Implementing and Evaluating Nested Parallel Transactions in STM

Woongki Baek, Nathan Bronson,
Christos Kozyrakis, Kunle Olukotun
Stanford University

Introduction

```
// 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,…);
 }}
```

- Transactional Memory (TM) simplifies parallel programming
  - Atomic and isolated execution of transactions

- Current practice: Most TMs do not support nested parallelism

- Nested parallelism in TM is becoming more important
  - To fully utilize the increasing number of cores
  - To integrate well with programming models (e.g., OpenMP)

Previous Work: NP in STM

- ☐ [ECOOP 09] NePaLTM with practical support for nested parallelism
  - Serialize nested transactions

- ☐ [PPoPP 08] CWSTM that supports nested parallel transactions
  - With the lowest upper bound of time complexity of TM barriers
  - No (actual) implementation / (quantitative) evaluation

- ☐ [PPoPP 10] a practical, concrete implementation of CWSTM
  - With depth-independent time complexity of TM barriers
  - Use rather complicated data structures such as concurrent stack

- ☐ Remaining question: Extend a timestamp-based, eager-versioning STM
  - To support nested parallel transactions

Contributions

- Propose NesTM with support for nested parallel transactions
  - Extend a timestamp-based, eager-versioning STM
- Discuss complications of concurrent nesting
  - Describe subtle correctness issues
  - Motivate further research on proving / verifying nested STMs
- Quantify NesTM across different use scenarios
  - Admittedly, substantial runtime overheads to nested transactions
    - E.g., Repeated read-set validation
  - Motivate further research on performance optimizations

Outline

- ☐ Introduction
- ☐ Background
- ☐ NesTM Algorithm
- ☐ Complications of Nesting
- ☐ Evaluation
- ☐ Conclusions

Background: Semantics of Nesting

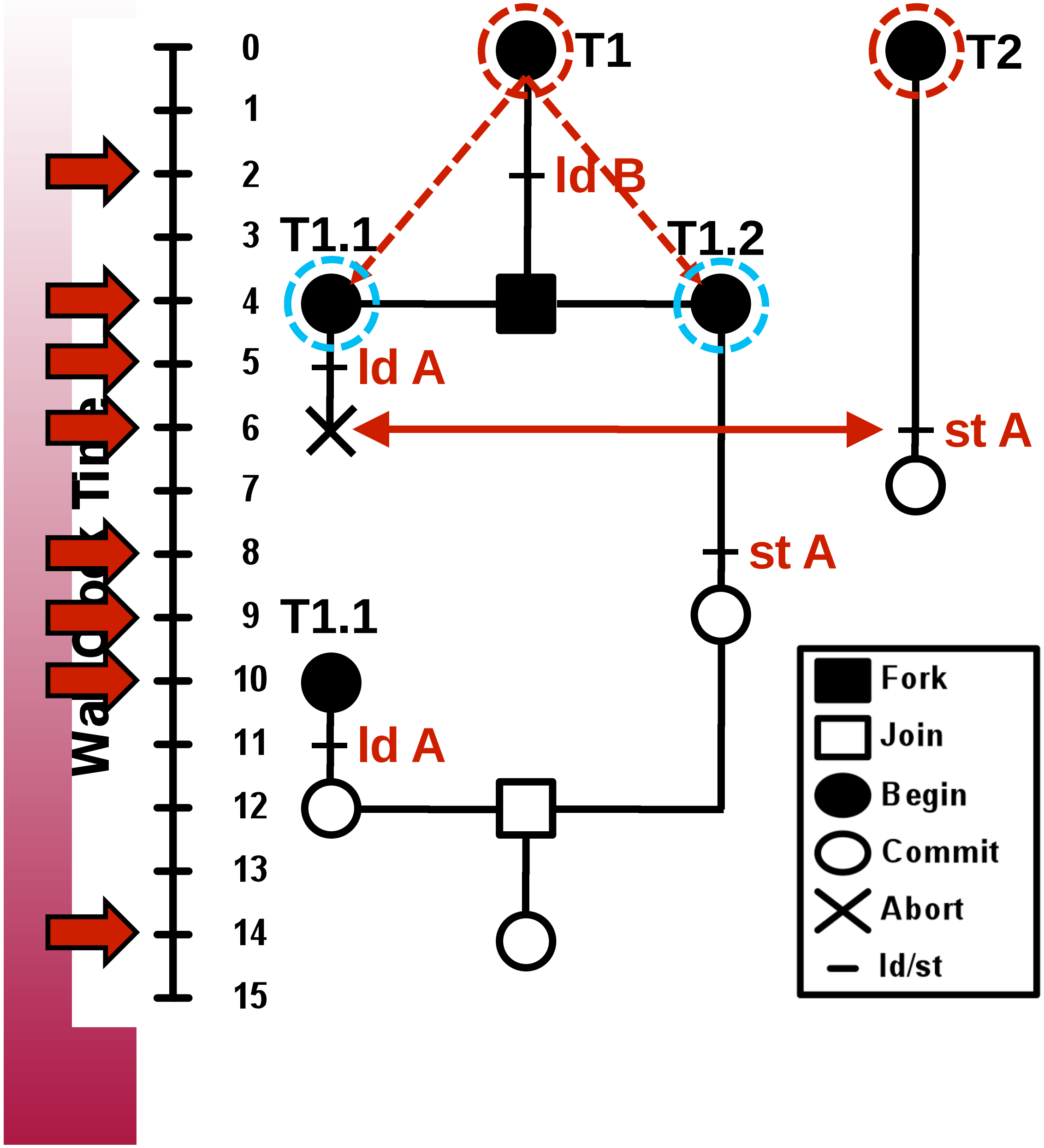
❑ Definitions

- Transactional hierarchy has a tree structure
  - Ancestors(T) = Parent(T) $\cup$ Ancestors(Parent(T))
- 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)

Background: Example of 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 □ Readers(A)
  - T1.1 □ Ances(T2)
- ☐ T=8: No conflict
  - T1.2 writes to A
  - Readers(A)=Writers(A)= □
- ☐ Serialization order
  - **T2 → T1**

NesTM Overview

- ❑ Extend an eager data-versioning STM
  - In-place update ➔ No need to look up parent's write buffer
  - Useful property: Once acquire ownership, keep it until commit / abort

- ❑ Global data structures
  - A global version clock (GC)
  - A set of version-owner locks (voLocks):
    - T LSBs: Owner's TID / Remaining bits: Version Number

- ❑ Transaction descriptor
  - Read-version (RV): GC value sampled when the txn starts
  - R/W sets: Implemented using a doubly linked list
  - Pointer to parent's transaction descriptor
  - Commit-lock: to synchronize concurrent commits of children

TxLoad

```
TxLoad(Self,addr){
 vl=getVoLock(addr);
 owner=getOwner(vl);
 if(owner==Self){ // Read data }
 } else if(isAnces(Self,owner)){
  cv=getTS(vl);
  if(cv>Self.rv){ // Abort }
  else{ // Read data }
 } else{ // Abort }}
```

- If the owner (of the memory object) is the transaction itself
  - Read the memory value
- Else if the owner is an ancestor of the transaction
  - If the version number is newer than the transaction's RV → Abort
  - Else → Read the memory value
- Else → Abort

TxStore

```
TxStore(Self,addr,val){
 owner=getOwner(addr);
 if(owner==Self){ // Write data }
 else if(isAnces(Self,owner)){
  if(atomicAcqOwnership(Self,owner,addr)==success){
   if(validateReaders(Self,owner,addr)==success){
    // Write data }
   else{ // Abort }
  } else { // Abort }}
 else { // Abort }}
```

- If the owner is the transaction itself ➔ Write
- Else if the owner is an ancestor of the transaction
  - If the atomic acquisition of the ownership is successful
    - If the validation of all the readers in the hierarchy is successful ➔ Write
    - Else ➔ Abort
  - Else ➔ Abort
- Else ➔ Abort

TxCommit

```
TxCommit(Self){
 wv=IncrementGC();
 for each e in Self.RS {
  // Perform the same check in TxLoad
  // If fails, the transaction aborts }
 mergeRWSetsToParent(Self);
 for each e in Self.WS {
  // Increment version number using “wv” and
  // transfer ownership to parent }
 …}
```

- Validate every memory object in RS
    - Using the same conditions checked in TxLoad ➔ If fails, abort
- Merge R/W sets to the parent ➔ Linking the pointers
    - Loss of temporal locality on these entries
- Validation / Merging is protected by parent’s commit-lock
    - To address the issue with non-atomic commit (See the paper)
- Increment version number / transfer ownership for the objects in WS

TxAbort

```
TxAbort(Self){
 for each e in Self.WS {
  // Restore the memory value to the previous value
 }
 for each e in Self.WS {
  // Restore the voLock value to the previous value
 }
 // Retry the transaction
}
```

- For every memory object in WS
  - Restore the memory value to the previous value
- For every memory object in WS
  - Restore the voLock value to the previous value
    - Refer to the paper for the “invalid read” problem
- Retry the transaction

Outline

- ☐ Introduction
- ☐ Background
- ☐ NesTM Algorithm
- ☐ Complications of Nesting
- ☐ Evaluation
- ☐ Conclusions

Complications of Nesting

- ☐ Subtle correctness issues discovered while developing NesTM
  - Invalid read, non-atomic commit, zombie transactions
- ☐ Current status: No hand proof of correctness/liveness of NesTM
- ☐ Model checking: ChkTM [ICECCS 10]
  - Checked correctness with a very small configuration
    - Thread configuration: [1, 2, 1.1, 1.2] / Two memory op's per txn
  - Failed to check with larger configurations due to large state space
    - Motivate reduction theorem / partial order reduction techniques
- ☐ Random tests: Using the implemented NesTM code
  - Tested with larger configurations (e.g., nesting depth of 3)

Evaluating NesTM

- Q1: Runtime overhead for top-level parallelism
  - Used STAMP applications (Baseline STM vs. NesTM)
  - Maximum performance difference is ~25%
    - Due to the extra code in NesTM barriers

- Q2: Performance of nested transactions
  - More in the following slides

- Q3: Using nested parallelism to improve performance
  - Used a u-benchmark based on two-level hash tables
  - If single-level parallelism is limited (e.g., frequent conflicts)
    - Exploiting nested parallelism can be beneficial

Q2: Performance of Nested Txns

Flat version

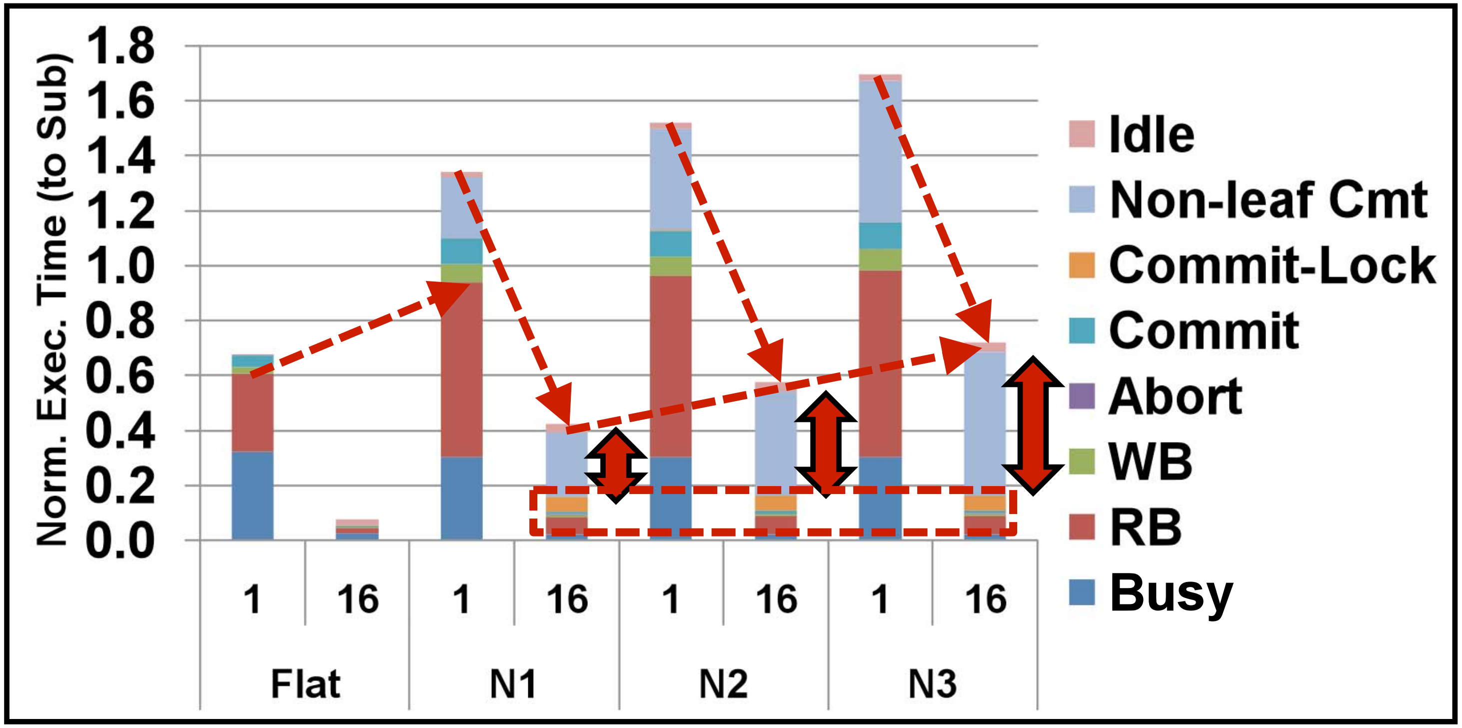
```
// Parallelize this loop
for(i=0;i<numOps;i+=C){
 atomic{
  for(j=0;j<C;j++){
   accessHT(i,j,…);}
 }
}
```

Nested version (N1)

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

- hashtable: perform operations on a concurrent hash table
  - Two types of operations: Look-up (reads) / Insert (reads/writes)
- Subsumed: Sequentially perform all the operations in a single txn
  - Emulate an STM that flattens and serializes nested transactions
- Flat: Concurrently perform operations using top-level txns
- Nested: Repeatedly add outer-level transactions
  - N1, N2, and N3 versions

Q2: Performance of Nested Txns



- ☐ Scale up to 16 threads (N1 with 16 threads ➔ 3x faster)
- ☐ Performance issues
  - Non-parallelizable, linearly-increasing overheads
    - E.g., Repeated read-set validation
  - More expensive read/write barriers (loss of temporal locality)
  - Contention on commit-lock (Many nested txns simultaneously commit)

Conclusion

- ☐ Propose NesTM with support for nested parallel transactions
  - Extend a timestamp-based, eager-versioning STM
- ☐ Discuss complications of concurrent nesting
  - Describe subtle correctness issues
  - Motivate further research on proving / verifying nested STMs
- ☐ Quantify NesTM across different use scenarios
  - Admittedly, substantial runtime overheads to nested transactions
    - E.g., Repeated read-set validation
  - Motivate further research on performance optimizations
    - Software: more efficient algorithm / implementation
    - Hardware: cost-effective hardware acceleration [ICS 10]