Unlocking Concurrency with Transactional Memory

Paolo Romano
romano@inesc-id.pt

TopConf, Tallinn, Estonia, November 20 2014
Roadmap

• Bliss & pitfalls of concurrent programming

• Transactional Memory (TM)
  – What it is?
  – How it works?

• TM support in programming languages
The era of free performance gains is over

- Over the last 30 years:
  - new CPU generation ➞ free speed-up

- Since 2003:
  - CPU clock speed plateaued...
  - but Moore’s law chase continues:
    - Multi-cores,
    - Hyperthreading...

FUTURE IS PARALLEL
Traditional Software Scaling

- Speedup
  - 1.8x
  - 3.6x
  - 7x

- User code
- Traditional Uniprocessor

Time: Moore’s law
Ideal Multicore Software Scaling

Unfortunately this is not the case in practice....
Real-World Multicore Scaling

Hard to parallelize application efficiently:
- correct synchronization
- load balancing
- data locality
Coarse-grained Locking?

simple but does not scale

Amdahl’s Law:

\[ \text{Speedup} = \frac{1}{(\text{ParallelPart}/N + \text{SequentialPart})} \]

Pay for \( N = 128 \) cores

\( \text{SequentialPart} = 25\% \)

As num cores grows the effect of 25% becomes more acute

2.3/4, 2.9/8, 3.4/16, 3.7/32, …
The reason we get only limited speedup is that fine-grained parallelism has huge performance benefit.

Fine-grained Locking?

Coarse Grained

Fine Grained

75% Unshared

25% Shared

75% Unshared

25% Shared
Fine-grained Locking?

easier said than done

• Fine grained locking is **hard to get right:**
  – deadlocks, livelocks, priority inversions:
  – complex/undocumented lock acquisition protocols
  – no composability of existing software modules

... and a **verification nightmare:**
  • subtle bugs that are extremely hard to reproduce
Lock-based synchronization does not support modular programming

- Synchronize moving an element between lists
  
  ```
  void move(list l1, list l2, element e)
  {
    if (l1.remove(e)) l2.insert(e);
  }
  ```

- Assume remove/insert acquire a per-list lock

- Consider two threads that execute:

  **Thread1**
  
  - move(list1, list2, e)
  - list1.lock() ➔ OK
  - list2.lock() ➔ wait T1

  **Thread2**
  
  - move(list2, list1, e')
  - list2.lock() ➔ OK
  - list1.lock() ➔ wait T2
% Transactional memory (TM)

- Same idea as in a ACID database transaction:
  - “Write simple sequential code & wrap \texttt{atomic} around it”.
  - Hide away synchronization issues from the programmer
    - Programmers say what should be made atomic...
      and not how atomicity should be achieved
  - way simpler to reason about, verify, compose
  - similar performance to fine-grained locking
    - via speculation & possibly hardware support
TM : Brief historic overview

– Original idea dating back to early 90s
  • Herlihy/Moss ISCA 1993 ➔ hardware-based
– Over the last 10 years: one of the hottest research topic in parallel computing in academy and industry
– Latest generations of IBM® and Intel® CPUs ship with hardware support for TM
– Standardization efforts on C/C++
– TM supports in lots of programming languages
How does it work?

• Various implementations are possible:
  
  – Software (STM):
    • instrumenting read and write accesses
      – PRO: flexibility
      – CON: instrumentation overheads
  
  – Hardware (HTM):
    • extension of the cache consistency mechanism
      – PRO: no instrumentation overheads
      – CON: hw is inherently limited
  
  – Hybrid (HyTM)
    • mix of the two worlds that tries to achieve the best of both
STM

• **Many** algorithms proposed in the last 10 years:
  – DSTM, JVSTM, TL, TL2, LSA, TinySTM, SwissTM, TWM, NOREC, AVSTM...

• **Key design choices**
  – word vs object vs field based
  – single-version vs multi-version
  – in-place write & undo logs vs deferred writes & redo logs
  – lock-based vs lock-free
  – lazy locking vs eager locking
  – visible vs invisible reads
  – progress: no deadlock, no livelocks, no abort for RO tx,...
Example STM Algorithm : TL2
(Transactional Locking 2)

Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. DISC 2006
TL2 overview

• Key design choices
  • **word**- vs object vs field based
  • **single-version** vs multi-version
  • in-place write + undo logs vs **deferred writes + redo logs**
  • **lock-based** vs lock-free
  • **lazy locking** vs eager locking
  • visible vs **invisible reads**
  • progress: **no deadlock**, no livelocks, no abort for RO tx

...achieved via an external contention manager
(e.g., exponential back-off of aborted transactions)
Versioned Locks

Application Memory

Map

Array of Versioned-Write-Locks

PS = Lock per Stripe (separate array of locks)

PO = Lock per Object (embedded in object)
# Read-only Transactions

<table>
<thead>
<tr>
<th>Mem</th>
<th>Locks</th>
<th>VClock</th>
</tr>
</thead>
<tbody>
<tr>
<td>87</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>34</td>
<td>0</td>
<td>87</td>
</tr>
<tr>
<td>88</td>
<td>0</td>
<td>88</td>
</tr>
<tr>
<td>99</td>
<td>0</td>
<td>99</td>
</tr>
<tr>
<td>44</td>
<td>0</td>
<td>44</td>
</tr>
<tr>
<td>50</td>
<td>0</td>
<td>50</td>
</tr>
</tbody>
</table>

### On Tx begin

RV ← VClock

### On Read

read lock, read mem, read lock:
check unlocked, unchanged, and
v# <= RV

### On Commit

nothing to be done!

---

Reads from a consistent snapshot of memory.
No need to track and validate read set!
Update transactions

On Tx begin
    RV ← VClock
On Read/Write
    check unlocked and v# ≤ RV
    then add to Read/Write-Set
On Commit
    1. Acquire Locks
    2. WV = F&I(VClock)
    3. Validate each v# ≤ RV
    4. Release locks with v# ← WV

```
<table>
<thead>
<tr>
<th>Mem</th>
<th>Locks</th>
<th>Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>87</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>121</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>88</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>121</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
STM Performance: the bright side

(Azul – Vega2 – 2 x 48 cores)
STM Performance: the dark side

[Graph showing performance comparison between Lock and STM]
Sources of overhead in STMs

• STM scalability is as good if not better than locking, but overheads are much higher.

• Key sources of overhead:
  - Instrumented Memory accesses: constant overhead on every read/write.
  - Readset validation: proportional to the number of read items.
  - Lock acquisition: turns out to be less relevant.
How does it work?

• Various implementations are possible:
  – Software (STM):
    • instrumenting read and write accesses
      – PRO: flexibility
      – CON: instrumentation overheads
  – Hardware (HTM):
    • extension of the cache consistency mechanism
      – PRO: no instrumentation overheads
      – CON: hw is inherently limited
  – Hybrid (HyTM)
    • mix of the two worlds that tries to achieve the best of both
HTM is now available in several CPUs

- Intel: Haswell in desktops, laptops, tablets, servers...
- IBM: BG/Q, zEC12, Power8

HTM implementations are NOT born equal...

...yet they share two important commonalities:

1. Extend pre-existing cache coherency protocol
2. Best-effort nature

Catch: INTEL detected an undisclosed bug, which will be fixed in future Haswell releases.
Overview of Haswell’s HTM: TSX

CPU 1

xbegin
read x: 0 // Set bit read on x cache line
write y = 1 // Buffer write in L1 cache
xend // Atomically clean bits and publish

write y = 2

invalidation

CPU 2

xbegin
read y: 1
write y = 2
xabort

snoop write invalidates tx read

xbegin
read y: 1

…

y: 1 -- r
HTM’s best effort nature

No progress guarantees:

- A transaction may **always** abort

...due to a number of reasons:

- Forbidden instructions
- Capacity of caches
- Faults and signals
- Contending transactions, aborting each other
Fallback plan!

• After a few attempts using HTM, the tx is executed using a software synch. mechanism:
  – Single global lock *(current standard approach)*
    • PRO: success guarantee, support for not-undoable ops.
    • CON: no parallelism (extermination of concurrent hw tx)

  – STM ➔ Hybrid TM
    • PRO: fallback path does support parallelism
    • CON: tricky to coordinate concurrent execution of HTM
HTM performance (1/2)

- HTM shines when fallback is rarely executed, e.g.:
  - concurrent data structures
  - applications with short transactions

Kmeans: parallel implementation of data clustering algorithm (machine learning domain)
HTM Performance (2/2)

• STM is very competitive for applications with long, conflict prone transactions
• HybridTM are not quite there yet:
  – worst of both worlds 😞

Yada: parallel version of Delaunay triangulation (graph analysis algorithm)
TM support in programming languages
TM in C/C++

• No data annotations or special data types required:
  ```c
  __transaction_atomic { if (y> x) x++; }
  ```
  – Existing (sequential) code can be used in transactions: function calls, nested transactions, ...

• Code in atomic transactions must be *transaction-safe*
  – Compiler checks whether code is safe (gcc -fgnu-tm)
  – Unsafe: use of locks or atomics, asm, volatile, functions not known to be safe
  – For cross-CU calls / function pointers, annotate functions:
    • `void foo() __attribute__((transaction_safe)) { x++; }`

• Further information: ISO C++ paper N3718
GCC implementation (4.1.7+) : TM runtime library (libitm)

• Enforces atomicity of transactions at runtime
• libitm ships with different STM implementations
  – Does not require special hardware
  – Default:
    • Write-through with undo logging
    • Multiple locks (automatic memory-to-lock mapping)
• as well as HTM-based implementations!
  – libitm uses HTM with a global lock as fallback
  – no hybrid STM/HTM yet
Cool, but I only do JAVA...

- HTM support not yet integrated in standard JVM
- Yet, there are several high-quality STM implementations for JAVA:
  - JVSTM: http://inesc-id-esw.github.io/jvstm/
    - used in production at Lisbon University
      - manage life of entire campus (>10K users, highly available system)
    - requires manual annotation of transactional objects
  - DeuceTM: https://sites.google.com/site/deucestm/
    - automatic instrumentation via bytecode rewriting
  - Akka: http://doc.akka.io/docs/akka/2.1.0/scala/stm.html
    - based on SCALA STM (http://nbronson.github.io/scala-stm/)
(S)TM support in other languages

• (S)TM has been integrated in a growing number of programming languages:
  – C#, Clojure, Haskell, Javascript (based on node.js), Perl, Python, ...

• ...and to Distributed/Cloud computing settings:
  – Cloud-TM Project: [www.cloudtm.eu](http://www.cloudtm.eu)
Get involved!

• TM can drastically simplify parallel programming...
• ...but it is a relatively new technology!
  – only ~10 years of intense research
  – industrial quality TM implementations are much more recent!

• Feedback of software developers is essential:
  – to improve existing TM implementations
  – to focus research on truly relevant problems

• Try it out and report about your findings and experience
  – blog about it and let us know
  – measure performance for your code
  – report bugs in existing TM implementations!
Thanks for the attention

Q&A

eurotm@gsd.inesc-id.pt
http://www.eurotm.org