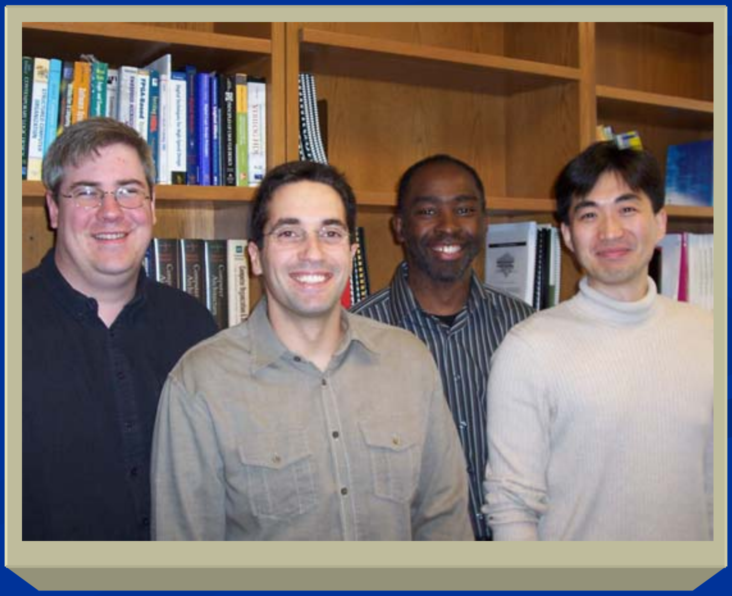
The Software Stack for Transactional Memory

Challenges and Opportunities

Brian D. Carlstrom
JaeWoong Chung,
Christos Kozyrakis, Kunle Olukotun

Computer Systems Lab
Stanford University
http://tcc.stanford.edu



Parallel Programming on Shared Memory

- Traditionally done using locks
- But locks are hard to use
  - Semantic problems
    - Deadlock
    - Priority inversion
  - Performance problems
    - Simplicity at the expense of concurrency
    - High concurrency at the expense of simplicity
    - **Pessimistic concurrency**

Transactional Memory

- Allows for lock-free parallel programming
- Transactions mark critical sections
- Same properties as database transactions
  - Atomicity : all or nothing
  - Isolation : no partial updates
- Transactions are easier to use than locks
  - Coarse-grained non-blocking synchronization
  - **Optimistic concurrency**

Opportunities and Challenges

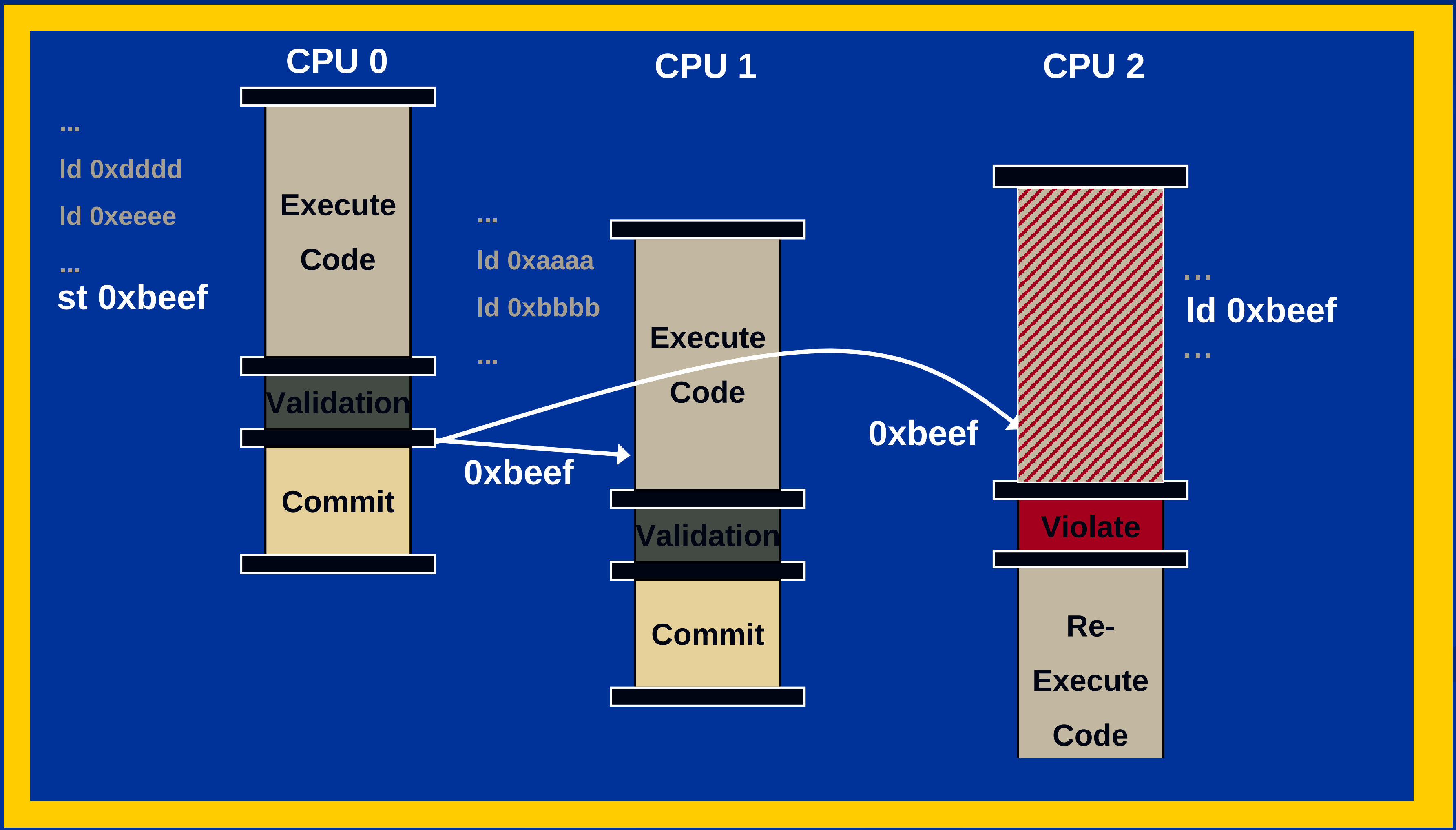
- TM is a promising solution for easy and efficient parallel programming on multi-core systems
- TM brings up both opportunities and challenges to software stack
- Today’s talk focuses on, but not limited to, the software stack on top of hardware TM

Contents

- Transactional Memory Overview
  - What is TM?
  - Why is it interesting to Multi-core systems?
  - TM example and primitives
- Software Stack
  - Data Structure
  - Programming Composition
  - Operating System
  - Language Implementation
  - Programming Models
  - Distributed Transactions
- Conclusion

TM execution model example

(Transactional Coherence and Consistency)



Advanced TM primitives

- Nesting [isca06]

Core 0 | Core 1

```
// A is initially 0;
Atomic {
    ....
    Atomic {
        A++;  // 1          A = ;
        ....
    }
                            = A; // 0
    A++; // 2
}
                            = A; // 2
```

< Closed Nesting >

Core 0 | Core 1

```
// A is initially 0;
Atomic {
    ....
    Open_Atomic {
        A++;  // 1          A = ;
        ....
    }
                            = A; // 1
    A++; // 2
}
                            = A; // 2
```

< Open Nesting >

- Callback
  - Violation Handler and Commit Handler

Data Structure
(Opportunity)

- Coarse-grain non-blocking synchronization
  - Both ease-to-use and performance
- Nesting – reduces violation overhead
  - Open nesting reduces the frequency of conflicts
  - Closed nesting reduces the penalty of violation
- Callback – provides more programmability
  - Violation Handler
    - Automatic clean-up at conflicts
    - Application specific conflict handler

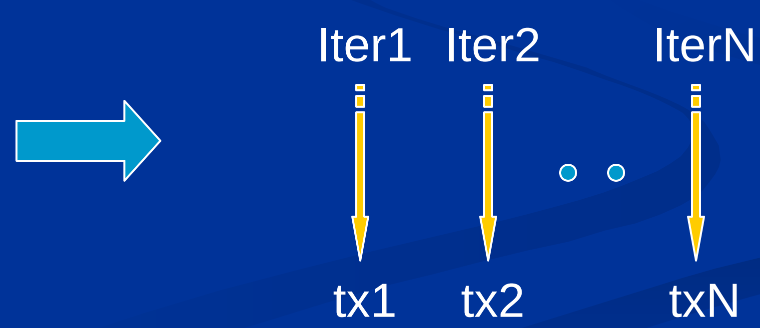
Data Structure
(Challenge)

- How to hide the advanced techniques for novice programmers?
  - TM-based library
  - like GNU classpath Java library

Programming Composition
(Opportunity)

- Transaction nesting
  - Flexibility in composing transactions
- Speculative parallel loops
  - To avoid the hassle of setting and wrapping up multiple threads

```
T_FOR(..)
{
   // loop iteration
}
```



Programming Composition
(Challenge)

- Transactional I/O
  - I/O buffering
    - Defer I/O operations by the commit
    - Execute the deferred operations at commit
- Conditional Waits
  - Wait() is related to lock objects
  - Composible conditions for atomic regions
- Overflows
  - Deep call stacks make transactions long
  - Buffer overflow mechanism

Operating System

(Opportunity)

- Non-blocking synchronization
  - Easier kernel construction
  - Potential for speedup
- Atomicity for fault-tolerance
  - TM undoes instructions at rollback
  - Easy check-pointing
- Isolation for security [sosp03]
  - TM isolates instructions by commit

Operating System

(Challenge)

- Context-switch
  - In hardware TMs, transactional states have affinity to processors
- Interrupt [hpca06]
  - Swapping in/out transactions
- I/O

- Software TM runs on virtual address space

Programming Models
(Opportunity)

- TM-based models tuned for parallelism
  - Atomos [pldi06]
    - Java – old synchronization APIs + new TM primitives
    - support for nesting, callback, and high-level language construct

  - X10, Fortress, and Chapel also explore transactions

Programming Models
(Challenge)

- Many different semantics for TM
  - Different definitions for the same term
  - Strong vs. weak consistency
    - We prefer strong consistency
      - No need to worry about possible bugs due to interaction between transaction code and non-transactional code
  - APIs for application and system programming

Language Implementation
(Opportunity)

- Aggressive JIT compiler optimization
  - Try unsafe optimization
    - Constant Propagation
  - Rollback the computation if there is a problem
  - Restart with safe code

- Speculative Parallelism
  - Make a code segment run in parallel

Language Implementation
(Challenge)

- Memory allocation
  - Private memory pool or Nesting
- Incremental/Concurrent garbage collection
  - Use violation handlers to deal with conflicts between collectors and mutators

Distributed Transactions
(Opportunity)

- Integration with distributed transaction systems
  - Transaction Service in .Net, J2EE, and CORBA
- Extracting parallelism from distributed objects with transactional properties
  - Enterprise Java Beans
    - TX_REQUIRED, TX_BEAN_MANAGED

Distributed Transactions
(Challenge)

- E-commerce transactions are long
  - Longer than time quanta
  - I/O operations
- TM virtualization can be helpful

Conclusion

- Transactional Memory is a promising solution for parallel programming
- Transactional memory brings up both opportunities and challenges to software stack

- We hope research forces from many areas join the efforts for Transactional memory