Improving the Practicality of Transactional Memory

Woongki Baek

Electrical Engineering
Stanford University

Programming Multiprocessors

- Multiprocessor systems are now everywhere
  - From embedded to datacenter systems
- Scalable performance requires parallel programming
- Parallel programming significantly increases complexity
  - Synchronization is required to avoid data races
  - Current practice: lock-based synchronization
- Lock-based synchronization is hard
  - Option #1: coarse-grain locking
    - Simplicity at less concurrency
  - Option #2: fine-grain locking
    - Better performance at higher complexity (e.g., deadlock)

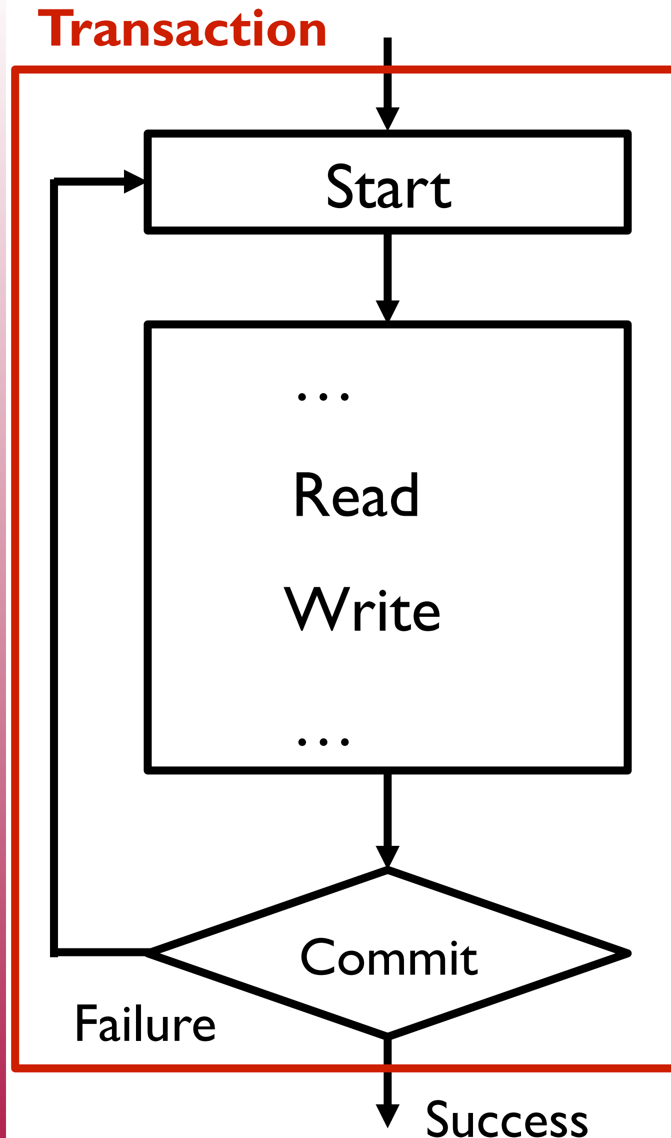
Transactional Memory (TM)

- Transaction
  - An atomic and isolated sequence of memory accesses
    - Atomicity: All or nothing
    - Isolation: Intermediate results are not visible
  - Code example

```
atomic {
  Tree1.remove(3);
  Tree2.insert(7);
}
```

- Key idea: Use transactions as synchronization primitives
  - Large atomic blocks simplify parallel programming

How Do Transactions Work?



- Start
- Speculative (optimistic) execution
- Build read and write sets
  - R/W or W/W conflict detection
- Commit
- Abort (Rollback) / Retry

Advantages of TM

- Provide an easy-to-use synchronization construct
  - As simple as coarse-grain locking
  - Programmer declares → System provides correctness/liveness
- Perform as well as fine-grain locking
  - Optimistic and fine-grain concurrency control
- Composability
  - Safe composition of software modules
  - Software modules synchronized with locks often cause deadlock

Challenge #1: Nested Parallelism

```
// Parallelize the outer loop
for(i=0;i<numCustomer;i++){
 atomic{
  // Can we parallelize the inner loop?
  for(j=0;j<numOrders;j++)
   processOrder(i,j,…);
 }}
```

- Nested parallelism is becoming more important
  - To fully utilize the increasing number of cores
- Current practice: Most TMs do not support nested parallelism
- ***What we need: TM with practical support for nested parallelism***
  - To achieve the best possible performance
  - To integrate with popular programming models (e.g., OpenMP, Cilk)

Challenge #2: Programmability

Sequential Code

```
void histogram(){
 for(i=0;i<N;i++){
  j=getIndex(i);
  bin[j]+=1;
 }
}
```

TM Code with Low-level API

```
void histogram(){
 for(i=0;i<P;i++){fork(work,…);}}
void work(){
 for(i=start;i<end;i++){
  TxStart();
  j=getIndex(i);
  val=TxLoad(&bin[j]);
  val+=1;
  TxStore(&bin[j],val);
  TxCommit(); }}
```

- Current practice: Using low-level TM API
- Programmability issues with low-level TM API
  - Synchronization: Manual instrumentation of TM barriers
    - Error-prone, limited portability & compiler optimizations
  - Thread management/scheduling: Intrusive code modifications
- ***What we need: Integrate TM into a high-level prog. model***

Challenge #3: Verification

- TM manages all concurrency control
  - If a TM is incorrect → All TM apps on it are potentially incorrect
- TM is performance critical → Subtle but fast implementation
  - Highly vulnerable to various correctness bugs
- Hand proof of the correctness of a TM system is hard
  - Many TM systems are used without formal correctness guarantees
- ***What we need: a TM verification environment***
  - Model TM systems close to the implementation level
  - Flexible to model various TM systems

Thesis Contributions

- Challenge #1: Nested Parallelism
  - NesTM [SPAA 10]
    - Extend a non-nested STM to support nested parallel transactions
  - FaNTM [ICS 10]
    - Practical hardware support to accelerate software nested transactions
- Challenge #2: Programmability
  - OpenTM [PACT 07]
    - Integrate TM into a high-level programming model (OpenMP + TM)
- Challenge #3: Verification
  - ChkTM [ICECCS 10]
    - A flexible model checking environment for TM systems

Outline

- ❑ Background & Motivation
- ❑ Challenge #1: Nested Parallelism
- ❑ Challenge #2: Programmability
- ❑ Challenge #3: Verification
- ❑ Conclusions

Challenge #1: Nested Parallelism

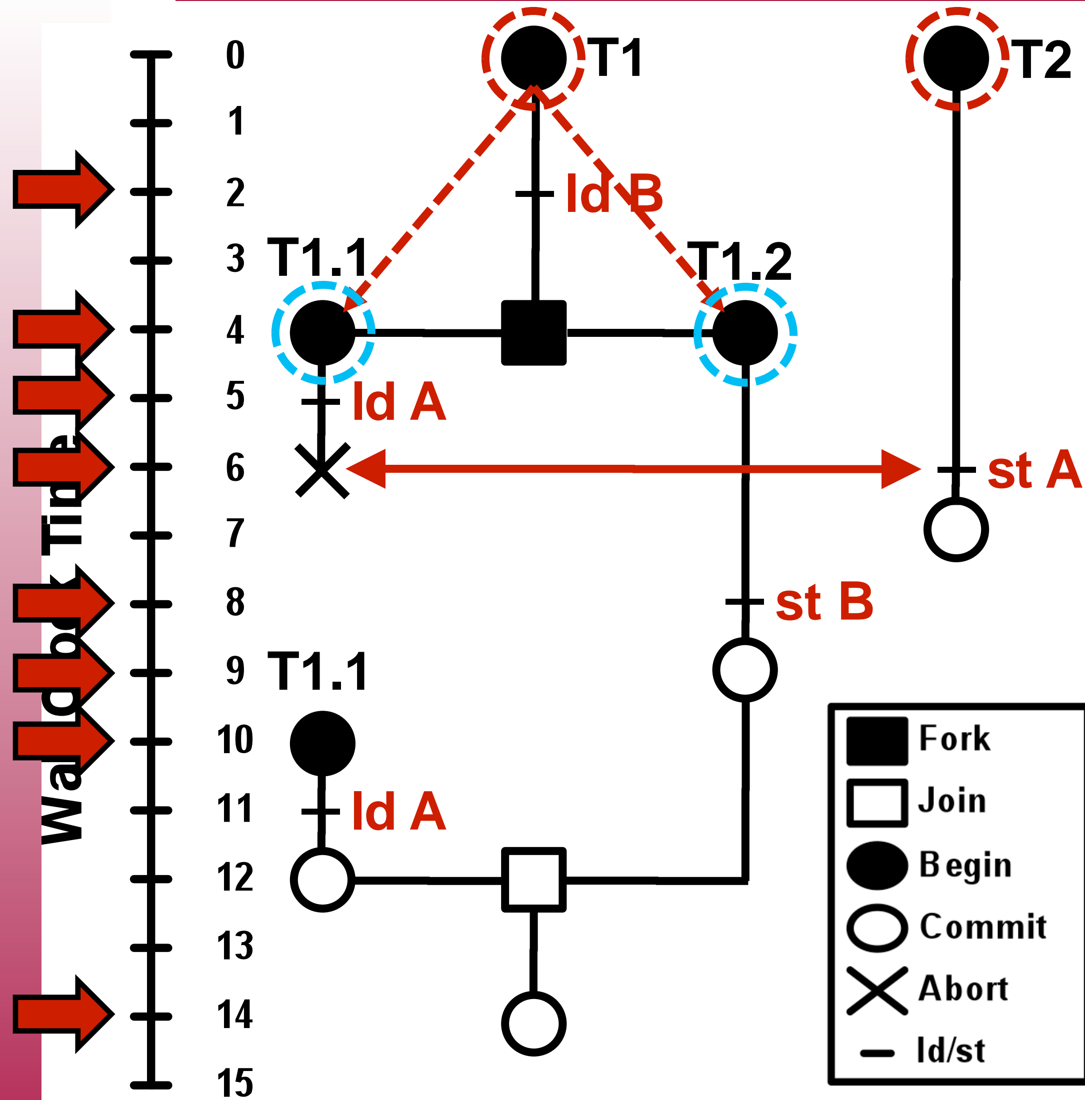
- What we need: TM with practical support for nested parallelism
  - To achieve the best possible performance
  - To integrate well with popular programming models

- My proposal: Filter-accelerated Nested TM (FaNTM) [ICS'10]
  - ***Goal: Make nested parallel transactions practical***
    - In terms of both performance and implementation cost
  - Eliminate excessive runtime overheads of SW nested transactions
  - Simplify hardware by decoupling nested transactions from caches

Semantics of Concurrent Nesting

- Definitions
  - Family(T) = ancestors(T) $\cup$ descendants(T)
    - Transactional hierarchy has a tree structure
  - Readers(o): a set of active transactions that read “o”
  - Writers(o): a set of active transactions that wrote to “o”
- Conflicts
  - T reads from “o”: R/W conflict
    - If there exists T’ such that T’ $\in$ writers(o), T’≠T, and T’ $\notin$ ancestors(T)
  - T writes to “o”: R/W or W/W conflict
    - If there exists T’ such that T’ $\in$ readers(o) $\cup$ writers(o), T’≠T, and T’ $\notin$ ancestors(T)

Example of Concurrent Nesting

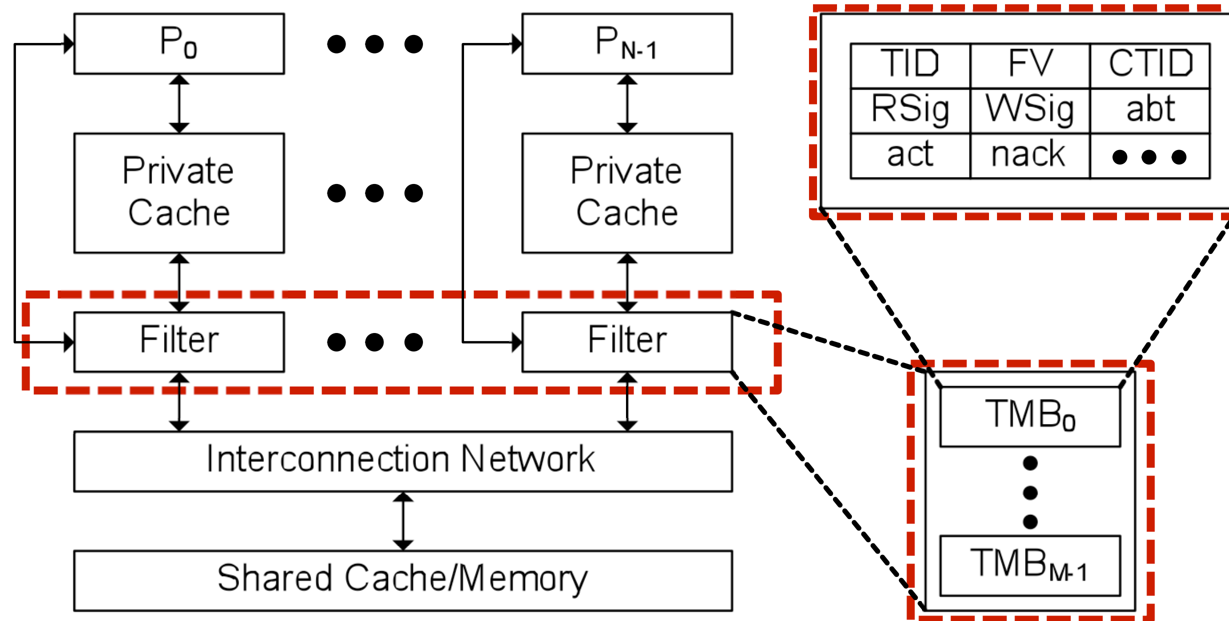


- T1 and T2 are top-level
  - T1.1, T1.2: T1's children
- T=6: R/W conflict
  - T2 writes to A
  - T1.1 [illegible] Readers(A)
  - T1.1 [illegible] Family(T2)
- T=8: No conflict
  - T1.2 writes to B
  - T1 [illegible] Readers(B)
  - T1 [illegible] Family(T1.2)
- Serialization order
  - **T2 → T1**

FaNTM Overview

- FaNTM is a hybrid TM that extends SigTM [Cao Minh 07]
  - Advantage: Decoupling txns from caches using HW signatures
    - No TM metadata in caches → Simplified HW
- Hardware extensions
  - Multiple sets of HW structures to map multiple txns per core
  - Network messages to remotely communicate signatures
- Software extensions
  - Additional metadata to maintain transactional hierarchy information
  - Extra code in TM barriers for concurrent nesting

Hardware: Overall Architecture



- Filters snoop coherence messages for nesting-aware conflict detection
  - Filters may intercept or propagate messages to caches
- Each filter consists of multiple Transactional Metadata Blocks (TMBs)
  - R/W Signatures: conservatively encoding R/W sets
  - FV: a bit vector encoding Family(T)

TMB: Conflict Detection (Ld)

Software: Commit Barrier

```
TxCommit(){
 if(topLevel()){
  resetTmMetaData();}
 else{
  mergeSigsToParent();
  mergeUndoLogToParent();
  resetTmMetaData();
 }
}
```

- If a top-level transaction
  - Finish by resetting TM metadata
- Otherwise (i.e., nested transaction)
  - Merge R/W signatures to its parent by sending messages over network
  - Merge its undo-log entries to its parent
  - Finish by resetting TM metadata

Evaluating FaNTM

- Three questions to investigate
  - Q1: What is the runtime overhead for top-level parallelism?
    - Used STAMP applications
    - Runtime overhead is small (2.3% on average across all apps)
    - Start/commit barriers are infrequently executed ➔ No major impact
  - Q2: What is the performance of nested parallel transactions?
  - Q3: How can we use nested parallelism to improve performance?

Q2: Performance of Nested Txns

Flat version

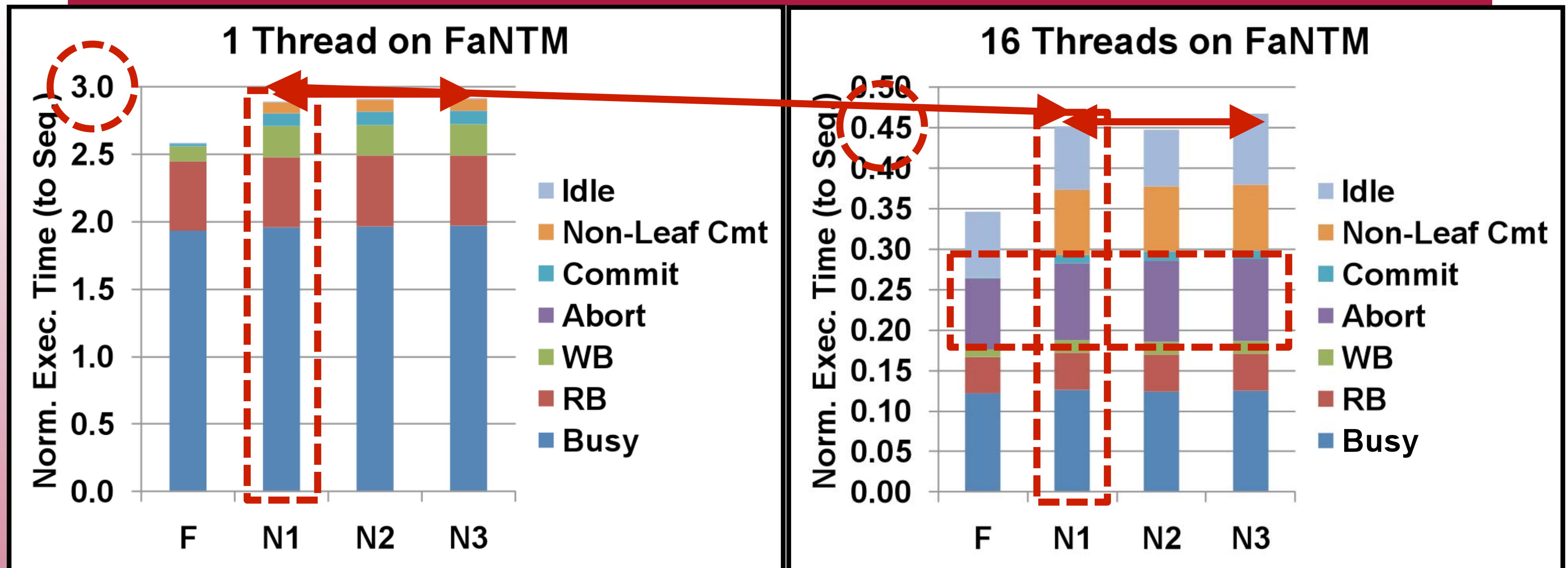
```
// Parallelize this loop
for(i=0;i<numOps;i+=C){
 atomic{
  for(j=0;j<C;j++){
   accessRBtree(i,j,…);}
 }
}
```

Nested version (N1)

```
atomic{
 // Parallelize this loop
 for(i=0;i<numOps;i+=C){
  atomic{
   for(j=0;j<C;j++){
    accessRBtree(i,j,…);}
  }}
}
```

- rbtree: perform operations on a concurrent RB tree
  - Two types of operations: Look-up (reads) / Insert (reads/writes)
- Sequential: sequentially perform operations
- Flat: Concurrently perform operations using top-level txns
- Nested: Repeatedly add outer transactions
  - N1, N2, and N3 versions

Q2: Performance of Nested Txns



- ❑ Scale up to 16 threads (e.g., N1 with 16 threads ➔ 6.5x faster)
  - Scalability is mainly limited by conflicts among transactions
- ❑ No performance degradation with deeper nesting
  - Conflict detection in HW ➔ No repeated validation across nesting
- ❑ Significantly faster (e.g., 12x) than NesTM (software-only)
  - ***Making nested parallel transactions practical***

Q3: Exploiting Nested Parallelism

Flat version

```
// Parallelize outer loop
for(i=0;i<numOps;i++){
 atomic{
  for(j=0;j<numTrees;j++){
   accessTree(i,j,…);
  }
 }
}
```

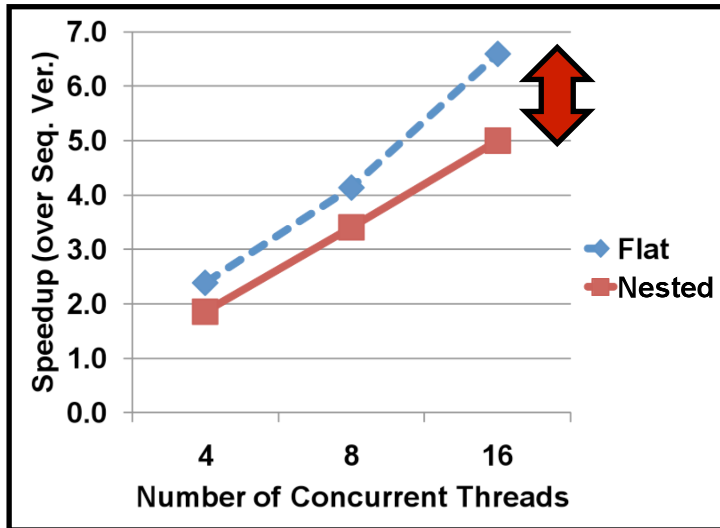
Nested version

```
// Parallelize outer loop
for(i=0;i<numOps;i++){
 atomic {
  // Parallelize inner loop
  for(j=0;j<numTrees;j++){
   atomic{
    accessTree(i,j,…);
   }}
 }}
```

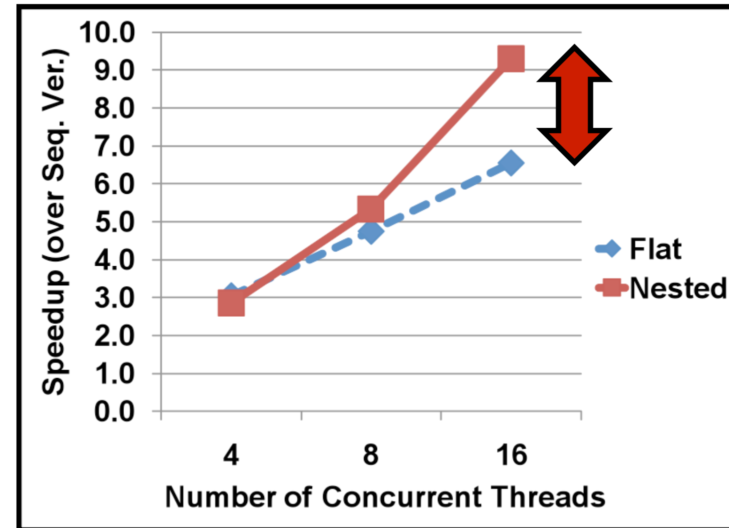
- np-rbtree: based on a data structure using multiple RB trees
  - Two types of operations: Look-up / Insert
    - Higher the percentage of inserts ➔ Higher contention (top-level txns)
  - After accessing each tree, computational work is performed
- Two ways to exploit the available parallelism
  - Flat version: outer-level parallelism
  - Nested version: inner- and outer-level parallelism

Q3: Flat vs. Nested

Lower-Cont/Small Work



Higher-Cont/Large Work



- Lower contention (top-level) & small work → ***Flat version*** is faster
  - Due to sufficient top-level parallelism & lower overheads
- Higher contention (top-level) & large work → ***Nested version*** is faster
  - By efficiently exploiting the parallelism available at both levels
- ***Motivate research on a nesting-aware runtime system***
  - Dynamically exploit nested parallelism

Outline

- ❑ Background & Motivation
- ❑ Challenge #1: Nested Parallelism
  - FaNTM makes nested parallel transactions practical
    - Eliminate excessive runtime overheads of software nested txns
    - Simplify hardware by decoupling nested txns from caches
- ❑ Challenge #2: Programmability
- ❑ Challenge #3: Verification
- ❑ Conclusions

Challenge #2: Programmability

- What we need
  - Integrate TM into a popular, high-level programming model

- My proposal: The OpenTM Transactional API [PACT 07]
  - ***Goal: Improve the programmability of TM***
    - Abstract low-level TM programming issues (e.g., barrier inst.)
    - Allow high-level management of TM programming options
  - Extend a popular programming model (OpenMP) to support TM
    - Language constructs: memory transactions, loops, sections
  - A prototype implementation
    - Provide portability across different TM platforms

OpenTM Transactions

- Define the boundary of a transaction
- Syntax: #pragma omp transaction [clauses] {structured-block}
  - nesting(open|closed) clause: specify the type of nesting
- Code example

OpenTM Code

```
void histogram(){
 #pragma omp parallel for
 for(i=0;i<N;i++){
  #pragma omp transaction {
   j=getIndex(i);
   bin[j]+=1;
  }
 }
}
```

TM Code with Low-level API

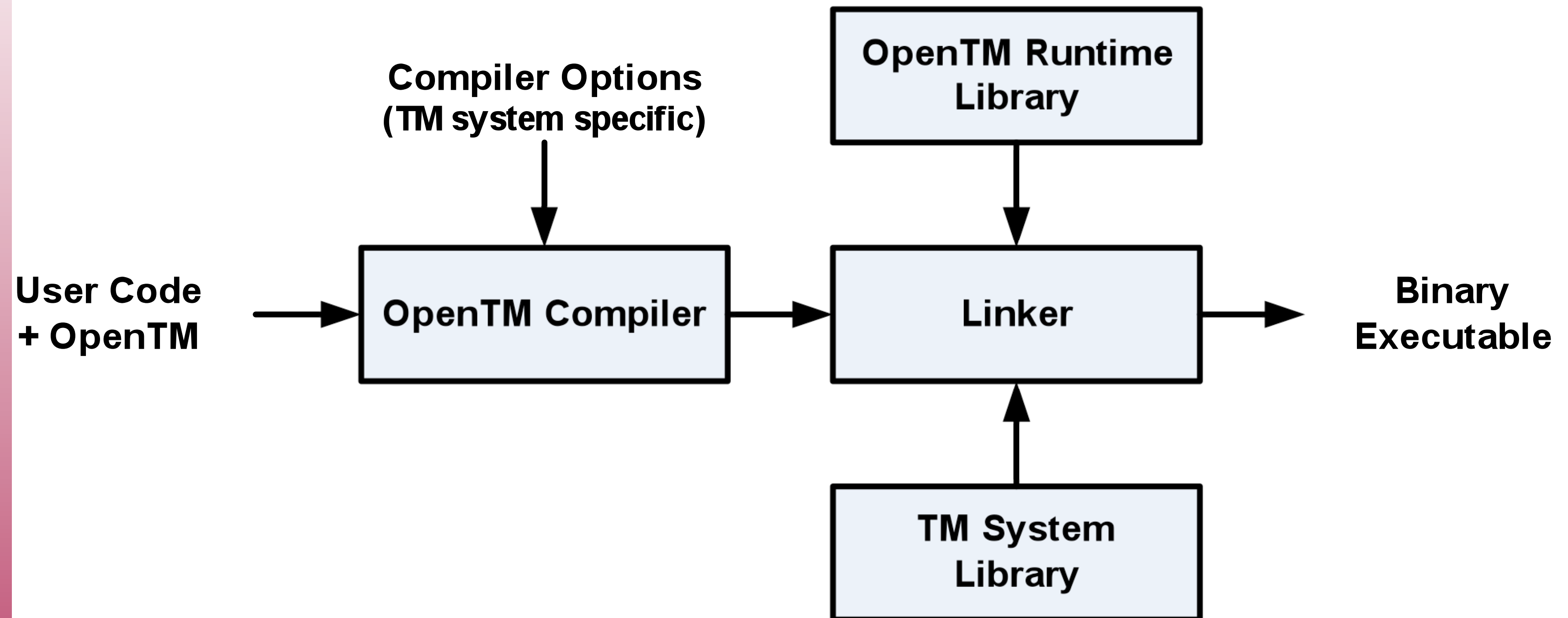
```
void histogram(){
 for(i=0;i<P;i++){fork(work,…);}}
void work(){
 for(i=start;i<end;i++){
  TxStart();
  j=getIndex(i);
  val=TxLoad(&bin[j]);
  val+=1;
  TxStore(&bin[j],val);
  TxCommit(); }}
```

OpenTM Transactional Loops

- Define a parallel loop with iterations running as transactions
- Syntax: #pragma omp transfor [clauses]
  - Schedule clause
    - Scheduling policy: static, dynamic, …
    - Loop chunk size (= transaction size)
- Code example

```
#pragma omp transfor schedule (dynamic, 3)
for (i=0; i<N; i++) {
  bin[A[i]] = bin[A[i]]+1;
}
```

OpenTM Code Generation

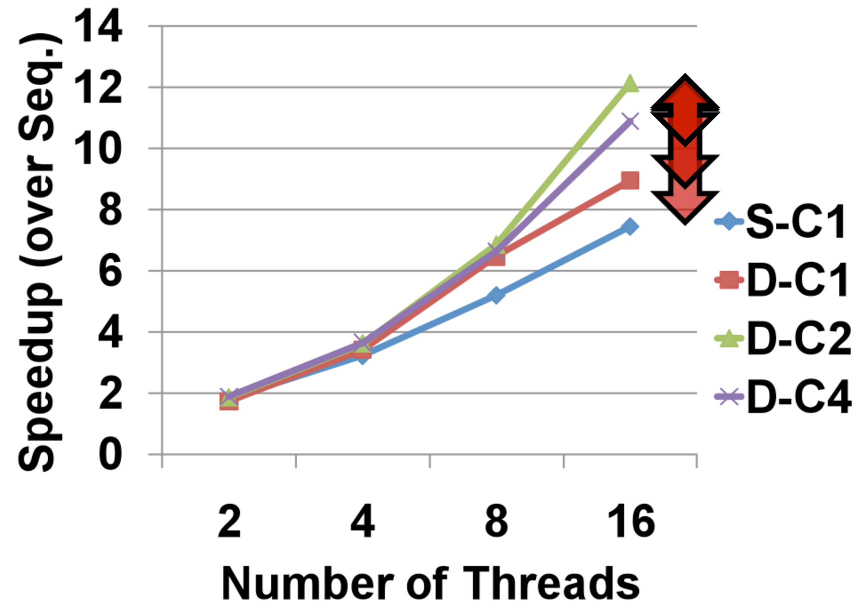


Programmability (Manual vs. OpenTM)

Data Structure	Manual	OpenTM
heap	29	4
list	16	11
queue	23	3
rbtree	177	19

- Counted the number of annotated C code lines
  - To implement the transactional data structures used in STAMP
- Manual programming
  - Need to manually instrument functions with TM barriers
    - Highly error-prone
- OpenTM programming
  - No need for manual instrumentation
    - Simply mark TM-callable functions ➔ Compiler generates the code

Using the transfor Construct



- Ran a simple histogram benchmark
  - Static (C=1) / dynamic (C=1,2,4)
- Dynamic scheduling with C=2 performs best
  - Static: Work imbalance
  - Dynamic with C=1: Scheduling overhead (contention on global work queue)
  - Dynamic with C=4: More conflicts among (larger) transactions
- ***Benefit: Simply change the scheduling policies and params of transfor***

Outline

- ❑ Background & Motivation
- ❑ Challenge #1: Nested Parallelism
- ❑ Challenge #2: Programmability
  - OpenTM improves the programmability of TM
    - Abstract low-level TM programming issues (e.g., barrier inst.)
    - Allow high-level management of TM programming options
- ❑ Challenge #3: Verification
- ❑ Conclusions

Challenge #3: Verification

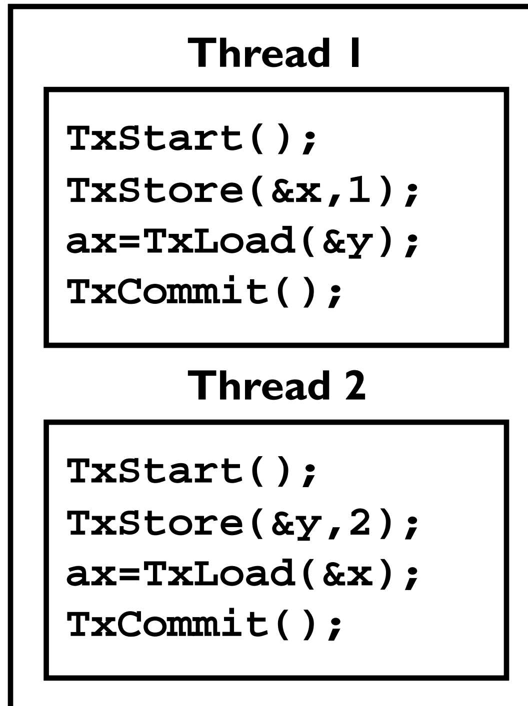
- What we need: a TM verification environment
  - To check correctness of TM implementations
- My proposal: ChkTM [ICECCS 10]
  - A flexible model checking environment for TM
  - ***Goal: Bridge the gap between the abstract model and actual implementation***
    - Model TM barriers close to the implementation level
    - Accurately model the timestamp-based version control
  - Using ChkTM
    - Found a correctness bug in a widely-used STM implementation
    - Checked the serializability of several TMs

ChkTM Overview

- Test program generator
  - Generate all the possible test programs
    - E.g., two threads, one txn per thread, at most two ops per txn
  - The fidelity of verification with a small configuration
    - Flat TM: High fidelity with a reduction theorem [Guerraoui 08]
    - Nested TM: No reduction theorem → Open research question
- TM model specifications
  - Model the behavior of each TM barrier
- Architectural state space explorer
  - Architectural simulator: model a simple shared-memory system
  - State space explorer: explore every possible execution of a prog

ChkTM Execution Flow

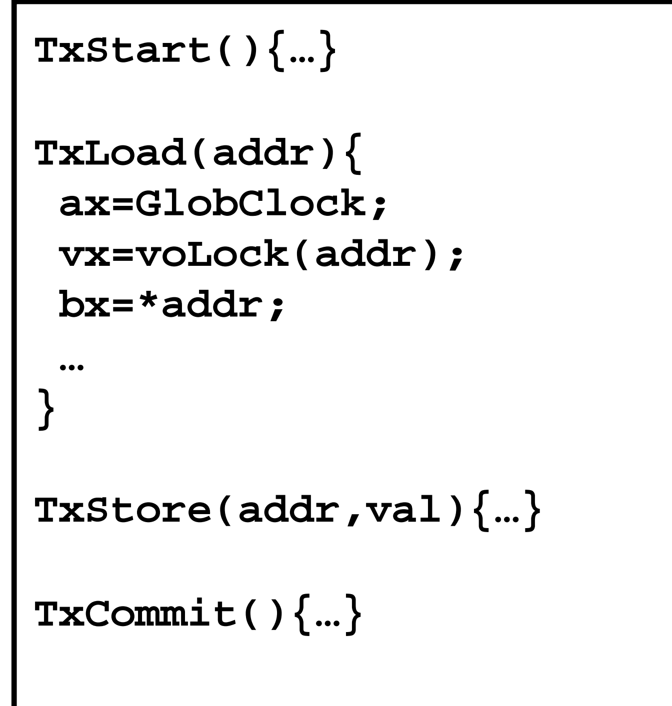
Test Program Generator



Call

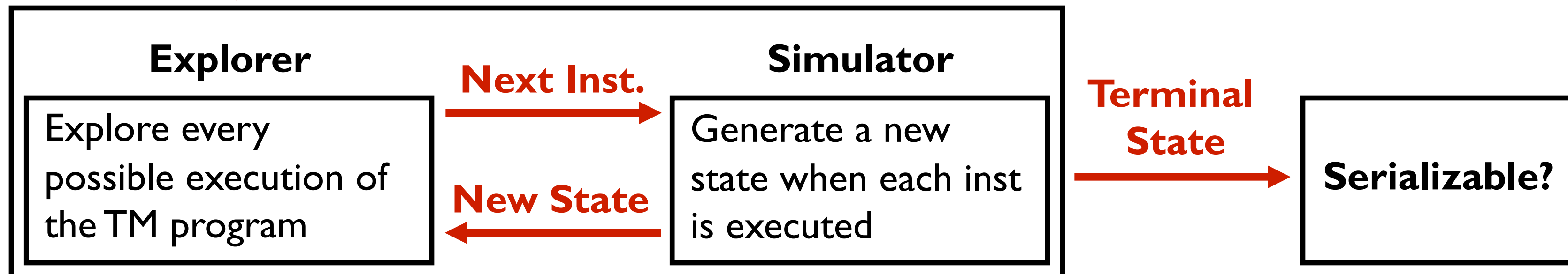
Return

TM Model Specifications



Input

Architectural State-Space Explorer



Checking Serializability

- First step: Perform coarse-grain exploration (at transaction level)
  - Generate all serial schedules
  - The valid terminal corresponding to each serial schedule
    - VORs (values observed by reads)
    - Final memory state
- Second step: Perform fine-grain exploration (at instruction level)
  - Each terminal state is checked with valid terminals
  - If a matching valid terminal exists, the execution is valid
    - Intuition: VORs are same with a serial schedule ➔ View serializable
  - If no matching valid terminal, ChkTM reports serializability violation
    - With a counterexample (invalid execution path)

Modeling Timestamps

- Goal: Accurately model timestamps
  - Used in high-performance STMs (e.g., TL2, McRT-STM)
    - Without an accurate model, the fidelity of verification remains low
- Challenge: State space explosion
  - Unbounded # of states with different timestamp values
- Our solution: Timestamp canonicalization
  - Key idea: the relative ordering among timestamps is important
    - But not the exact values
  - Canonicalize all the timestamp values after each inst is executed
    - Step 1: Compute the set of all the timestamp values
    - Step 2: Numerically sort them
    - Step 3: Replace each value with its ordinal position in the sorted set

Using ChkTM

- Case study: Found a correctness bug in eager TL2
  - The bug: The abort barrier incorrectly handles timestamps
  - Random tests: Hard to expose this bug
  - ChkTM: Explore every possible execution path ➔ Always found
    - Additional benefit: Counterexamples ➔ Help finding the bug

- Serializability results
  - Checked TL2 and SigTM models with a small configuration
    - 2 threads, 1 txn per thread, at most 3 operations per txn
    - Each run: < 1s / Entire test of each model: ~4 hours
  - No serializability violation has been reported

Conclusions

- Challenge #1: Nested Parallelism
  - NesTM [SPAA 10]
    - Extend a non-nested STM to support nested parallel transactions
  - FaNTM [ICS 10]
    - Practical hardware support to accelerate software nested transactions
- Challenge #2: Programmability
  - OpenTM [PACT 07]
    - Integrate TM into a high-level programming model (OpenMP + TM)
- Challenge #3: Verification
  - ChkTM [ICECCS 10]
    - A flexible model checking environment for TM systems