Implementing and Evaluating a Model Checker for TM Systems

Woongki Baek, Nathan Bronson,
Christos Kozyrakis, Kunle Olukotun
wkbaek@stanford.edu

Stanford University

Introduction

- Transactional Memory (TM) simplifies parallel programming
  - User-specified "transactions" run in an atomic and isolated way
  - TM provides correctness and liveness guarantees
- Performance critical: subtle but fast TM implementations are favored
  - Vulnerable to correctness bugs
  - The resulting systems become difficult to prove correctness
    - Many TMs are used without any formal correctness guarantees
- A few recent works attempted to model check TMs
  - [PLDI'08]
    - An important reduction theorem: 2 threads, 2 variables, …
    - Model checked the abstract models of several STMs including TL2
  - [ICDCS'09]
    - Model checked Intel's McRT STM

Limitations of Previous Works

- Too "abstracted" models
  - E.g., timestamp-based version control of TL2 is not modeled [PLDI'08]
    - Committing transactions invalidate other conflicting transactions
  - Need a proof that "abstract model" == "actual implementation"
    - Otherwise, correctness of the evaluated TM still remains ***unchecked***
- Lack of use-cases of model checking for a wider range of TM systems
  - E.g., No previous study on hybrid TMs or nested TMs
- Lack of modeling both txn and non-txn memory operations
  - To investigate subtle correctness issues with weak isolation
- Lack of an in-depth quantitative analysis to understand practical issues
  - E.g., Sensitivity of the state space to various system parameters

Contributions of This Work

- Proposing ChkTM:
  - Flexible model checker for TMs
    - ***TL2: a timestamp-based, high-performance STM***
    - SigTM: a hybrid TM that accelerates an STM using hardware sigs
    - NesTM: an STM that supports nested parallel transactions
  - Model STMs close to the implementation level
    - E.g., timestamp-based version control is accurately modeled
- Using ChkTM:
  - Case study: found a subtle correctness bug in the current TL2 code
  - Verify the correctness of TL2 and SigTM
  - Provide an in-depth quantitative analysis on ChkTM

Outline

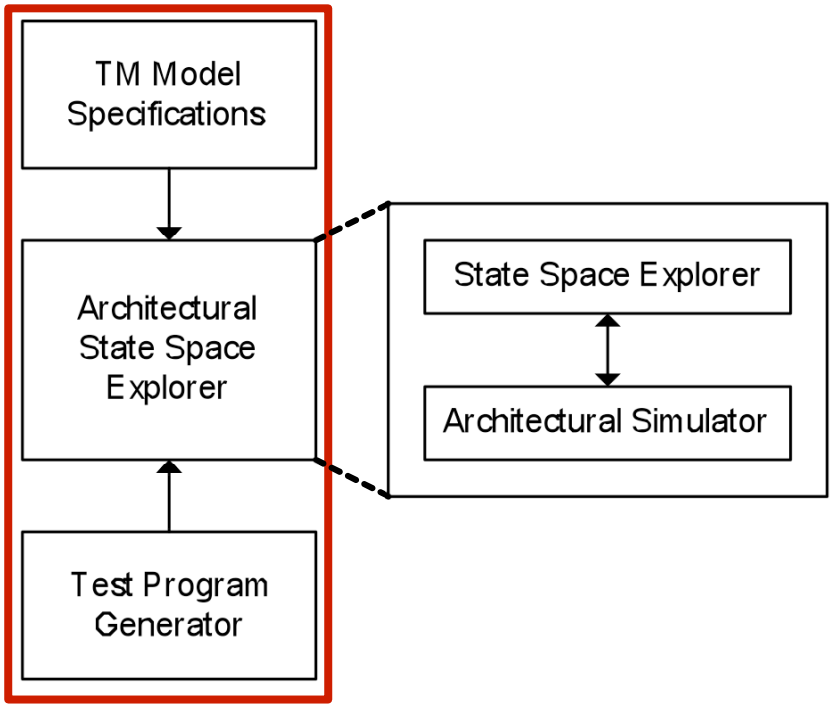
- Introduction
- Background
- Design and Implementation of ChkTM
- Case Study: Debugging Eager TL2
- Evaluation
- Conclusions

Background

- Correctness criterion: conflict serializability
  - Conflict equivalence: same order of every pair of conflicting op's
  - Conflict-serializable: conflict-equivalent to a serial schedule

- TL2 (STM)
  - A global version clock is used to establish serializability
  - Each memory loc. is associated with a version-owner lock (voLock)
  - On commit, each transaction validates its read set
    - Checking all the voLocks in the read set
    - ***Success*** ➔ updates are visible to others, ***Fail*** ➔ updates are discarded
  - Two data-versioning schemes
    - Lazy: buffers updates in write buffers until the commit time
    - Eager: performs in-place updates (undo-logs hold previous values)

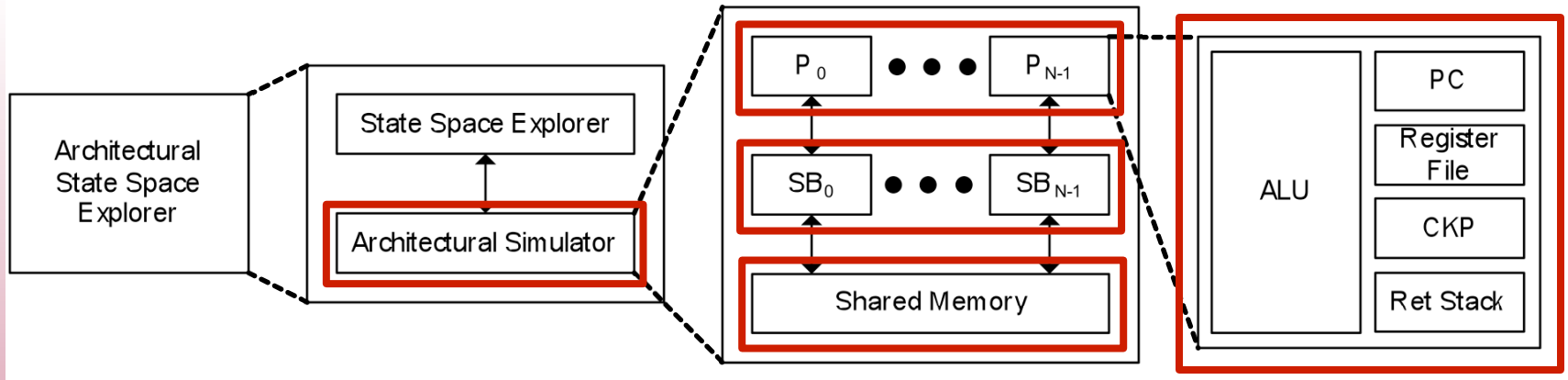
ChkTM: Overall Architecture



- The three components of ChkTM
  - Architectural state space explorer (ASE)
  - TM model specifications
  - Test program generator (see the paper)

- Implemented in Scala ➔ Concise implementation

ASE: Architectural Simulator



- Models a simple shared-memory multiprocessor system

- Processors
  - Model simple RISC processors with ALU, PC, registers, etc.

- Store buffers (SBs)
  - Every update to shared memory is made via a bounded SB
  - SB may retire stores in any order ➔ similar to SPARC's TSO
  - If SBS=0, the simulator emulates sequential consistency

- Shared memory
  - Consists of a fixed (configurable) number of shared memory words

ASE: State Space Explorer

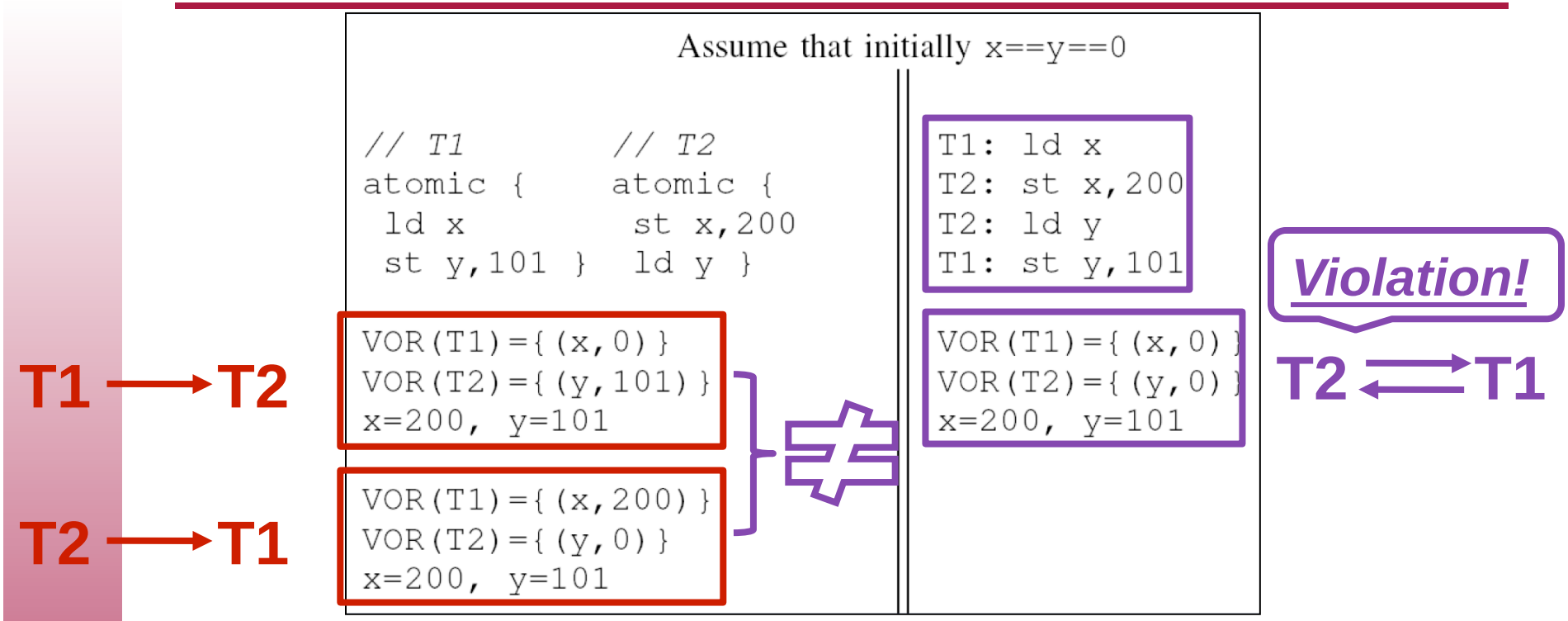
Globally Visible Variables					P_0 private		P_1 private	
m[0]	m[1]	m[2]	m[3]	• • •	pc[0]	• • •	pc[1]	• • •
77	11	10	42	• • •	100	• • •	200	• • •

- Architectural state
  - Describe the current state of the system using state variables
    - Processor-private: PC, SB, registers, …
    - Global: shared memory, …
- State transition
  - Dynamic executions of instructions generate new states
    - Instructions: load, store, branch, halt, ...
- BFS is performed to explore every possible interleaving of a program
  - Initial state: all the state variables (including PCs) are initialized
  - Terminal state: all the proc's are halted after executing a "halt" inst.

ASE: Verifying Serializability (1)

- First step: coarse-grain state space exploration (CSE)
  - Generate all "serial" schedules at ***transaction granularity***
    - Only a single processor is active at any time
    - The active processor cannot be changed while a transaction is active
  - Goal: to produce all the ***valid terminal states***
    - VOR: values observed by transactional reads
    - VOW: values overwritten by transactional writes
    - Final shared-memory state
  - Every transactional store in a test program writes a unique value
    - To establish one-to-one mapping between conflicting op's

ASE: Verifying Serializability (2)



- Second step: fine-grain state space exploration (FSE)
  - Explore every possible interleaving at ***instruction granularity***
  - Check every terminal state is identical to one of the valid terminals
    - If this check fails, ChkTM reports a serializability violation
  - Checking with VORs guarantee view-serializable schedules
    - VOWs are used to check conflict-serializable schedules (see the paper)

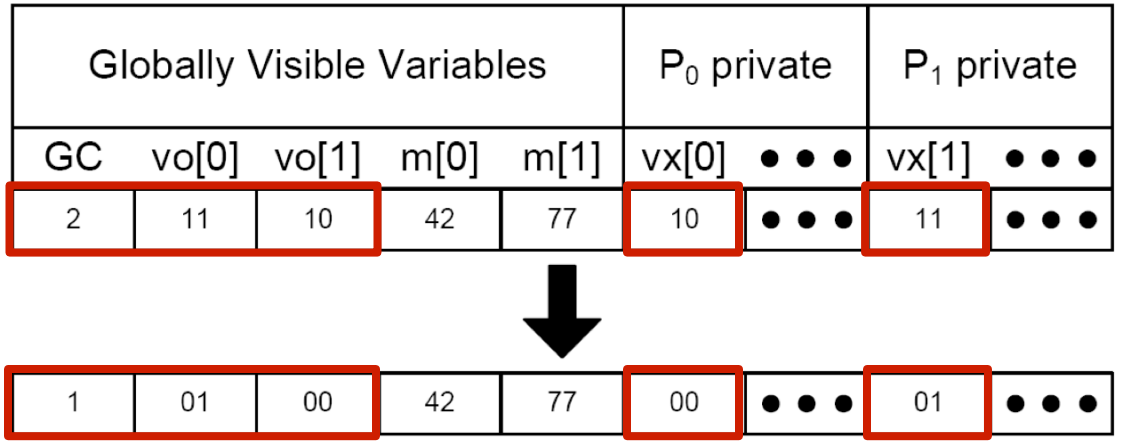
TM Model Specifications: TL2

```
long TxLoad(Self, addr)
{
  if (Self.WS.contains(addr)){
    val = Self.WS.lookup(addr);
    Self.RS.insert(addr);
    return val; }
  cv = getVo(addr);
  val = *addr;
  if (isLocked(cv) || extractTS(cv)>Self.rv ||
      cv!=getVo(addr)){
    TxAbort(Self);
  }
  Self.RS.insert(addr);
  return val;
}
```

```
// TxLoad, BX: addr, AX: return value
instrs = instrs ++ (List(
  (0 -> new Instr {
    nextPC=TxLoad+(if (ws.contains(bx)) 10 else 40) }),
  (10 -> new Instr {
    assign(AXKey(cpu), ws.apply(bx).wsVal) }),
  (20 -> new Instr {
    rs = rs + bx
    reads = (bx, ax) :: reads }),
  (30 -> new Ret),
  (40 -> new Instr { vx = read(VoLockKeys(bx)) }),
  (50 -> new Instr { ax = read(AppMemKeys(bx)) }),
  (60 -> new Instr {
    nextPC = (if ((extractOwner(vx) != -1) ||
    (extractTS(vx)>rv) || (vx!=read(VoLockKeys(bx))))
    TxAbort else (TxLoad + 70)) }),
  (70 -> new Instr {
    rs = rs + bx
    reads = (bx, ax) :: reads }),
  (80 -> new Ret),
).map(e => (e._1 + TxLoad, e._2)))
```

- Additional state variables to model TL2
  - E.g., R/W sets of transactions, global version clock, voLocks
- TM barriers are modeled close to the implementation level
  - Left: C-styled pseudocode of the lazy TL2 read barrier
  - Right: the ChkTM model of the lazy TL2 read barrier (in Scala)

TM Model Specifications: Timestamp Canonicalization



- The problem: state space explosion
  - An infinite # of states corresponding different timestamp values
- Our solution: timestamp canonicalization
  - Key idea: the relative ordering among timestamp values is important
    - But not the exact values
  - Canonicalize all the timestamp values in each step
    - 1: compute the set of all the timestamp values
    - 2: sort them
    - 3: replace each value with its ordinal position in the sorted set

Outline

- Introduction
- Background
- Design and Implementation of ChkTM
- Case Study: Debugging Eager TL2
- Evaluation
- Conclusions

Case Study: Debugging Eager TL2 (1)

Assume that initially x==y==0

```
// T1                    // T2
atomic {                 atomic {
  st x,1                   st y,2
  ld y }                   ld x }
```

Can VOR(T1)=={(y,2)} and VOR(T2)=={(x,1)}?

- We modeled the eager TL2 close to its current implementation
- With the test program above, ChkTM reported a serializability violation
  - VOR(T1)=={(y,2)}: T2 ➔ T1 (T2 precedes T1)
  - VOR(T2)=={(x,1)}: T1 ➔ T2
  - A cycle in the precedence graph ➔ Not a serializable schedule
- Current TL2 code is buggy ➔ How can we locate the bug using ChkTM?

Case Study: Debugging Eager TL2 (2)

```
    //T1:TxLoad                    //T2:TxStore,TxAbort
0:  cv=getVo(addr)                 ...
1:  ...                            lock(addr)
2:  ...                            *addr=2
3:  val=*addr // 2!                ...
4:  ...                            *addr=0
5:  ...                            unlock(addr)
6:  if(...cv!=getVo(addr)){        ...
7:    ... }                        ...
```

- ChkTM generates a counterexample shown above
  - Steps are not necessarily consecutive (some are skipped for brevity)
- T1 executing TxLoad, T2 executing TxStore and TxAbort
- Step 0: T1 samples the value of the voLock of "y" (addr == &y)
- Step 1: T2 sets the lock bit of the voLock of "y"
- Step 2: T2 "speculatively" updates "y" to 2

Case Study: Debugging Eager TL2 (3)

```
   //T1:TxLoad                    //T2:TxStore,TxAbort
0: cv=getVo(addr)                 ...
1: ...                            lock(addr)
2: ...                            *addr=2
3: val=*addr // 2!                ...
4: ...                            *addr=0
5: ...                            unlock(addr)
6: if(...cv!=getVo(addr)){        ...
7:   ... }                        ...
```

Abort! Incorrect!

- Step 3: T1 reads a "dirty" value (i.e., 2) of "y"
- Step 4: T2 restores the value of "y" to 0 (executing TxAbort)
- Step 5: T2 ***restores*** the voLock of "y" to the ***previous value***
- Step 6: T1 observes that "cv" matches the current value of voLock
- Step 7+: T1 continues (and commits) even after it read a "dirty" value
  - This is incorrect!

Case Study: Debugging Eager TL2 (4)

```
1:  procedure TXABORT
2:    rs.reset()
3:    for all addr in ws do
4:      Memory[addr] ← ws.lookup(addr)
5:    for all addr in ws do
6:      unlock(addr)          ▷ Timestamp value should have been incremented.
7:    ws.reset()
8:    doContentionManagement()
9:    restoreCheckpoint()
```

- ***Invalid-read bug***: Line 6 in TxAbort
  - On abort, voLocks in the write set are merely restored ➔ ***Wrong***
    - Timestamp values should have been incremented
  - Reported this bug to the TL2 developers

- Note: difficult to find this kind of subtle bugs using random tests
  - May increase the possibility by inserting random delays in the code
  - Require non-trivial intuition (where potential bugs would be)

Evaluation

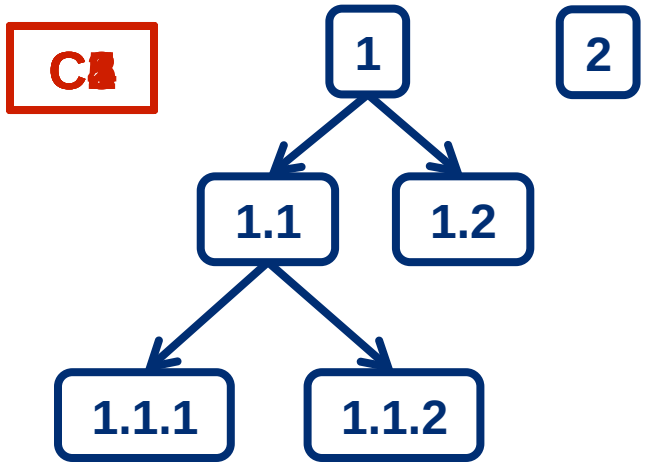
- Three issues to investigate
  - Correctness guarantees of TL2 and SigTM
    - Serializability / Strong isolation (refer to the paper)
  - Sensitivity of the state space to system parameters
    - E.g., number of threads
  - Tradeoff between state space and fidelity of approximate models
    - Refer to the paper

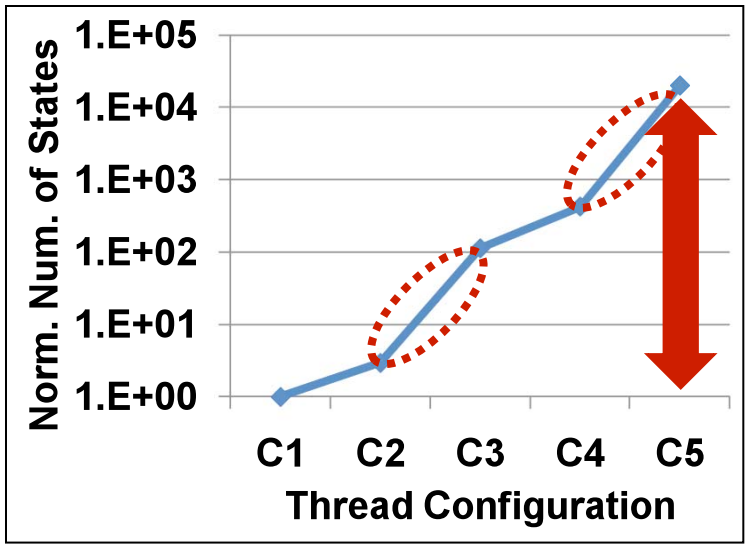
- Methodology
  - Processors: two quad-core 2.33GHz Intel Xeon CPUs
  - Memory: 32GB
  - OS: Linux x86_64 kernel 2.6.18
  - JVM: the 64-bit Server VM in Sun's JAVA JRE (build: 1.6.0-14-b08)
  - Scala: compiler version 2.7.5

Correctness Results: Serializability

- Generated all the possible test programs where:
  - Two threads in each program
  - One transaction per thread
  - At most three txn. memory op's (read or write) per transaction
    - Shared memory: two shared-memory words
  - This configuration was inspired by the approach in [ICDCS'09]
- Ran all the generated test programs on TL2 and SigTM models
  - ChkTM did not report any serializability violation
  - It took up to 4 hours to verify each TM system
- Thus, we make the following statement:
  - "*TL2 and SigTM (both lazy and eager) guarantee the serializability of every possible execution of every possible program that runs two threads, each of which executes one transaction that performs no more than three transactional memory operations*"

Sensitivity Results: Number of Threads





- Thread configurations
  - C1={1,2}, C2=C1+{1.1}, C3=C2+{1.2}, C4=C3+{1.1.1}, …
- Results
  - Ran a test program where each txn only performs 2 reads on NesTM
  - State space explosively grows when a new sibling is added
    - E.g., C2 → C3, C4 → C5
    - No predefined ordering between siblings ➔ more possible interleavings
  - State space with C5 is 20,000x larger than C1
    - Clearly motivate the need for a reduction theorem for nested TM

Conclusions

- Propose ChkTM, a flexible model checker for TM systems
  - TL2 (STM), SigTM (Hybrid), and NesTM (nested STM) are modeled
  - STMs are modeled close to the implementation
    - Including the timestamp mechanism using our canonicalization tech.
- Present a case study in which a bug in eager TL2 is revealed
- Model check the correctness of TL2 and SigTM
  - Serializability: guaranteed by both (at least for small TM programs)
  - Strong isolation: no weak isolation anomaly was detected for SigTM
- Motivate the need for reduction theorem/techniques
  - Especially for nested TMs