Understanding Hardware Transactional Memory

Gil Tene, CTO & co-Founder, Azul Systems
@giltene
Agenda

Brief introduction

What is Hardware Transactional Memory (HTM)?

Cache coherence basics & how HTM works

What it looks like from a runtime point of view

Interesting coding considerations
About me: Gil Tene

- co-founder, CTO @Azul Systems
- Have been working on “think different” GC and runtime approaches since 2002
- Built world’s 1st commercially shipping HTM system, along with JVM support for HTM
- A Long history building Virtual & Physical Machines, Operating Systems, Enterprise apps, etc...
- I also depress people by demonstrating how terribly wrong their latency measurements are...

* working on real-world trash compaction issues, circa 2004
As far as I am concerned GC is a solved problem

@mjpt777 My GC tuning is well practised. 1) Enable Zing. 2) Open beer. /cc @giltene
Why does HTM matter now?

Because it is (finally) here!

HTM already available in the past
- e.g. Azul Vega, since 2004
- e.g. some later variants of Power architecture
- e.g. some designs for SPARC

But it is now here in commodity server chips
- Intel TSX, on modern Intel Xeons
- Already in E7-V3 (4+ sockets), E3-V4, E3-V5 (1 socket)
- Coming (1H2016) in E5-26xx V4 ("Broadwell") chips
- Most new servers will have HTM by 2H2016
What is HTM?
What is HTM?
(For the kind of HTM I will be talking about...)

- Can be thought of as "Speculative Multi-Address Atomicity"
- Transaction starts and ends with explicit instructions
- No special load or store instructions
- All memory operations in a successfully completed transaction appear to execute atomically (to other threads)
- Transactions may abort. All memory operations in an aborted transaction appear “to have never happened"
Cache Coherence

Protocols can be messy
PR = processor read
PW = processor write
S/∼S = shared/NOT shared
BR = observed bus read
BW = observed bus write
MSI Coherence Protocol States and Transitions
But conceptually it’s not that messy...
Cache line state from an individual CPU’s point of view

I  I don’t have it
   (Invalid)

S  I have a copy (and someone else may, too)
   (Shared)

E  I have the only copy
   (Exclusive)

M  I have the only copy, and I’ve changed it
   (Modified)
Cache line state from an individual CPU's point of view

M 🎓 I have the only copy, and I've changed it
(Modified)

E 🎓 I have the only copy
(Exclusive)

S 🎓 I have a copy (and someone else may, too)
(Shared)

I 🎓 I don't have it
(Invalid)
HTM builds on existing cache coherence
Conceptual cache line state additions for HTM

- Line was accessed during speculation:
  - Line was read from during speculation
  - Line was modified during speculation

- When transaction completes, clear all speculation tracking state

- Losing track of a line that was accessed during speculation aborts the transaction

- Aborts invalidate speculatively modified lines
What can make the cache “lose track” of a line?

- Another CPU wants to write to it
  - Other CPU would need it “exclusive”
  - It would first need to invalidate it in this cache

- Another CPU wants to read from it
  - Cache line would need to be in “shared” state here
  - If it were in speculatively modified state: abort

- Capacity related self-eviction
  - E.g. current Xeons: 32KB, 8 way set associative
That’s it…. For memory.
CPU state. E.g. Intel TSX

In addition to memory transactionality, CPU architectural state is maintained

- On abort, PC moves to location provided in XBEGIN
- EAX is changed to indicate abort information
- All other architecture state remains the same as it was before XBEGIN was executed
So when can you do with HTM?

Well, transact on memory, of course...
Speculative Lock Elision

2001 PhD. thesis by Ravi Rajwar

** An independent work on a somewhat similar lock serialization avoidance concept by Jose F. Martinez and Josep Torrellas, UIUC, also published in 2001
Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution

Here is a VERY interesting article by the same guy (Ravi Rajwar). Published Dec. 2001. Well worth the downloading of a postscript viewer (Ghostview). This guy really deserves that PhD!

Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution

The gist of it is a microarchitectural support (with no ISA or program modification) to allow for speculative removal of synchronization operations while maintaining full and correct semantic execution through the detection of false speculation and roll-back/recovery using common (existing) cache coherence mechanisms. Threads will speculate lock acquisition without actually going through the lock acquisition memory/cache operation. Speculative memory state is stored in caches, and is "rolled back" in the event that the speculation is found to be wrong.

This sort of speculation could facilitate an amazing improvement in thread concurrence of code that uses conservative, non-performance-optimized synchronization.

-- Gil.
Using HTM under the hood in a JVM

A trip down transactional memory lane
New JVM capabilities improve multi-threaded application scalability. How can this affect the way you code?

Speculative locking reduces effects of Amdahl's law
Why do we care?

Lock contention vs. Data contention

Transactional execution of `synchronized {...}`

Measurements

Effects on how you code for contention

Summary
Amdahl’s Law

Serialized portions of program limit scale

- efficiency $= 1/(N \times q + (1-q))$
  - $N$ = # of concurrent threads
  - $q$ = fraction of serialized code
Amdahl’s Law Effect on Throughput

Throughput scale factor vs. processors for different percentages of parallelizable code:
- 0.10%
- 0.50%
- 1.00%
- 2.00%
- 5.00%
- 20.00%
- Ideal (no parallelization)

As the number of processors increases, the throughput scale factor increases linearly for a given percentage of parallelizable code, approaching the ideal throughput as the percentage increases.
Amdahl’s Law Example

- The theoretical limit is usually intuitive
  - Assume 10% serialization
  - At best you can do 10x the work of 1 CPU

- Efficiency drops are dramatic and may be less intuitive
  - Assume 10% Serialization
  - 10 CPUs will not scale past a speedup of 5.3x (Eff. 0.53)
  - 16 CPUs will not scale past a speedup of 6.4x (Eff. 0.48)
  - 64 CPUs will not scale past a speedup of 8.8x (Eff. 0.14)
  - 99 CPUs will not scale past a speedup of 9.2x (Eff. 0.09)
  - ...
  - It will take a whole lot of inefficient CPUs to [never] reach a 10x
Agenda

Why do we care?

Lock contention vs. Data contention

Transactional execution of synchronized {...}

Measurements

Effects on how you code

Summary
Lock Contention vs. Data Contention

- Lock contention:
  
  *An attempt by one thread to acquire a lock when another thread is holding it*

- Data contention:
  
  *An attempt by one thread to atomically access data when another thread expects to manipulate the same data atomically*
Data Contention in a Shared Data Structure

- Readers do not contend
- Readers and writers don’t always contend
- Even writers may not contend with other writers
Synchronization and Locking

Locks are typically very conservative

- Need synchronization for correct execution
  - Critical sections, shared data structures

- Intent is to protect against data contention

- Can’t easily tell in advance
  - That’s why we lock…

- Lock contention $\geq$ Data contention
  - In reality: lock contention $\gg= Data$ contention
Database Transactions

The industry has already solved a similar problem

• Semantics of potential failure exposed to the application
• Transactions: atomic group of DB commands
  – All or nothing
  – From "BEGIN TRANSACTION" to "COMMIT"
• Data contention results in a rollback
  – Leaves no trace
• Application can re-execute until successful
• Optimistic concurrency does scale
Agenda

Why do we care?
Lock contention vs. Data contention

**Transactional execution of `synchronized {...}`**

Measurements
Effects on how you code for contention
Summary
There is no spoon.
What does *synchronized* mean?

- It does not actually mean:
  
  `grab lock, execute block, release lock`

- It does mean:
  
  `execute block atomically in relation to other blocks 
synchronizing on the same object`

- It can be satisfied by the more conservative:
  
  `execute block atomically in relation to all other threads`

- That looks a lot like a transaction

---

Transactional execution of synchronized {...}

- Two basic requirements
  - Detect data contention within the block
  - Roll back `synchronized` block on data contention

- `synchronized` can run concurrently
  - JVM can use hardware transactional memory to detect data contention
  - JVM must rolls back `synchronized` blocks that encounter data contention
Transactional execution of `synchronized {...}

- The JVM maintains the semantic meaning of: `execute block atomically in relation to all other threads`
- Uncontended `synchronized` blocks run just as fast as before
- Data contended `synchronized` blocks still serialize execution
- `synchronized` blocks without data contention can execute in parallel
Transactional execution of `synchronized { ... }

- It’s all transparent!
- No changes to Java code
  - The VM handles everything
- Nested `synchronized` blocks
  - Roll back to outermost transactional `synchronized`
- Reduces serialization
- Amdahl’s Law now only reflects data contention
  - Desire to reduce data contention
Implementation in a JVM

How does it fit in the current locking schemes?

- Thin locks handle uncontended synchronized blocks
  - Most common case
  - Uses CAS, no OS interaction

- Thick locks handle data contended synchronized blocks
  - Blocks in the OS

- Transactional monitors handle contended synchronized blocks that have no data contention
  - Execute synchronized blocks in parallel
  - Uses HW transactional memory support
Agenda

Why do we care?
Lock contention vs. Data contention
Transactional `synchronized {…}`

Measurements
Effects on how you code for contention
Summary
Data Contention and Hashtables

• Examples of no data contention in a Hashtable
  – 2 readers
  – 1 reader, 1 writer, different hash buckets
  – 2 writers, different hash buckets

• Examples of data contention in a Hashtable
  – 1 reader, 1 writer in same hash bucket
  – 2 writers in same hash bucket
Measurements: Hashtable (0% writes)
Measurements:Hashtable (5% writes)

- Locking
- Spec. Locking

<table>
<thead>
<tr>
<th>Threads</th>
<th>Locking</th>
<th>Spec. Locking</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Measurements: Hashtable (5% writes)
Agenda

Why do we care?
Lock contention vs. Data contention
Transactional synchronized {...}
Measurements
Effects on how you code for contention
Summary
private Object table[];
private int size;

public synchronized void put(Object key, Object val) {
    ...
    // missed, insert into table
    table[idx] = new HashEntry(key, val, table[idx]);
    size++; // writer data contention
}

public synchronized int size() {
    return size;
}
private Object table[];
private int    sizes[];

public synchronized void put(Object key, Object val) {
    ...
    // missed, insert into table
    table[idx] = new HashEntry(key, val, table[idx]);
    sizes[idx]++; // reduced writer data contention
}

public synchronized int size() {
    int size = 0;
    for (int i=0; i<sizes.length; i++) size += sizes[i];
    return size;
}
private Object table[];
private int sizes[];
private int cachedSize;

public synchronized void put(Object key, Object val) {
    ...
    // missed, insert into table
    table[idx] = new HashEntry(key, val, table[idx]);
    sizes[idx]++;
    cachedSize = -1; // clear the cache
}

public synchronized int size() {
    if (cachedSize < 0) {
        // reduce size recalculation
        cachedSize = 0;
        for (int i=0; i<sizes.length; i++)
            cachedSize += sizes[i];
    }
    return cachedSize;
}
private Object table[];
private int sizes[];
private int cachedSize;

public synchronized void put(Object key, Object val) {
    ...
    // missed, insert into table
    table[idx] = new HashEntry(key, val, table[idx]);
    sizes[idx]++;
    if (cachedSize >= 0) cachedSize = -1; // avoid contention
}

public synchronized int size() {
    if (cachedSize < 0) {
        cachedSize = 0;
        for (int i=0; i<sizes.length; i++)
            cachedSize += sizes[i];
    }
    return cachedSize;
}
Agenda

Why do we care?
Lock contention vs. Data contention
Transactional synchronized {...}
Measurements
Effects on how you code
Summary
Summary

- Hardware Transactional Memory is here!
- Expect speculative use for locking and synchronization in libraries and runtimes
  - JVMs, CLR, ...
  - POSIX mutexes, semaphores, etc.
- It may be useful for some other cool stuff...