# 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

```
xbegin
   lots_of_work()
   count++
xvalidate; xcommit
```

```
xbegin
...
  xbegin
    count++
  xvalidate; xcommit
xvalidate; xcommit
```

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

# Closed-nested Transactions

**Closed-nested Semantics**

T1

```
xbegin
. . .
   xbegin
      ld A
      st B
   xvalidate; xcommit
xvalidate; xcommit
```

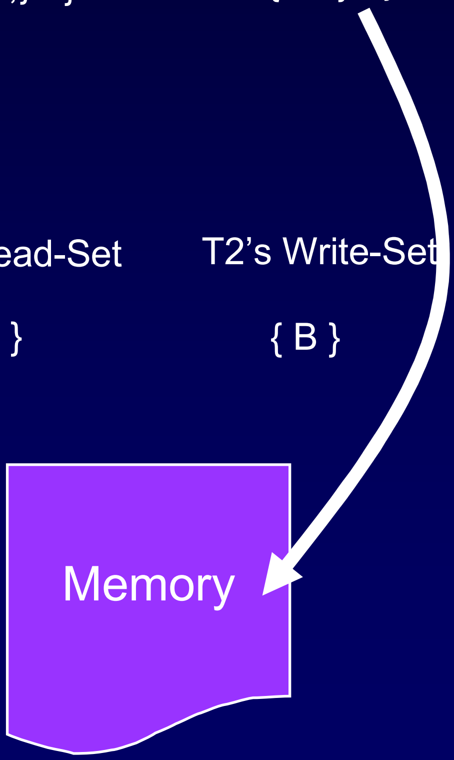
T2

T1's Read-Set { …, A }

T1's Write-Set { …, B }

T2's Read-Set { A }

T2's Write-Set { B }



# Open-nested Transactions

```
xbegin
...
sbrk: ...
        [modify free list]
...
xvalidate; xcommit
```

Shared OS state

→

```
xbegin
...
sbrk:
    xbegin_open
      ...
      [modify free list]
    xvalidate; xcommit
...
xvalidate; xcommit
```

- 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

# Open-nested Transactions

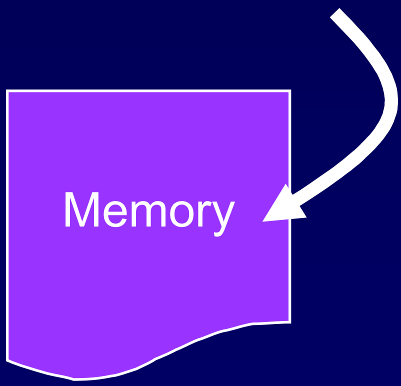
**Open-nested Semantics**

T1

```
xbegin
  ...
  xbegin_open      (T2)
     ld A
     st B
  xvalidate; xcommit
xvalidate; xcommit
```

| T1's Read-Set | T1's Write-Set |
|---|---|
| { … } | { … } |

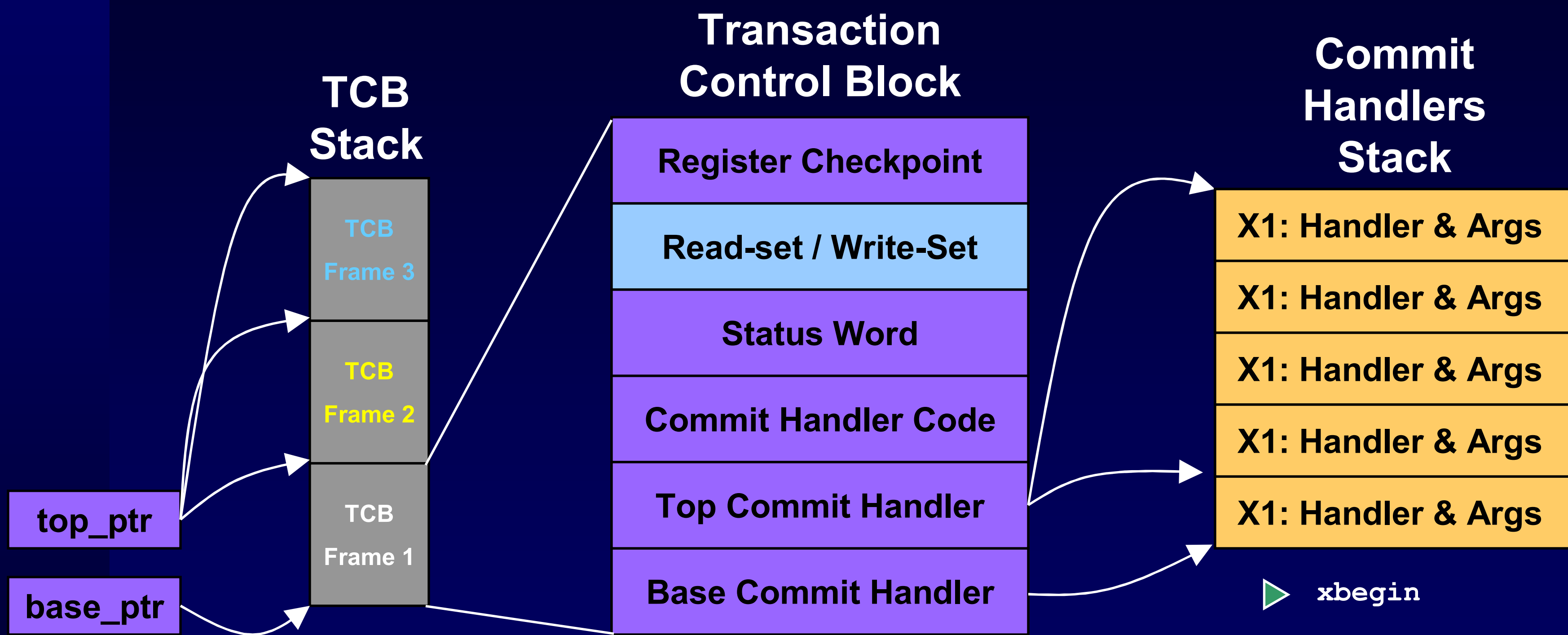
| T2's Read-Set | T2's Write-Set |
|---|---|
| { A } | { B } |



# 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




```
xbegin
...
   xbegin
      ...
      xbegin
         ...
      xend
   xend
xend
```

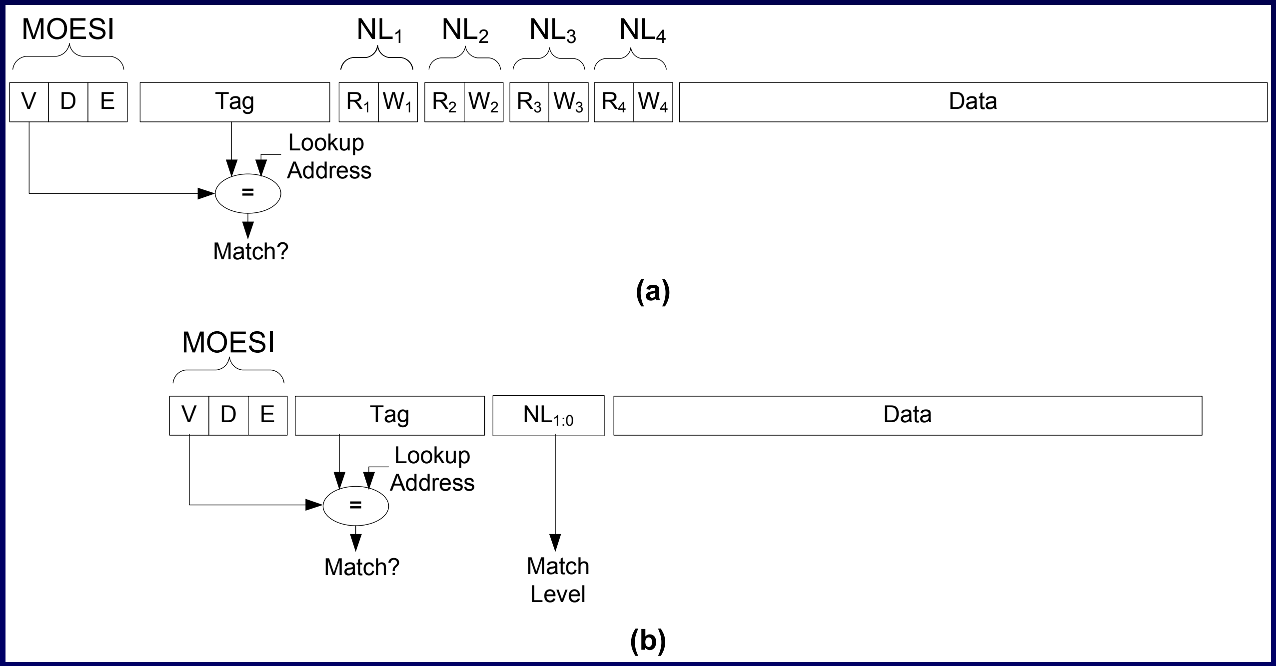
= in cache / log

= in registers

= in thread-private, cachable main memory

# 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



# Example Use: Transactional I/O

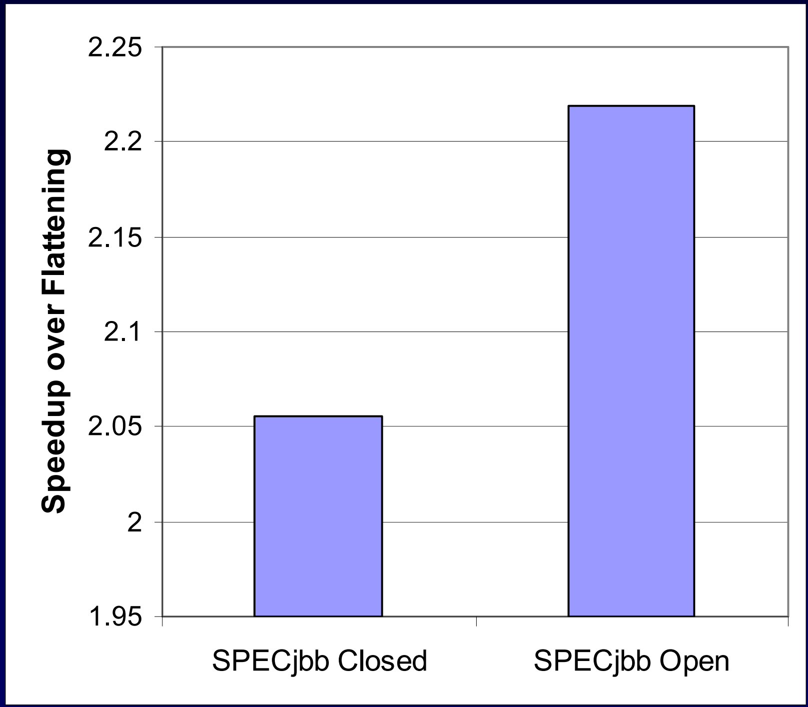
```
xbegin
  write(buf, len):
    register violation handler to de-alloc tmpBuf
    alloc tmpBuf
    cpy tmpBuf <- buf

    push &tmpBuf, len; commit handler stack
    push _writeCode; commit handler stack

xvalidate
  pop _writeCode and args
  run _writeCode
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



# 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