Architectural Semantics for Practical Transactional Memory

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

Computer Systems Laboratory
Stanford University
http://tcc.stanford.edu
We Need Transactional Memory

- CMPs are here but their programming model is broken
  - Uniprocessors limited by power, complexity, wire latency…
  - Coarse- vs. fine-grained locks
    - serialization vs. deadlocks, races, and priority inversion
  - Poor composability, not fault-tolerant, …

- Transactional Memory (TM) systems are promising
  - Programmer-defined, atomic, isolated regions
  - Demonstrated performance potential
  - Many TM systems exist with different tradeoffs
    - [TRL], [TCC], [U/LTM], [VTM], [LogTM], [ASTM], [McRT], …
  - But we lack something…
TM Needs an Architecture

- A hardware/software interface
  - Unified semantic model for developers
    - Support transactional programming languages
    - Support common OS functionality
  - Enables fair evaluation of TM systems
  - Now we have “xbegin” and “xend”
    - Need more to implement real systems, compare designs, and evaluate tradeoffs
- Questions…
  - How does TM interact with library-based software?
  - How do we handle I/O & system calls within transactions?
  - How do we handle exceptions & contention within transactions?
  - How do we build implement TM programming languages?
Architectural Semantics for TM

- We define rich semantics for transactional memory
  - Thorough ISA-level specification of TM semantics
    - Applicable to all TM systems
  - Rich support for PL & OS functionality

- Our approach: identify three ISA primitives
  1. Two-phase commit
  2. Transactional handlers for commit/abort/violations
  3. Nested transactions (closed and open)

- PL & OS use primitives for higher level functionality
  - ISA provides primitives, but not end-to-end solutions
  - Software defines user-level API and other properties
Outline

- Motivation

- Architectural semantics for TM
  - Basic ISA-level primitives
  - ISA implementation (hardware & software)

- Implementation Overview
  - HW and SW components

- Examples and Evaluation
  - Example ISA uses
  - Performance analysis

- Conclusions
Two-phase Transaction Commit

- **Conventional**: monolithic commit in one step
  - Finalize validation (no conflicts)
  - Atomically commit the transaction write-set

- **New**: two-phase commit process
  - `xvalidate` finalizes validation, `xcommit` commits write-set
  - Other code can run in between two steps
    - Code is logically part of the transaction

- **Example uses**
  - Finalize I/O operations within transactions
  - Coordinate with other software for permission to commit
    - Correctness/security checkers, transaction synchronizers, …
Transactional Handlers

- **Conventional:** TM events processed by hardware
  - Commit: commit write-set and proceed with following code
  - Violation on conflict: rollback transaction and re-execute

- **New:** all TM events processed by software handlers
  - Fast, user-level handlers for commit, violation, and abort
  - Software can register multiple handlers per transaction
    - Stack of handlers maintained in software
  - Handlers have access to all transactional state
    - They decide what to commit or rollback, to re-execute or not, …

- **Example uses:**
  - Contention managers
  - I/O operations within transactions & conditional synchronization
  - Code for finalizing or compensating actions
Closed-nested Transactions

- Closed Nesting
  - Composable libraries
  - Performance improvement
  - Alternative control flow upon nested abort
Closed-nested Transactions

Closed-nested Semantics

\[
\begin{align*}
\text{xbegin} & \quad \text{T1's Read-Set} \\
\ldots & \quad \{\ldots, A\} \\
\text{xbegin} & \quad \text{T1's Write-Set} \\
\text{ld A} & \quad \{\ldots, B\} \\
\text{st B} & \\
\text{xvalidate}; \text{xcommit} & \quad \text{T2's Read-Set} \\
\text{xvalidate}; \text{xcommit} & \quad \text{T2's Write-Set} \\
\text{xcommit} & \\
\end{align*}
\]

Memory
Open-nested Transactions

- Open Nesting
  - Escape surrounding atomicity to update shared state
    - System calls
    - Communication between transactions/OS/scheduler/etc.
  - Performance improvements
  - Preserves atomicity unlike pause/non-transactional regions

\[
xbegin
\ldots
sbrk:\ \ldots
[\text{modify free list}]
\ldots
xvalidate; \ xcommit
\]

\[
xbegin\_open
\ldots
[\text{modify free list}]
\]
\[
xvalidate; \ xcommit
\]
Open-nested Transactions

Open-nested Semantics

\[ \text{xbegin} \]
\[ \ldots \]
\[ \text{xbegin\_open} \]

\[ \begin{array}{ll}
T1 & \text{ld A} \\
T2 & \text{st B} \\
\end{array} \]

\[ \text{xvalidate; xcommit} \]

\[ \text{xvalidate; xcommit} \]

T1’s Read-Set: \{ \ldots \}
T1’s Write-Set: \{ \ldots \}

T2’s Read-Set: \{ A \}
T2’s Write-Set: \{ B \}

Memory
Implementation Overview

- **Software**
  - Stack to track state and handlers
    - Like activation records for function calls
    - Works with nested transactions, multiple handlers per transaction
    - Handlers like user-level exceptions

- **Hardware**
  - A few new instructions & registers
    - Registers mostly for faster access of state logically in the stack
  - Modified cache design for nested transactions
    - Independent tracking of read-set and write-set
Transaction Stack

TCB Stack

TCB Frame 1

TCB Frame 2

TCB Frame 3

Transaction Control Block

Register Checkpoint

Read-set / Write-Set

Status Word

Commit Handler Code

Top Commit Handler

Base Commit Handler

Commit Handlers Stack

X1: Handler & Args

X1: Handler & Args

X1: Handler & Args

X1: Handler & Args

X1: Handler & Args

xbegin

... 

xbegin

...

xbegin

... 

xend

xend

xend

= in registers

= in cache / log

= in thread-private, cachable main memory

A. McDonald, ISCA’33, June 2006
Nesting Implementation

- Track multiple read-set and writes-sets in hardware
- Two Options: multi-tracking vs. associativity-based
  - Differences in cost of searching, committing, and merging
  - Multi-tracking best with eager versioning, associativity best with lazy
  - Both schemes benefit from lazy merging on commit
- Need virtualization to handle overflow
  - See our upcoming ASPLOS paper [Chung, et al.]
- See paper for further details

![MOESI Diagram](a)

![MOESI Diagram](b)
Example Use: Transactional I/O

\texttt{xbegin}

\texttt{write(buf, len):}

\texttt{\hspace{1em}register violation handler to de-alloc tmpBuf}
\texttt{\hspace{1em}alloc tmpBuf}
\texttt{\hspace{1em}cpy tmpBuf <- buf}

\texttt{\hspace{1em}push &tmpBuf, len; commit handler stack}
\texttt{\hspace{1em}push _writeCode; commit handler stack}

\texttt{xvalidate}

\texttt{pop _writeCode and args}
\texttt{run _writeCode}

\texttt{xcommit}
Example Use: Performance Tuning

- Single warehouse SPECjbb2000
  - One transaction per task
  - Order, payment, status, …
  - Irregular code with lots of concurrency

- On an 8-way TM CMP
  - Closed nesting: speedup 3.94
    - Nesting around B-tree updates to reduce violation cost
    - 2.0x over flattening
  - Open nesting: speedup 4.25
    - For unique order ID generation to reduce number of violations
    - 2.2x over flattening

- Similar results for other benchmarks

A. McDonald, ISCA’33, June 2006
Conclusions

- **Transactional memory must provide rich semantics**
  - Support common PL & OS features
  - Enable PL & OS research around transactions

- **This work**
  - Architectural specification of rich TM semantics
  - Three basic primitives
    - Two phase commit, transactional handlers, nested transactions
  - Hardware and software conventions for implementation
  - Demonstrated uses for rich functionality & performance
    - Implemented Atomos [Carlstrom, et al.] transactional programming language