Inevitability Mechanisms for Software Transactional Memory

Michael Spear (Rochester)
Maged Michael (IBM)
Michael Scott (Rochester)
Why Inevitability

• Irreversible operations (I/O)
  – Especially “I after O”
  – Bufferable output when order matters or interleaving is forbidden
  – Preserves local reasoning about correctness
    • “atomic” means “all or nothing” and “all at once”

• Non-transactional code
  – Precompiled libraries (if binary rewriting is not available)
  – Lock-based code
  – Syscalls that change kernel state

• Speed
  – Turn off read/write instrumentation
  – e.g. matrix math
Caveats

• **Condition Synchronization**
  – Inevitable code can synchronize up to first potentially irreversible operation
  – Or at any point before becoming inevitable
  – Or via a special (limited applicability) closed nested transaction
  – All but last option are statically checkable (future work)

• **Library code**
  – Unpredictable read/write sets may dictate mechanism
  – May need **inevitable prefetching**
  – Indirection-based backends cause problems

• **I/O deadlocks remain**
  – Inevitable blocking read from empty pipe by T1 before inevitable write to same pipe by T2
How to Achieve Inevitability

- Only permit one inevitable transaction at a time
- Don’t let it abort
  - No explicit aborts: use eager locking, in-place update, augmented CM
  - No self aborts
  - No implicit aborts
    - Concurrent writer cannot commit changes to locations read by active inevitable transaction
    - This is the hard part

- Note: concurrent writer can’t commit if its read set overlaps with inevitable transaction’s write set
Inevitability Mechanisms

- No concurrency
  - Global Read/Write Lock (GRL)

- Concurrent readers
  - Global Write Lock (GWL)
  - Global Write Lock with Fence (GWL + Fence)
  - Drain

- Concurrent writers
  - Inevitable Read Locks (IRL)
  - Inevitable Read Filter (Filter)

- See the paper for implementation details

Note: Drain, GRL, and GWL+Fence may delay at inevitability point
## Sources of Latency

<table>
<thead>
<tr>
<th></th>
<th>Inev Read Instr</th>
<th>Inev Write Instr</th>
<th>Inev Read Logging</th>
<th>Inev Commit</th>
<th>Tx Begin</th>
<th>Non-Inev Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GWL</td>
<td>Wait</td>
<td>Acquire</td>
<td></td>
<td></td>
<td></td>
<td>Test</td>
</tr>
<tr>
<td>GWL + Fence</td>
<td>Store</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Test</td>
</tr>
<tr>
<td>Drain</td>
<td>Store</td>
<td></td>
<td>CAS</td>
<td></td>
<td></td>
<td>2 CASes (writers only)</td>
</tr>
</tbody>
</table>

**Notes:**
- **GWL:** Great Write Lock
- **GWL + Fence:** Great Write Lock + Fence
- **Drain:** Drain operation
- **CAS:** Compare and Swap
- **WB R:** Write Before Read
- **Test:** Testing phase
- **2 CASes (writers only):** Two Compare and Swap operations for writers only
Suitability to Tasks

- Library / Syscall with unpredictable write set
  - GRL
- Library / Syscall with unpredictable read set
  - Drain, GWL+Fence, GRL
- Short inevitable transactions with likely conflicts
  - GWL
- Short inevitable transactions with few conflicts
  - IRL, Bloom
- Long but infrequent inevitable transactions
  - GWL+Fence
- Long, frequent inevitable transactions
  - Drain
Evaluation

• In the paper: microbenchmarks
  – Only Drain increases latency of short non-inevitable transactions
  – GWL and “small” Filter flat-line a scalable benchmark
  – Drain starts higher, but dampens scaling
  – Fences are relatively fair, but don’t accelerate workloads with >1 thread
  – For big tasks, Drain is a good accelerator

• In this talk: a new benchmark
  – Asynchronous OpenGL 3-D rendering
  – Joint work with Michael Silverman and Luke Dalessandro
Why Write a New Benchmark?

• Today’s programs written by today’s programmers
  – Trained to think about critical sections, locking, deadlock, and mutual exclusion

• Who writes tomorrow’s programs?
  – We hope they will think about transactions, rollback, conflicts, and atomicity

• Social experiment: get a smart undergraduate to write code with (moderate) supervision
  – Takes a couple of iterations to get the code “right”
  – But the programmer has a different (more transaction-friendly) philosophy
  – The result will probably have some relation to a game 😊
A 3-D OpenGL Scene Graph

- Animated Multisegment Objects (AMOs)
  - Big transaction does physics, animation, collision detection
  - Not a “read then write” transaction
  - Collision detection with anything that is “close”

- Gravity Emitting Objects (GEOs)
  - Not animated, don’t have initial velocity, but do collision detection
  - Attract AMOs

- Game: rescue AMOs before they fall into a GMO (about 2 minutes)
- Benchmark: nobody playing the game, 500 AMOs, 10 GEOs
Screenshot: Early in Simulation
Inevitability Mechanisms for STM

Screenshot: AMOs Converging
Thread Configuration

• One thread continuously renders
  – Read AMOs in transactions to render new frame, then make an OpenGL call

• All other threads continuously update AMOs / GEOs
  – Simulate physics based on time

• Inevitable rendering or inevitable AMO updates
  – Without inevitability, renderer must explicitly buffer reads to ensure consistency
  – With inevitability, can aggressively batch renderer’s reads

• “Best” choice is a function of the number of cores
  – Frame rate is decoupled from update rate, so higher not always better
  – Ideally, update rate \( \approx \) frame rate \( \approx \) screen refresh rate
Environment

• Code
  – Uses new RSTM v2 API for word-based STMs
  – TL2-like back-end
  – Open source (will release soon)

• Platform
    • Also OS X, Linux versions
  – 2.6 GHz Q6600 (quad core), 4 GB RAM
  – NVIDIA 8800 GTS
• 60 FPS is the refresh rate
• Inevitable render overheads
• Fences hurt
• Do updaters impede renderer?
AMO Commits per Second

- Desire 30,000 commits
- Bloom filter precision

- No writer commit != blocked
- GWL starves, fences don’t
Conclusions

• Mechanisms have benefits and drawbacks
  – We think the mechanisms can compose
  – Many are also applicable to HTM, HyTM

• If transactions are for the masses, inevitability is crucial
  – Local (and simple) reasoning about correctness
  – Need not sacrifice concurrency

• New open-source OpenGL benchmark to further our understanding of transactions and inevitability
Questions / Discussion
Supplemental Slides
Global Read/Write Lock

- Acquire exclusive permission to read / write shared locations
  - Independent of oreCs
- Must wait for clean-up
  - Otherwise, would have to instrument reads and writes
- Concurrent readers won’t detect conflicts, so they can’t run

- State of the art for STM
Read-Only Concurrency (1/2)

- Global Write Lock
  - Acquire exclusive permission to write shared locations
    - Update metadata when writing
    - With commit-time locking, writers can run up to commit point
  - No waiting, but instrument reads to handle delayed cleanup
  - Rapid succession of inevitable transactions can starve big concurrent writers
- Global Write Lock + Fence
  - Wait for cleanup after becoming inevitable
    - No risk of delayed cleanup… no read instrumentation for inevitable transaction
    - Inevitable transaction acquires with stores, not CASes
Read-Only Concurrency (2/2)

• The Drain
  – Like a fair reader-writer lock
    • Inevitable transaction is “writer”
    • Concurrent writer transactions are “readers”
  – No inevitable read instrumentation, store to acquire inevitably
  – Serialization on single global
    • 2 CASes to commit any writer
    • CAS to release inevitability
Inevitability Mechanisms for STM

Read/Write Concurrency

• Inevitable Read Locks
  – Add an inevitable read bit to each orec
    • Noninevitable writers can’t acquire orec if bit is set
  – CAS on every inevitable read
    • Cache misses for concurrent readers

• Inevitable Read Filter
  – Approximate IRL bits as a Bloom filter
    • Less precise, but no misses for concurrent readers
  – WBR ordering to update filter
    • Write filter before checking if orec is held
  – WBR ordering to coordinate concurrent writers
    • Acquire orec before checking filter (PPC only)
    • Favors commit-time locking
## Concurrency Summary

<table>
<thead>
<tr>
<th></th>
<th>Delay upon becoming inevitable</th>
<th>Concurrent read-only</th>
<th>Concurrent writes</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRL</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GWL</td>
<td>No</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>GWL + Fence</td>
<td>Yes</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Drain</td>
<td>Sometimes</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>IRL</td>
<td>No</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Bloom</td>
<td>No</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>