Boosting Timestamp-Based TM by Exploiting HW Cycle Counters

Wenjia Ruan, Yujie Liu, and Michael Spear
Lehigh University
Global Counters… A Blessing and Curse

- Starting with TL2 in 2006, virtually every STM algorithm has accepted a global bottleneck
  - Exception: visible readers (e.g., TLRW)

- Some bottlenecks worse than others
  - Privatization-safe STM → serialized commit
  - Non-Privatization-safe STM → usually just incrementing a counter

- These bottlenecks play a vital role in lowering the cost of consistency checks
  - Every writer commits at a unique time
  - Transactions start relative to a writer commit time
  - Easy to determine if a location changed after a transaction started
  - Typically avoids quadratic validation
Problems with Global Counters

- Just a heuristic… no asymptotic improvement
  (Not addressed in this paper)

- Multi-chip results do not match single-chip results
  - Off-chip coherence costs are huge

- Small, frequent, disjoint writer transactions do not scale

- Require an extra atomic operation at commit time

- This paper presents one way to avoid global counters on x86 CPUs
Prior Work

- Most significant work in this area from Riegel et al., SPAA 2007
  - Used Altix MMTimer in place of a clock
  - Faced issues of clock skew and drift
  - MMTimer relatively slow

- Another approach was part of original TL2
  - A failed CAS on the counter is often *good enough*
  - So is a read of the counter instead of an increment
This Work

• We focus on x86-specific features
  – May generalize to SPARC (stick), ARM, POWER
  – Works on Nehalem and Sandy Bridge, forward-compatible
  – But don’t try it on a Core 2

• We cannot prove this system correct
  – Underspecified ordering for today’s x86 CPUs

• Our solution is not a panacea
  – Sacrifice key optimization for privatization safety
Background: Check-Twice Ownership Records

- Most of us understand TL2-style oreCs
  (Note: Ignoring timestamp extension)

```
begin():
    start_time = time

read(addr):
    if addr in writes
        return writes[addr]
    o2 = oreCs[addr].val
    val = *addr
    o1 = oreCs[addr].val
    if o1 == o2 and
        o1 <= start_time and
        !o1.locked
        reads += addr
        return val
    else Abort()

write(addr, val):
    writes[addr] = val

commit():
    if writes == nil return
    acquire_locks()
    end = atomicinc(time)
    validate(end)
    writes.writeback()
    release_locks(end)
```
Fewer of us understand check-once orecs [Wang CGO 07]
- Less code in read(), different order in commit()

begin():
    start_time = time

read(addr):
    if addr in writes
        return writes[addr]
    val = *addr
    ol = orecs[addr].val
    if
        ol <= start_time and !ol.locked
        reads += addr
        return val
    else
        Abort()

write(addr, val):
    writes[addr] = val

commit():
    if writes == nil
        return
    acquire_locks()
    validate()
    writes.writeback()
    end = atomicinc(time)
    release_locks(end)
Check-Once Ownership Records

- Asymptotically the same cost
  - But fewer branches and instructions in read()

- Why it works:
  - Suppose $T_A$ starts after $T_B$ increments the counter, then $T_A$ reads
  - The first check in TL2 is for the case when $orec[addr]$ is locked when $T_A$ reads $*addr$
    - Necessary… $T_B$ may not have done writeback yet, $T_A$ might still see old value
    - In this case, $T_B$ already did its writeback
      - Even if $T_B$ still owns $orec[addr]$, $T_A$ can read $*addr$

- Note: can’t skip commit-time validation like in TL2
TSO Reminders

• Incrementing a shared counter via `lock cmpxchg` or `lock inc` is a full memory barrier
  – Happens after locks acquired
  – Happens before locks released
  – Algorithm (check-once, check-twice) determines whether it happens *completely* before or *completely* after validation and writeback

• Key point: it can’t reorder with respect to anything

• Also note: monotonic, consistent across cores/CPUs
Replacing the Counter

• The x86 `rdtsc` and `rdtscp` instructions have a long, ugly history
  – At times, considered unreliable
  – In modern x86 processors, invariant w.r.t. clock scaling
  – Note: `rdtscp` is ordered after preceding loads
    • No guarantees relative to subsequent loads
    • No guarantees relative to any stores

• Critical guarantee:
  “On modern 64-bit x86 architectures, if one core writes the result of an `rdtscp` instruction to memory, and another core reads that value prior to issuing its own `rdtscp` instruction, then the second `rdtscp` will return a value that is not smaller than the first.”
Can We Just Use \texttt{rdtscp}?

• Since the cycle counter is continuously increasing:
  – We could read it as the start time of the transaction
  – We could read it as the commit time of the transaction (no \texttt{cmpxchg})
    – But we need to be sure that it has the same guarantees (full memory fence), or else enforce ordering some other way

• Note: validation at commit time will be unavoidable
  – Disincentive for check-twice oreCs
Challenge #1: Begin Time

- This looks benign:
  - But no guarantees between `rdtscp` and subsequent loads

- Could it delay all the way to here?
  - If so, $T_A$ could read old value, then $T_B$ could update value and commit, then $T_A$ could receive a start time after $T_B$’s commit
  - Seems unlikely... lots of data dependencies in the asm to get to this point, especially due to function calls
  - Using an `LFENCE` for now
  - We’d like a guarantee!

```c
begin():
    start_time = rdtscp

read(addr):
    if addr in writes
        return writes[addr]
    val = *addr
    ol = orecs[addr].val
    if
        ol <= start_time and
        !ol.locked
        reads += addr
        return val
    else Abort()
```
Challenge #2: Commit Time (Check-Twice Orecs)

- What if `rdtscp` happens too late?
  - Not possible!
  - Data dependence with `validate()`, and control dependence with `writeback()`

- What if `rdtscp` happens too soon?
  - Not possible!
  - Control dependence, and `rdtscp` has load fence semantics, and `acquire_locks()` is implemented with `lock cmpxchg` (entails a read)

```python
commit():
    if writes == nil return
    acquire_locks()
    end = rdtscp
    validate(end)
    writes.writeback()
    release_locks(end)

read(addr):
    if addr in writes
        return writes[addr]
    o1 = orecs[addr].val
    val = *addr
    o2 = orecs[addr].val
    if o1 == 02 and o1 <= start_time and not o1.locked
        reads += addr
        return val
    else
        Abort()
```

3/19/2013
Challenge #3: Commit Time (Check-Once Orecs)

- **What if `rdtscp` happens too late?**
  - Not possible! Data dependence with `release_locks()`

- **What if `rdtscp` happens too soon?**
  - Error!
  - Concurrent reader can read before `writeback()`
  - If reader `begin()` after writer `rdtscp`, then reader does not detect inconsistency

```plaintext
commit():
    if writes == nil return
    acquire_locks()
    validate()
    writes.writeback()
    end = rdtscp
    release_locks(end)

read(addr):
    if addr in writes
        return writes[addr]
    val = *addr
    o1 = orecs[addr].val
    if
    o1 <= start_time and !o1.locked
        reads += addr
        return val
    else Abort()
```
Enforcing Store/Load Ordering

- Recall that `rdtscp` has LFENCE semantics
  - Can’t bypass preceding load, can bypass preceding store

- Solution: use an atomic read-modify-write operation
  - Add 0 to top of stack
  - Involves a load, so the `rdtscp` must follow it
  - But it has full (MFENCE) ordering after writeback

```c
commit():
  if writes == nil return
  acquire_locks()
  validate()
  writes.writeback()
  lock add 0, (%esp)
  end = rdtscp
  release_locks(end)
```
Privatization

• A common solution to the privatization problem is to split it into two halves:

• The delayed cleanup problem (writeback version)
  – Prevent a writer from exiting its commit() function if an earlier writer is still in commit()

• The doomed transaction problem
  – Poll the timestamp on every read, validate whenever it changes
  – Note: this requires using a deferred reclamation allocator

• When using rdtscp, there is nothing to poll!
  – Must use a heavyweight quiescence barrier instead
  – Can use validation fence: update thread state on every validation
Evaluation

- Single and dual-chip Xeon 5650 (Sandy Bridge) machines
  - Each chip has 6 cores / 12 threads
  - GCC 4.7, Ubuntu 12.04, C++

- Implementation in RSTM

- No surprises
  - We avoid bottlenecks, and we have the same latency as non-rdtscp algorithms
  - Privatization is costly

- Note: non-conventional Labyrinth
  - You should never use the version in STAMP… ever…
Evaluation (Microbenchmarks)

(a) Hash Table, single-chip

(b) Red-Black Tree, single-chip

(c) Hash Table, dual-chip

(d) Red-Black Tree, dual-chip

3/19/2013

Boosting STM with HW Cycle Counters
Evaluation (STAMP, single-chip system, 1/2)

(a) Intruder

(b) Genome

(c) SSCA2

(d) Labyrinth

3/19/2013

Boosting STM with HW Cycle Counters
Evaluation (STAMP, single-chip system, 2/2)

(a) KMeans (low contention)
(b) KMeans (high contention)
(c) Vacation (low contention)
(d) Vacation (high contention)
Evaluation (STAMP, dual-chip system, 1/2)

(a) Intruder

(b) Genome

(c) SSCA2

(d) Labyrinth

3/19/2013

Boosting STM with HW Cycle Counters
Evaluation (STAMP, dual-chip system, 2/2)

(a) KMeans (low contention)

(b) KMeans (high contention)

(c) Vacation (low contention)

(d) Vacation (high contention)
Conclusions

• The performance is quite nice

• You can’t trust this code yet
  – Need better specification regarding effectiveness of LFENCE in begin()

• Is rdtsc+mfence a useful instruction?

• Is it time to revisit strong isolation?
  – No bottlenecks on a counter to maintain opacity
  – Load becomes 2 loads, 1 branch, and an rdtscp
  – Store becomes lock cmpxchg, 1 branch, 2 stores, and an rdtscp
Questions

We extend a special thanks to our reviewers, and to Ravi Rajwar, Stephan Diestelhorst, and Dave Dice

Latest RSTM code available at http://code.google.com/p/rstm/ (this work in the ‘wenjia’ branch)