Testing Implementations of Transactional Memory

Chaiyasit Manovit (*Sun Microsystems)*

Sudheendra Hangal (*Magic Lamp Software)*

Hassan Chafi (*Stanford University)*
Austen McDonald
Christos Kozyrakis
Kunle Olukotun

Overview

- Transactional Memory
- Formal Specification of a Generic TM
- Our Testing Methodology
- Results
- Summary

Transactional Memory

- Multiprocessors becoming norm
- Parallel programs difficult with locks
- Motivating Example = Hash table
  - 1 lock = simple but no concurrency
  - Many locks = space overhead
- TM system guarantees atomicity (+isolation)
  - Simpler with same or better performance

Flavors of TM

- Implementations
  - Hardware/Software/Hybrid
  - Conflict detection
  - Version management
  - Contention management
- Features
  - Granularity, e.g., location-based vs. object-based
  - Nested transactions
  - Non-transactional mem ops
- etc.

Formal Specification of a Generic TM

- A Generic TM
  - Granularity = location-based
  - Closed nesting transactions
  - Permits non-transactional mem ops
  - No reordering of transactions
  - Aborted transactions are invisible
- Formal Specification
  - Axioms describing *perceived* serialization order of committed mem ops and load values – similar to Sindhu et al [SFC91]
  - Axioms may appear restrictive (capture functionality, not implementations/optimizations)

Transaction Semantics

- Notation
  - Op memory operation
  - [ ] transaction boundary
  - ; program order (per thread/processor)
  - $\leq$ memory order (the *perceived* serialization order, global)
  - Properties: Transitive: $a \leq b \wedge b \leq c \Rightarrow a \leq c$
    Anti-symmetric: $a \leq b \Rightarrow \neg (b \leq a)$
- Axioms

TransOpOp: $[Op_1;Op_2] \Rightarrow Op_1 \leq Op_2$

TransMembar: $Op_1 ; [Op_2] \Rightarrow Op_1 \leq Op_2$

$[Op_1] ; Op_2 \Rightarrow Op_1 \leq Op_2$

TransAtomicity: $[Op_1;Op_2] \wedge \neg[Op_1;Op;Op_2] \Rightarrow (Op \leq Op_1) \vee (Op_2 \leq Op)$

Our Testing Methodology: TSOtool [HVM04]

- Three steps
  - Generate pseudo-random test programs with data races
    - Execution results depend on race resolutions
    - More aggressive than test programs with predictable results
  - Execute on system under test
    - Only observe load values, no instrumentation to observe race resolutions
  - Analyze results for memory consistency
    - Determine if the observed results are allowed by the memory model
    - Key step
- Aggressive test programs help expose corner cases

Analysis

- Input: execution trace + load values
- Assumption: stores write distinct values
- Graph-based
  - Node = Op
  - Edge = $\leq$
  - Goal = find a valid serialization order = total order
  - Cycle = "$\leq$" not anti-symmetric = mem model violation
- NP-complete problem [Pap79, GK94]

Analysis: Baseline (P Algorithm)

- Goal = find as many order as we can = partial order
- Incomplete analysis
- Rules for adding edges
  - Static : determined from program order
    e.g. $[Op_1;Op_2] \Rightarrow Op_1 \leq Op_2$
  - Observed : determined from the load values
    e.g. $Val[S]=Val[L] \Rightarrow S \leq L$ (for SC & transactional)
  - Inferred : inferred from known mem order & transitive closures
    e.g. $S \leq S' \Rightarrow L \leq S'$ (for same location & $Val[S]=Val[L]$)
  - Iterate until no more inferred edges
  - Enforce *TransAtomicity* at all time
- Loose bound = $O(n^5)$

Analysis: Deriv+Back

- Goal = find a valid total order
- Complete analysis
- Baseline + Topological sort, with backtracking

Results

- Stanford TCC [HWC04]
  - Transactional only
  - Flattening nested transactions
  - Compiler + APIs for C/Java
  - Two implementations
    - TCC-A = Small scale with bus-based protocol
    - TCC-B = Large scale with directory-based protocol
- Found 1 bug in TCC-A, 2+ bugs in TCC-B

TCC-A Bug

P1

```
...

// X1
A = 1;
TCC_Commit();

// X2
x = A;
y = A;
TCC_Commit();

...
```

P2

```
...

// X3
A = 2;
TCC_Commit();

...
```

Execution result: `x=1, y=2`

TransAtomicity violation: `x` and `y` should read same value!

`TCC_Commit()` needs to clobber memory.

TCC-B Bug

P1

```
...

// X1
A = 1;
TCC_Commit();

// X2
B = 12;
A = 2;
TCC_Commit();

...
```

P2

```
...

// X3
x = A;
y = B;
TCC_Commit();

...
```

E1 (X1 TCC_Commit() → X2), E2 (A = 1 → x = A), E3 (B = 12 → y = B), E4 (x = A → A = 2)

E1: Program order

E2: $Val[S]=Val[L] \Rightarrow S \leq L$

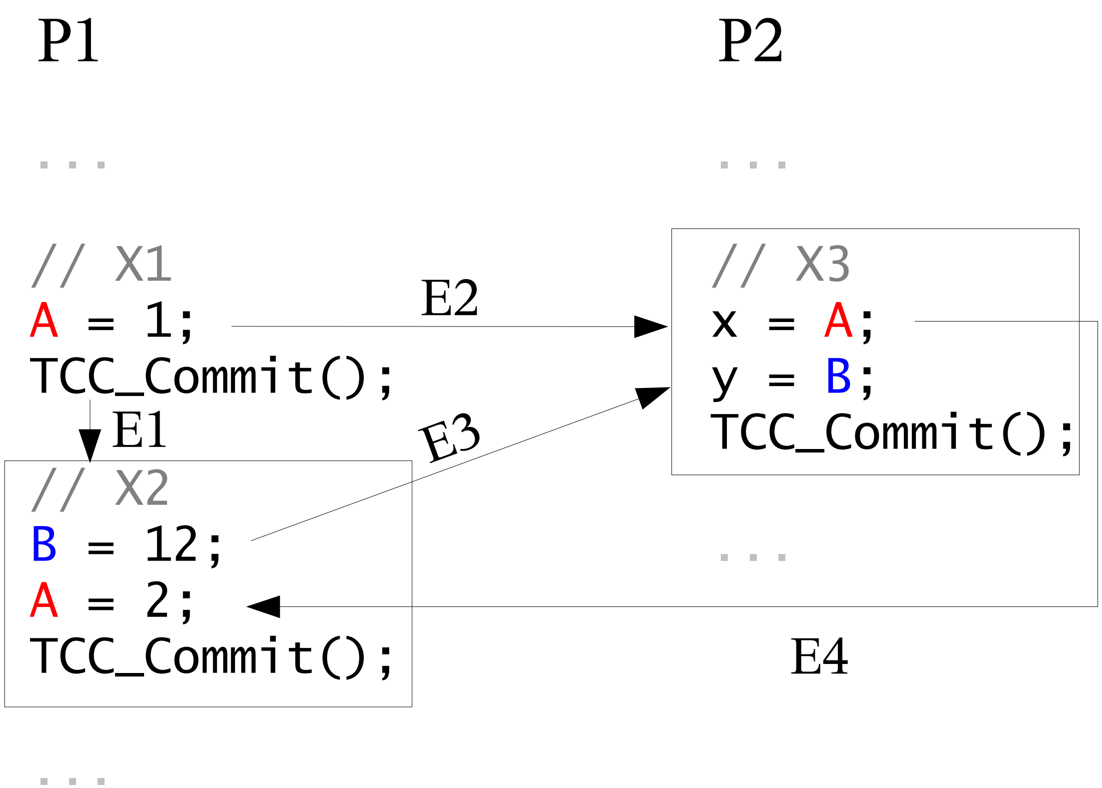
E3: $Val[S]=Val[L] \Rightarrow S \leq L$

E4: $S \leq S' \Rightarrow L \leq S'$

Execution result: `x=1, y=12`

No cycles yet. OK if non-transactional.

TCC-B Bug



E1: Program order

E2: $Val[S]=Val[L] \Rightarrow S \leq L$

E3: $Val[S]=Val[L] \Rightarrow S \leq L$

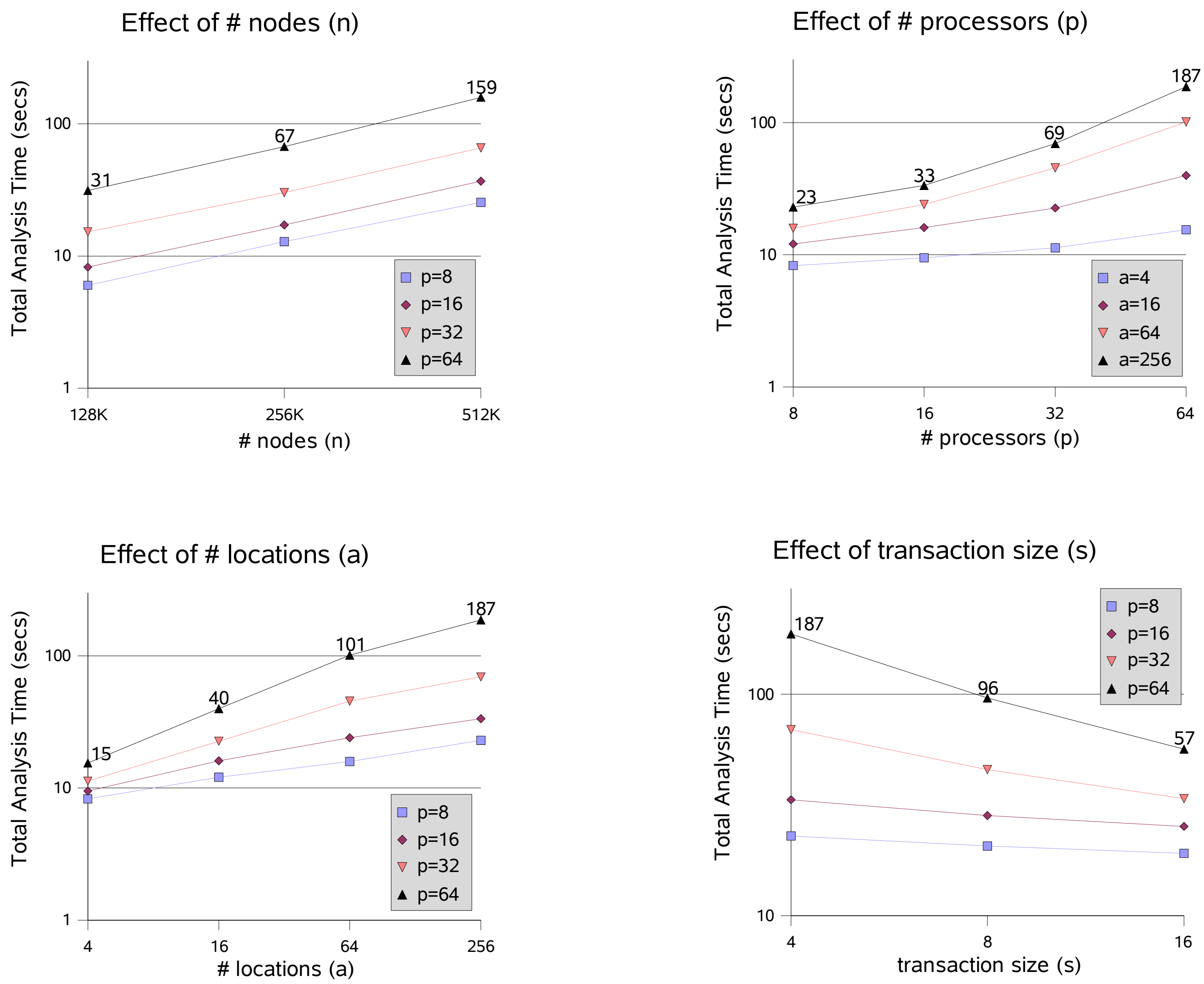
E4: $S \leq S' \Rightarrow L \leq S'$

Execution result: `x=1, y=12`

No cycles yet. OK if non-transactional.

A cycle is formed with *TransAtomicity*.

Analysis Time



Summary

- Extended work for conventional to transactional
  - Axiomatic framework
  - Testing methodology
  - Complete analysis in reasonable time
- Found bugs in a “real” design