative nondeterministic behavior

---

*Suggested to rewrite the code for lock semantics.*
Our Pessimistic Lock-Elision

• Non-speculatively replace read-write locks
• by pessimistic software transactions

Preserves the lock semantics – no code rewriting

No-speculation: pessimistic concurrency between read and write critical sections

No failure: every transaction is executed once
Pessimistic STM [MatveevShavit2011]

• A commit-time privatizing STM in which all transactions execute once and never abort

• And read-only transactions run in parallel with themselves and writes

• To create PLE, we designed a new encounter-order version of this pessimistic STM that has wait-free read-only transactions
Pessimistic Read-Write Interaction

- **Problem:** Write transactions must not write to locations being read by overlapping reads.

- **Solution:**
  - On a write, the old value is logged publically before writing the new value.
  - In read phase, logged values of concurrent writes are read.
  - In the commit-phase, the old values are discarded after it is ensured using the *quiesence mechanism* that no-one reads them.
Why does this work well?

- No need for CAS or even memory barriers in common case
- Even though logging is public, its only by one transaction at a time so very easy to implement
Applying Pessimistic Lock-Elision

Program with **RW-Locks**

STM Compiler  
(Intel STM Compiler)  
(Add PLE)

Program with **PLE**

Processor with HLE (Intel’s Haswell)  
(HLE code is executed with software fallback to PLE)

Standard Processor  
(PLE code is executed)

The semantics are not changed with PLE addition

Concurrency between read and write critical sections

HLE has limitations, but HLE + PLE does not have

PLE works on current processors
Performance

- We empirically evaluated our algorithm on an Intel 40-way machine with 2 Xeon E7-4870 chips in a NUMA setup. The algorithms we benchmarked are:
  
  1. **PLE**: Our fully pessimistic encounter-time STM
  2. **RW_Lock_Egress**: An ingress-egress counter based reader-writer mutex implementation for Intel platform.
  3. **MCS-Lock**: Michael and Scott’s MCS Lock
  4. **RW_Lock_SPAA**: The new RWLock proposal from SPAA 2012

![Graphs showing performance comparison between different algorithms under different conditions.]
TL2 Style Memory Consistency
[Dice-Shalev-Shavit]

- The shared memory is divided to stripes
- Every stripe has an associated version number
- **Global version number** with stripes version numbers are used to synchronize access to the shared memory

<table>
<thead>
<tr>
<th>global version</th>
<th>ver nums</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>23</td>
<td>Stripe 1</td>
</tr>
<tr>
<td></td>
<td>23</td>
<td>Stripe 2</td>
</tr>
<tr>
<td></td>
<td>18</td>
<td>Stripe 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>Stripe 20</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Write Transactions Using Signalling

- Write transaction ops are executed sequentially
- Commits and ops can overlap
- Add a single global lock. A baton style *signaling mechanism* avoids common case lock acquire and release:
  - First write transaction acquires a lock
  - While lock taken all write transactions register requests
  - Before committing signal one new write to start
  - if none registered release lock
**Read Transaction**

- **On TxStart():**
  - RV := GV
- **On TxRead(addr):**
  - value = load(addr)
  - If (stripe(addr) <= RV)
    - Return value
  - Snoop into the concurrent writer log-set.
    - If addr is in log-set, then return the value from there
    - Else, return the value read on start of the function.
- **On TxCommit():**
  - Do nothing

**Write Transaction**

- **On TxStart():**
  - Wait for permission to run by the signalling mechanism
- **On TxRead(addr):**
  - return load(addr)
- **On TxWrite(addr, value):**
  - old_value = load(addr)
  - add (addr, old_value) to log-set
  - stripe(addr) = GV + 1
  - store(addr, value)
- **On TxCommit():**
  - GV := GV + 1
  - Signal the next writer
  - *Quiescence current parallel read transactions (that have RV < GV)*
  - reset log-set
**STAGE 1: Reads**

<table>
<thead>
<tr>
<th>Transaction 1</th>
<th>Transaction 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Read global version (RV == 22)</td>
<td>Version 22 in Stripe 1</td>
</tr>
<tr>
<td>2. Read from stripe 3 (18 &lt;= 22)</td>
<td>1. Read from stripe 1 (direct read)</td>
</tr>
<tr>
<td>3. Read from stripe 1 (22 &lt;= 22)</td>
<td>2. Read from stripe 3 (direct read)</td>
</tr>
</tbody>
</table>

**Log buffer:** empty

**Global version:** 22
**STAGE 2: Writes**

**Log buffer:**
- `addr1, old_val1`
- `addr2, old_val2`

<table>
<thead>
<tr>
<th>read transaction 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>write transaction 1</td>
</tr>
</tbody>
</table>

3. Write `(addr1, val1)` to stripe 1
   - 1. log old value,
   - 2. update the stripe version
   - 3. write the new value

4. Write `(addr2, val2)` to stripe 2
   - 1. log old value,
   - 2. update the stripe version
   - 3. write the new value

4. read `(addr1)` from stripe 1
   - (23 > 22 -> snoop
     - Scan the log buffer and return `old_val1`

5. read `(addr3)` from stripe 1
   - (23 > 22 -> snoop
     - Scan the log buffer;
     - `addr3` is not there;
     - return the value read from memory

**global version:** 22

<table>
<thead>
<tr>
<th>versions</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>Stripe 1</td>
</tr>
<tr>
<td>23</td>
<td>Stripe 2</td>
</tr>
<tr>
<td>18</td>
<td>Stripe 3</td>
</tr>
<tr>
<td>14</td>
<td>Stripe 4</td>
</tr>
</tbody>
</table>
**STAGE 3: Commit**

**global version:** 23

**Log buffer:** addr1, old_val1 | addr2, old_val2

1. **read transaction 1**

2. **write transaction 1**

3. **read transaction 2**

4. **versions**

5. **memory**

6. **Increment the global version**

7. **Quiescence** (Wait for read transaction 1 to finish)

8. **Quiescence done**

9. **Reset log buffer**

10. **Finish**