The Common Case
Transactional Behavior of Multithreaded Programs

JaeWoong Chung
Hassan Chafi, Chi Cao Minh,
Austen McDonald, Brian D. Carstrom,
Christos Kozyrakis, Kunle Olukotun

Computer Systems Lab
Stanford University
http://tcc.stanford.edu
The Parallel Programming Problem

- CMPs are here but no parallel software to run on them

- Lock-based parallel programming is simply broken
  - Coarse-grained locks: serialization
  - Fine-grained locks: deadlocks, races, priority inversion, ...
  - Poor composability, not fault-tolerant, ...

- Transactional Memory (TM): an promising alternative
  - Transactions: atomic & isolated access to shared-memory
  - Performance through optimistic concurrency

- Parallel programming with TM
  - Coarse grain **Non-blocking synchronization** for parallel algorithms
  - **Speculative parallelization** for sequential algorithms
A transactional memory system provides
- Basics: versioning, conflict resolution, commit, abort
- Desired: nesting for libraries, virtualization

Several proposed designs
- Software-only: [DSTM], [OSTM], [ASTM], [SXM], [McRT-STM]
- Hardware-assisted: [TLR], [TCC], [U/LTM], [VTM], [LogTM]
- Hybrids: [HyTM], [Hybrid-TM]
- Different tradeoffs in implementing basic/desired features

Key questions
- Which is the common case to optimize for?
In Search of the Common Case

- **Key metrics of transactional program**
  - Transaction length
    - Cost of fixed overheads, time virtualization issues
  - Read-/write-set size
    - Buffer space requirements, buffer virtualization issues
  - Write-set to length ratio
    - Amortize commit/abort overheads
  - Frequency of nesting & I/O in transactions
    - Support for nesting, syscalls, …
  - Frequency of conflicts
    - Scheduling and contention management policies

- **The “chicken & egg problem”**
  - Programmers need efficient TM systems to support development
  - Designers need TM applications to derive common case

- **Can we break the deadlock?**
Paper Summary

- **Study the TM behavior of existing parallel programs**
  - Map existing parallel constructs to transactions
  - 36 applications, multiple domains, 4 programming models

- **Analyzed common case** for
  - Transaction length, read-/write-set size, write-set to length ratio, nesting & I/O
  - For both non-blocking synchronization & spec. parallelization
  - Implementation agnostic measurements

- **Derived guidelines** for TM system design
  - Buffering requirements and virtualization approach
  - Overhead amortization, nesting & I/O support, …
Methodology Overview

- **Key assumption**
  - The inherent parallelism & synchronization patterns are likely the same regardless of language primitives used

- **Methodology**
  1. Trace parallel application on existing hardware
  2. Map parallel constructs to transaction boundaries
     - E.g. lock/unlock -> transaction begin/end
  3. Process trace to analyze metrics

- **Measurements are agnostic to TM design**
  - Limitation: cannot measure violation behavior
## Parallel Applications

<table>
<thead>
<tr>
<th>Transaction Usage</th>
<th>Languages</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-blocking Synchronization</td>
<td>Java</td>
<td>MolDyn, MonteCarlo, RayTracer, Crypt, LUFact, Series, SOR, SparseMatmult, SPECjbb2000, PMD, HSQLDB</td>
</tr>
<tr>
<td></td>
<td>Pthread</td>
<td>Apache, Kingate, Bp-vision, Localize, Ultra Tic Tac Toe, MPEG2, AOL Server</td>
</tr>
<tr>
<td></td>
<td>ANL</td>
<td>Barnes, Mp3d, Ocean, Radix, FMM, Cholesky, Radiosity, FFT, Volrend, Water-N2, Water-Spatial</td>
</tr>
<tr>
<td>Speculative Parallelism</td>
<td>OpenMP</td>
<td>APPLU, Equake, Art, Swim, CG, BT, IS</td>
</tr>
</tbody>
</table>

- Different domains: scientific, enterprise, AI/robotics, multimedia
- Different qualities: highly optimized Vs. less optimized
- Java, Pthreads, ANL → studied for non-blocking synchronizations
- OpenMP → studied for speculative parallelization
Non-Blocking Synchronization

- Transactions are used for critical sections in parallel algorithms
- Original primitives mapped to transactional boundaries
  - E.g. Java synchronized block, pthread_mutex, ANL LOCK macro mapped transaction boundaries
- Semantics issue
  - To conserve the original program semantics, \textit{wait} splits transaction
  - This mapping is not always safe, but was fine in our study

<table>
<thead>
<tr>
<th>Original Threading Primitive</th>
<th>Transaction Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock</td>
<td>BEGIN</td>
</tr>
<tr>
<td>Unlock</td>
<td>END</td>
</tr>
<tr>
<td>Wait</td>
<td>END-BEGIN</td>
</tr>
</tbody>
</table>
### Transaction Length

- Number of instructions executed in transaction

<table>
<thead>
<tr>
<th>Application</th>
<th>Avg</th>
<th>50th %</th>
<th>95th %</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java average</td>
<td>5949</td>
<td>149</td>
<td>4256</td>
<td>13519488</td>
</tr>
<tr>
<td>Pthreads average</td>
<td>879</td>
<td>805</td>
<td>1056</td>
<td>22591</td>
</tr>
<tr>
<td>ANL average</td>
<td>256</td>
<td>114</td>
<td>772</td>
<td>16782</td>
</tr>
</tbody>
</table>

- Up to 95% of transactions have less than 5000 instructions
  => **Light-weight transactional primitives** are required

- Some programs have rare but long transactions
  => **Time virtualization** is needed (transaction context-switching)
**Time Virtualization for TM**

- **Interrupt and context-switch procedure**
  - **Interrupt**
    - Can wait?  
    - No: Is there a young xaction?  
    - Yes: Wait for a short xaction to finish; use its CPU for interrupt
    - No: Swap out xaction to VM; use its CPU for interrupt
  - Yes: Abort a young xaction; use its CPU for interrupt

- **Other proposals**

**Common case**

**Rare case!**

Handle in VM software [Satya94][Chen97]
Read-/Write-Set Size

- Bytes of data read/written by transaction

- 98% of transactions: <16KB read-set, <6KB write set
  => 32K L1 Cache will be enough for most transactions

- There are few very large transaction > 32K
  => space virtualization is needed but it’s better be cheap
Write-Set to Length Ratio

- Ratio of # unique addresses written to # instructions in transaction

- < 25% in most transactions
  
  => Big challenge for SW TM because of high per-write overhead
  
  => Even HW TM needs sufficient bandwidth for versioning and commit
Transaction Nesting and I/O

- Nesting occurs only in java VM code
  - 2.2 average depth
  => **Limited support** for nesting is sufficient for now

- I/O within transactions is rare
  - 27 applications have less than 0.1% of transactions with I/O
  - 8 applications have up to 1% of transactions with I/O
  - No transactions include both input and output
  => **Buffered I/O** would not deadlock
Speculative parallelization

- Speculatively parallelize loops in sequential algorithms
  - E.g. each loop iteration becomes a transactions

- This study
  - 6 loop based applications
  - Mapped outermost loop iteration to single transaction

<table>
<thead>
<tr>
<th>Original Threading Primitive</th>
<th>Transaction Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Outermost Iteration Start</td>
<td>BEGIN</td>
</tr>
<tr>
<td>Outermost Iteration End</td>
<td>END</td>
</tr>
</tbody>
</table>
Read-/Write-Set Size

- Bytes of data read/written by transaction

The read-/write-sets get larger up to L2-sized buffers (~128K)

=> They don’t fit in L1 cache but still fits into L2-sized buffer

=> **Inner loop parallelization** might be better to reduce buffer requirement
## Take-away Points

<table>
<thead>
<tr>
<th>Transaction Usage</th>
<th>Observation</th>
<th>TM Design Guidelines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-blocking</td>
<td>Short-lived transactions</td>
<td>Light-weight TM primitives</td>
</tr>
<tr>
<td>Blocking Synchroniza</td>
<td>Read-/write-sets &lt; 16K</td>
<td>L1 cache for versioning</td>
</tr>
<tr>
<td>High write-set to</td>
<td>High write-set to length ratio</td>
<td>Per write overhead is critical</td>
</tr>
<tr>
<td>Limited nesting</td>
<td>Few nested transactions</td>
<td>Challenge for STM</td>
</tr>
<tr>
<td>Limited nesting</td>
<td>Few transactions with I/O</td>
<td>Limited nesting support</td>
</tr>
<tr>
<td>Speculative</td>
<td>Few transactions with I/O</td>
<td>Buffered I/O</td>
</tr>
<tr>
<td>Parallelism</td>
<td>Large read-/write-sets</td>
<td>L2 cache for versioning</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summary

Extensive study of transactional behavior of programs
- 36 parallel applications from multiple domains
- Map existing parallel constructs to transactions
- Covered both non-blocking synchronization & speculative parallelization

Contributions
- **Quantitative Observations** on transactional characteristics
  - Most transactions are short-lived, small, and not nested
- **Design Guidelines** for Transactional Memory systems

Effective Guideline for TM Architects