Making Nested Parallel Transactions Practical using Lightweight Hardware Support

Woongki Baek, Nathan Bronson, Christos Kozyrakis, Kunle Olukotun

Stanford University
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 TM

- Software-only approach: [PPoPP 10], [SPAA 10]
  - Use complex data structures or depth-dependent algorithm for NP
  - Degrade the performance of transactions
    - Excessive overheads even for single-level txns
  - Impractical unless performance issues are addressed

- Full HTM approach: [Vachharajani 08]
  - Intrusive modifications in caches ⇒ Complicate HW design
    - For nesting-aware conflict detection & data versioning
  - Unlikely to be adopted unless HW complexity is lowered

- Needed: TM with practical support for nested parallelism
Contributions

- Propose Filter-accelerated Nested TM (FaNTM)
  - **Goal:** Make nested parallel transactions practical
  - Performance: Eliminate excessive overheads of SW nested txns
    - By offloading nesting-aware conflict detection to HW filters
  - Implementation cost: Simplify hardware design
    - By fully decoupling nested transactions from caches

- Quantify FaNTM across different use scenarios
  - Small runtime overheads for top-level parallelism
  - Nested txns scale well, significantly faster than SW ones
  - Tradeoff between top-level and nested parallelism
Outline

- Introduction
- Background
- Design of FaNTM
- Evaluation
- Conclusion
Definitions

• \( \text{Family}(T) = \text{ancestors}(T) \uplus \text{descendants}(T) \)
  - Transactional hierarchy has a tree structure

• \( \text{Readers}(o) \): a set of active transactions that read “o”

• \( \text{Writers}(o) \): a set of active transactions that wrote to “o”

Conflicts

• \( T \) reads from “o”: R/W conflict
  - If there exists \( T' \) such that \( T' \cap \text{writers}(o) \), \( T' \neq T \), and \( T' \uplus \text{ancestors}(T) \)

• \( T \) writes to “o”: R/W or W/W conflict
  - If there exists \( T' \) such that \( T' \cap \text{readers}(o) \cap \text{writers}(o) \), \( T' \neq T \), and \( T' \uplus \text{ancestors}(T) \)
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 Family(T2)

- T=8: No conflict
  - T1.2 writes to B
  - T1 Readers(B)
  - T1 Family(T1.2)

- Serialization order
  - T2 → T1
FaNTM Overview

- FaNTM is a hybrid TM that extends SigTM [ISCA 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
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)

- Ld Req. from $T_R$
  - WSig Hit?
    - Y
      - R/W Conflict
        - Family?
          - Y
            - Propagate
          - N
            - Propagate
        - N
          - Nack Req.
    - N
      - Propagate

- Done
TMB: Conflict Detection (Ldx)

```
Ldx Req. from TR

WSig Hit?  

N  

Y  

Nack Req.  

Family?  

Y  

Propagate  

N  

Done

R/W Conflict

WSig Hit?  

N  

Y  

Rsig Hit?  

Family?  

Y  

Propagate  

N  

Abort

W/W Conflict
```
Software: Transaction Descriptor

```c
struct transaction {
    int Tid;
    Log UndoLog;
    struct transaction* Parent;
    lock CommitLock;

    ...
}
```

- **Tid**: Transaction ID
- **UndoLog**: Hold previous memory values (eager versioning)
  - Implemented using doubly-linked lists
  - Entry: <addr, previous memory value, ptrs to neighbors, …>
- **Parent**: Pointer to the parent’s descriptor
- **CommitLock**: Synchronize concurrent commits by children
Software: Read Barrier

```c
TxLoad(addr) {
  RSigInsert(addr);
  val=*addr;
  return val;
}
```

- !Insert the address of the memory object in RSig
  - ! No need to maintain a software read set

- !Attempt to read the memory value
  - ! If the load request is successful (i.e., not nacked)
    - ! The memory value is returned
  - ! Otherwise, the TMB interrupts the processor
    - ! To abort the transaction (R/W conflict)
Software: Write Barrier

```
TxStore(addr,val) {
    WSigInsert(addr);
    fetchEx(addr);
    undoLog.insert(addr,*addr);
    *addr=val;
}
```

- Insert the address of the memory object in WSig
- Broadcast an exclusive load request over the network
  - If this request is successful (i.e., not nacked)
    - The current memory value is inserted in the undo log
    - Memory object is updated in-place (eager versioning)
  - Otherwise, the TMB interrupts the processor
    - To abort the transaction (W/W/W conflict)
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/WSigs to its parent (sending messages over the network)
  - Merge its undo-log entries to its parent
  - Finish by resetting TM metadata
Outline

- Introduction
- Background
- Design of FaNTM
- Evaluation
- Conclusion
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

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

Nested version (N1)

```c
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 major performance degradation with deeper nesting
  - Conflict detection in HW ➔ No repeated validation across nesting
- Significantly faster (e.g., 12x) than a nested STM (NesTM) [SPAA 10]
  - Making nested parallel transactions practical
Q3: Exploiting Nested Parallelism

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

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,...);
      }
    }
  }
}
```
Q3: Flat vs. Nested

- 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 in both levels

- Motivate research on nesting-aware runtime systems
  - Dynamically exploit the parallelism in multiple levels
Conclusion

- Propose Filter-accelerated Nested TM (FaNTM)
  - **Goal:** Make nested parallel transactions practical
    - Performance: Eliminate excessive overheads of SW nested txns
      - By offloading nesting-aware conflict detection to HW filters
    - Implementation cost: Simplify hardware design
      - By fully decoupling nested transactions from caches

- Quantify FaNTM across different use scenarios
  - Small runtime overheads for top-level parallelism
  - Nested txns scale well, significantly faster than SW ones
  - Tradeoff between top-level and nested parallelism

- More details (e.g., complications of nesting) in the paper