Dependence-Aware Transactional Memory for Increased Concurrency

Hany E. Ramadan, Christopher J. Rossbach, Emmett Witchel
University of Texas, Austin
Concurrency Conundrum

- Challenge: CMP ubiquity
  - Parallel programming with locks and threads is difficult
    - deadlock, livelock, convoys…
    - lock ordering, poor composability
    - performance-complexity tradeoff
  
- Transactional Memory (HTM)
  - simpler programming model
  - removes pitfalls of locking
  - coarse-grain transactions can perform well under low-contention
High contention: optimism warranted?

- **TM performs poorly with write-shared data**
  - Increasing core counts make this worse
- **Write sharing is common**
  - Statistics, reference counters, lists...
- **Solutions complicate programming model**
  - open-nesting, boosting, early release, ANTs...

<table>
<thead>
<tr>
<th></th>
<th>Locks</th>
<th>Transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Read-Sharing</strong></td>
<td>![Sad Face]</td>
<td>![Happy Face]</td>
</tr>
<tr>
<td><strong>Write-Sharing</strong></td>
<td>![Happy Face]</td>
<td>![Happy Face]</td>
</tr>
</tbody>
</table>

Dependence-Awareness can help the programmer (transparently!)
Outline

• Motivation
• Dependence-Aware Transactions
• Dependence-Aware Hardware
• Experimental Methodology/Evaluation
• Conclusion
Two threads sharing a counter

Initially: count == 0

Thread A:
load count, r
inc r
store r, count
...

Thread B:
load count, r
inc r
store r, count
...

Result: count == 2

Schedule:
• dynamic instruction sequence
• models concurrency by interleaving instructions from multiple threads

time
**TM shared counter**

Initially: count == 0

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>xA</strong></td>
<td><code>xbegin</code> load count,r inc r store r,count <code>xend</code></td>
</tr>
<tr>
<td><strong>xB</strong></td>
<td><code>xbegin</code> load count,r inc r store r,count <code>xend</code></td>
</tr>
</tbody>
</table>

Conflict: HTM designs cannot accept such schedules,
- both transactions access a datum, at least one is a write
- intersection between read and write sets of transactions

despite the fact that they yield the correct result!
Does this really matter?

**DATM:** use dependences to commit conflicting transactions

**Common Pattern!**
- Statistics
- Linked lists
- Garbage collectors
DATM shared counter

Initially: count == 0

\[
\begin{align*}
\text{xbegin} & \quad \text{T1} \\
\text{load count},r & \quad \text{load count},r \\
\text{inc } & \quad \text{inc } \\
\text{store } r,\text{count} & \quad \text{store } r,\text{count} \\
\text{xend} & \quad \text{xend}
\end{align*}
\]

- T1 must commit before T2
- If T1 aborts, T2 must also abort
- If T1 overwrites count, T2 must abort

Forward speculative data from T1 to satisfy T2's load.

Model these constraints as dependences
Conflicts become Dependences

Dependence Graph
- Transactions: nodes
- dependences: edges
- edges dictate commit order
- no cycles → conflict serializable

Write-Read:
- forward data
- overwrite → abort
- A commits before B

Write-Write:
- A commits before B

Read-Write:
- A commits before B

Tx A
R→W
write X
write Y
write Z
read X
read Z
...
...

Tx B
W→W
W→R
...

Write-Write:
- A commits before B
Enforcing ordering using Dependence Graphs

T1
\texttt{xbegin}  
load X,r0  
store r0,X  
\texttt{xend}

T2
\texttt{xbegin}  
load X,r0  
store r0,X  
\texttt{xend}  
Wait for T1

T1 must serialize before T2

Outstanding Dependences!
Enforcing Consistency using Dependence Graphs

- cycle $\rightarrow$ not conflict serializable
- restart some tx to break cycle
- invoke contention manager
Theoretical Foundation

Correctness and optimality
Proof: See [PPoPP 09]

<table>
<thead>
<tr>
<th>Dependence</th>
<th>Forward</th>
<th>Restart</th>
</tr>
</thead>
<tbody>
<tr>
<td>W₀ → W₁</td>
<td>No</td>
<td>If in cycle</td>
</tr>
<tr>
<td>R₀ → W₁</td>
<td>No</td>
<td>If in cycle</td>
</tr>
<tr>
<td>W₀ → R₁</td>
<td>Yes</td>
<td>If in cycle, and T₁ must if either: a) T₀ does. b) T₀ overwrites forwarded data with new value.</td>
</tr>
</tbody>
</table>
Outline

- Motivation/Background
- Dependence-Aware Model
- Dependence-Aware Hardware
- Experimental Methodology/Evaluation
- Conclusion
DATM Requirements

Mechanisms:
• Maintain global dependence graph
  – Conservative approximation
  – Detect cycles, enforce ordering constraints
• Create dependences
• Forwarding/receiving
• Buffer speculative data

Implementation:
coherence, L1, per-core HW structures
Dependence-Aware Microarchitecture

- **Order vector**: dependence graph
- **TXID**, access bits: versioning, detection
- **TXSW**: transaction status
- **Frc-ND + ND** bits: disable dependences

*These are not programmer-visible!*
Dependence-Aware Microarchitecture

Order vector:
- dependence graph
- topological sort
- conservative approx.

These are not programmer-visible!
Dependence-Aware Microarchitecture

- **Order vector**: dependence graph
- **TXID**, access bits: versioning, detection
- **TXSW**: transaction status
- **Frc-ND + ND** bits: disable dependences

*These are not programmer-visible!*
Dependence-Aware Microarchitecture

- **Order vector**: dependence graph
- **TXID**, access bits: versioning, detection
- **TXSW**: transaction status
- **Frc-ND + ND** bits: disable dependences, no active dependences

*These are not programmer-visible!*
FRMSI protocol: forwarding and receiving

• MSI states
• TX states (T*)
• Forwarded: (T*F)
• Received: (T*R*)
• Committing (CTM)
• Bus messages:
  • TXOVW
  • xABT
  • xCMT
FRMSI protocol: forwarding and receiving

- **MSI states**
  - TX states (T*)
  - Forwarded: (T*F)
  - Received: (T*R*)
  - Committing (CTM)
- **Bus messages:**
  - TXOVW
  - xABT
  - xCMT
FRMSI protocol: forwarding and receiving

- MSI states
- TxMSI states (T*)
  - Forwarded: (T*F)
  - Received: (T*R*)
  - Committing (CTM)
- Bus messages:
  - TXOVW
  - xABT
  - xCMT
FRMSI protocol: forwarding and receiving

- MSI states
- TX states (T*)
  - Forwarded: (T*F)
  - Received: (T*R*)
  - Committing (CTM)
- Bus messages:
  - TXOVW
  - xABT
  - xCMT
Converting Conflicts to Dependences

### Code Examples

1. `xbegin`
2. `ld R, cnt`
3. `inc cnt`
4. `st cnt, R`
5. `xend`

### Outstanding Dependence

The diagram illustrates how conflicts are converted to dependences. Core 0's instructions cause a conflict with core 1's instruction, leading to a stall in the execution due to the outstanding dependence.

### Memory State

- **Core 0**
  - `PC`: 1
  - `R`: 100
  - `TXID`: TxA
  - `L1 OVec`: [A, B]
  - `cnt`: 100

- **Core 1**
  - `PC`: 1
  - `R`: 102
  - `TXID`: TxB
  - `L1 OVec`: [A, B]
  - `cnt`: 102

### Main Memory

- `cnt`: 100

---

**Diagram Details**

- The diagram shows the core's PC, registers, TXID, and the OVec of the L1 cache.
- The `xCMT(A)` arrow indicates the conflict caused by core 0's instruction on core 1.
- The `Outstanding Dependence` label highlights the conflict between the two cores, leading to a stall in core 1's execution.

---

23
Outline

• Motivation/Background
• Dependence-Aware Model
• Dependence-Aware Hardware
• Experimental Methodology/Evaluation
• Conclusion
Experimental Setup

• Implemented DATM by extending MetaTM
  – Compared against MetaTM: [Ramadan 2007]

• Simulation environment
  – Simics 3.0.30 machine simulator
  – x86 ISA w/HTM instructions
  – 32k 4-way tx L1 cache; 4MB 4-way L2; 1GB RAM
  – 1 cycle/inst, 24 cyc/L2 hit, 350 cyc/main memory
  – 8, 16 processors

• Benchmarks
  – Micro-benchmarks
  – STAMP [Minh ISCA 2007]
  – TxLinux: Transactional OS [Rossbach SOSP 2007]
DATM speedup, 16 CPUs

- pmake: 3%
- labyrinth: 2%
- vacation: 22%
- bayes: 39%
- counter: 15x
Eliminating wasted work

**Speedup:**
- pmake: 2%
- labyrinth: 3%
- vacation: 22%
- bayes: 39%
- counter: 15x

Lower is better

16 CPUs

Normalized to MetaTM

- restarts/tx
- avg bkcyc/tx
Dependence Aware Contention Management

Timestamp contention management

T3 → T7
T2 → T3
T7 → T1
T4 → T7
T1 → T4

Order Vector: T2, T3, T7, T6, T4

5 transactions must restart!

Cascaded abort?
Contestion Management

![Bar chart showing speedup over timestamp policy for different policies: Polka, Eruption, and Dependence-Aware. The chart indicates a 101% speedup for Polka, a 14% speedup for Dependence-Aware, and a significant decrease for Eruption.](image-url)
Outline

• Motivation/Background
• Dependence-Aware Model
• Dependence-Aware Hardware
• Experimental Methodology/Evaluation
• Conclusion
Related Work

• HTM
  – TCC [Hammond 04], LogTM[-SE] [Moore 06], VTM [Rajwar 05], MetaTM [Ramadan 07, Rossbach 07], HASTM, PTM, HyTM, RTM/FlexTM [Shriraman 07,08]

• TLS
  – Hydra [Olukotun 99], Stampede [Steffan 98,00], Multiscalar [Sohi 95], [Garzaran 05], [Renau 05]

• TM Model extensions
  – Privatization [Spear 07], early release [Skare 06], escape actions [Zilles 06], open/closed nesting [Moss 85, Menon 07], Galois [Kulkarni 07], boosting [Herlihy 08], ANTs [Harris 07]

• TM + Conflict Serializability
  – Adaptive TSTM [Aydonat 08]
Conclusion

• DATM can commit conflicting transactions
• Improves write-sharing performance
• Transparent to the programmer
• DATM prototype demonstrates performance benefits

Source code available! www.metatm.net
Broadcast and Scalability

Yes.
We broadcast.
But it’s less than you might think…

<table>
<thead>
<tr>
<th>benchmark</th>
<th>tx</th>
<th>broadcast w</th>
<th>forw</th>
<th>rest</th>
</tr>
</thead>
<tbody>
<tr>
<td>bayes</td>
<td>762</td>
<td>1</td>
<td>0.4%</td>
<td></td>
</tr>
<tr>
<td>config(8p)</td>
<td>4698136</td>
<td>10,132</td>
<td>19.7%</td>
<td></td>
</tr>
<tr>
<td>counter</td>
<td>160000</td>
<td>0</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>counter-tt</td>
<td>16000</td>
<td>0</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>genome</td>
<td>352376</td>
<td>104</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>kmeans</td>
<td>436986</td>
<td>40,723</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>labyrinth</td>
<td>128</td>
<td>4</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>list(8p)</td>
<td>78586</td>
<td>86</td>
<td>3.3%</td>
<td></td>
</tr>
<tr>
<td>pmake(8p)</td>
<td>251844</td>
<td>10,009</td>
<td>12.9%</td>
<td></td>
</tr>
<tr>
<td>sasca2</td>
<td>47304</td>
<td>0</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>vacation</td>
<td>20000</td>
<td>143</td>
<td>0.4%</td>
<td></td>
</tr>
</tbody>
</table>

Because each node maintains all deps, this design uses broadcast for:
• Commit
• Abort
• TXOVW (broadcast writes)
• Forward restarts
• New Dependences

These sources of broadcast could be avoided in a directory protocol:
keep only the relevant subset of the dependence graph at each node
FRMSI Transient states

19 transient states: sources:
• forwarding
• overwrite of forwarded lines (TXOVW)
• transitions from MSI $\rightarrow$ $T^*$ states

Ameliorating factors
• best-effort: local evictions $\rightarrow$ abort, reduces WB states
• $T^*R^*$ states are isolated
• transitions back to MSI (abort, commit) require no transient states
• Signatures for forward/receive sets could eliminate five states.
Why isn’t this TLS?

<table>
<thead>
<tr>
<th>Feature</th>
<th>TLS</th>
<th>DATM</th>
<th>TM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward data between threads</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Detect when reads occur too early</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Discard speculative state after violations</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Retire speculative</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Writes in order</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Memory renaming</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multi-threaded workloads</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Programmer/Compiler transparency</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

1. Txns can commit in arbitrary order. TLS has the daunting task of maintaining program order for epochs
2. TLS forwards values when they are the globally committed value (i.e. through memory)
3. No need to detect “early” reads (before a forwardable value is produced)
4. Notion of “violation” is different—we must roll back when CS is violated, TLS, when epoch ordering
5. Retiring writes in order: similar issue—we use CTM state, don’t need speculation write buffers.
6. Memory renaming to avoid older threads seeing new values—we have no such issue
Doesn’t this lead to cascading aborts?

<table>
<thead>
<tr>
<th></th>
<th>WR deps</th>
<th>RW deps</th>
<th>forward restarts</th>
<th>cascade aborts</th>
</tr>
</thead>
<tbody>
<tr>
<td>bayes</td>
<td>3.8%</td>
<td>8.5%</td>
<td>0.4%</td>
<td>0%</td>
</tr>
<tr>
<td>config (8p)</td>
<td>0.3%</td>
<td>0.2%</td>
<td>19.7%</td>
<td>2.3%</td>
</tr>
<tr>
<td>counter</td>
<td>90.0%</td>
<td>80.7%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>counter-tt</td>
<td>99.9%</td>
<td>0.3%</td>
<td>0%</td>
<td>100.0%</td>
</tr>
<tr>
<td>genome</td>
<td>0.1%</td>
<td>0.1%</td>
<td>0%</td>
<td>14.2%</td>
</tr>
<tr>
<td>kmeans</td>
<td>8.9%</td>
<td>6.0%</td>
<td>0%</td>
<td>3.1%</td>
</tr>
<tr>
<td>labyrinth</td>
<td>32.0%</td>
<td>32.0%</td>
<td>0%</td>
<td>0.1%</td>
</tr>
<tr>
<td>list</td>
<td>14.3%</td>
<td>3.8%</td>
<td>3.3%</td>
<td>0.2%</td>
</tr>
<tr>
<td>pmake (8p)</td>
<td>0.5%</td>
<td>0.5%</td>
<td>12.9%</td>
<td>6.5%</td>
</tr>
<tr>
<td>ssca2</td>
<td>0.1%</td>
<td>0.1%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>vacation</td>
<td>35.2%</td>
<td>7.9%</td>
<td>0.4%</td>
<td>3.5%</td>
</tr>
</tbody>
</table>

Rare in most workloads
Inconsistent Reads

<table>
<thead>
<tr>
<th>in-cons. reads</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>bayes</td>
<td>3</td>
</tr>
<tr>
<td>config (8p)</td>
<td>1</td>
</tr>
<tr>
<td>counter</td>
<td>0</td>
</tr>
<tr>
<td>counter-tt</td>
<td>0</td>
</tr>
<tr>
<td>genome</td>
<td>1</td>
</tr>
<tr>
<td>kmeans</td>
<td>0</td>
</tr>
<tr>
<td>labyrinth</td>
<td>1</td>
</tr>
<tr>
<td>list</td>
<td>0</td>
</tr>
<tr>
<td>pmake (8p)</td>
<td>10</td>
</tr>
<tr>
<td>ssc2</td>
<td>0</td>
</tr>
<tr>
<td>vacation</td>
<td>34</td>
</tr>
</tbody>
</table>

OS modifications:
- Page fault handler
- Signal handler
Increasing Concurrency

Lazy/Lazy (e.g. TCC)
- T1 and T2 conflict at time t_c
- T1 arbitrates after T2
- Serialized execution

Eager/Eager (MetaTM, LogTM)
- T1 and T2 conflict at time t_c
- T1 arbitrates after T2
- Overlapped retry

DATM
- T1 and T2 conflict at time t_c
- T1 arbitrates after T2
- No retry!
Design space highlights

• Cache granularity:
  – eliminate per word access bits

• Timestamp-based dependences:
  – eliminate Order Vector

• Dependence-Aware contention management
Design Points

![Bar Chart]

- **Cache-gran**
  - bayes
  - vacation
  - counter
  - counter-tt

- **Timestamp-fwd**
  - bayes
  - vacation
  - counter
  - counter-tt

- **DATM**
  - bayes
  - vacation
  - counter
  - counter-tt

**Relative to MetaTM**

- Cache-gran: bayes > vacation > counter > counter-tt
- Timestamp-fwd: vacation > counter > bayes
- DATM: vacation > counter > bayes
Key Ideas:
- Critical sections execute concurrently
- Conflicts are detected dynamically
- If conflict serializability is violated, rollback

Key Abstractions:
- Primitives
  - `xbegin`, `xend`, `xretry`
- Conflict
  \[ \emptyset \neq \{W_a\} \cap \{R_b \cup W_b\} \]
- Contention Manager
  - Need flexible policy

“Conventional Wisdom Transactionalization”: Replace locks with transactions
Hardware TM basics: example

CPU 0
PC: 0
Working Set R{A,B,C} W{}

CPU 1
PC: 8
Working Set R{} W{}

0: xbegin
1: read A
2: read B
3: if(cpu % 2)
4:   write C
5: else
6:   read C
7: ...
8: xend

CONFLICT: C is in the read set of CPU 0, and in the write set of CPU 1
Assume contention manager decides CPU 1 wins.
CPU 0 rolls back
CPU 1 commits