Transactional Memory

Konrad Lai
Microprocessor Technology Labs, Intel

Intel Multicore University Research Conference
Dec 8, 2005
Motivation

- Multiple cores face a serious programmability problem
  - Writing correct parallel programs is very difficult
- Transactional Memory addresses key part of the problem
  - Makes parallel programming easier by simplifying coordination
  - Requires hardware support for performance
What is Transactional Memory?

Transactional Memory (TM) allows arbitrary multiple memory locations to be updated atomically.

**Basic mechanisms:**

- **Isolation:** Track read and writes, detect when conflicts occur.
- **Version management:** Record new/old values.
- **Atomicity:** Commit new values or abort back to old values.

**Thread 1**

```
begin_xaction
A = A - 20
B = B + 20
A = A - B
C = C + 20
end_xaction
```

Thread 1’s accesses and updates to A, B, C are atomic.

**Thread 2**

```
begin_xaction
C = C - 30
A = A + 30
end_xaction
```

Thread 2 sees either “all” or “none” of Thread 1’s updates.
Problem: Lock-Based Synchronization

Lock-based synchronization of shared data access is fundamentally problematic.

- Software engineering problems
  - Lock-based programs do not compose
  - Performance and correctness tightly coupled
  - Timing dependent errors are difficult to find and debug

- Performance problems
  - High performance requires finer grain locking
  - More and more locks add more overhead

Need a better concurrency model for multi-core software

Copyright © 2005, Intel Corporation
Transactional Memory benefits

• Focus: Multithreaded programmability crisis
  ▫ Programmability & performance
    ➔ Allows conservative synchronization
    ➔ Programmer focuses on parallelism & correctness, HW extracts performance
  ▫ Software engineering and composability
    ➔ Allows library re-use and composition (locks make this very difficult)
    ▪ Critical for wide multi-core demand

• Makes high performance MT programming easier
  ▫ Captures a fundamental, well-known, intuitive “atomic” construct
  ▫ Been around for decades
  ▫ Similar to a “critical section” but without its problems
    ➔ No deadlocks, priority inversion, data races, unnecessary serialization
Software Transactional Memory

- Software Transactional Memory (1995 until now)
  - Significant work from Sun, Brown, Cambridge, Microsoft
  - Serious performance limitations
    - Degrades “common” case of no conflicts/contention
    - >90% of transactions are no conflicts
      - 90% of critical sections are uncontended: what if all these slowed down by 5X?
  - Serious deployability limitations
    - Relies on special runtime support
    - Invasive to applications and libraries
  - Is STM is too slow and too invasive to deploy?
    - Could there be better implementation?
    - But great to understand complex usage models of the future
Hardware Support for Performance

1. Record recovery state
2. Buffer updates/track accesses
3. Commit if no external access (discard all updates if conflict)

Core 1
begin_xaction
A.withdraw(20)
B.deposit(20)
end_xaction

A.sum = 100
B.sum = 200

Core 2
begin_xaction
Sum = A.sum + B.sum
end_xaction

A.sum = 100
B.sum = 200

Core 2 sees $300 – never $280 or $320
What if HW is not sufficient?

- **This is the deployability challenge**
  - The missing piece of the puzzle for all prior work...

- **Resource limitations are fundamental**
  - **Space**: caches
    - More HW delays the inevitable: will always be an n+1 case
  - **Time**: scheduling quanta
    - Programmers have no control over time
  - Affects functionality, not just performance
    - Some transactions may never complete

- **Making HW limit explicit is difficult**
  - Limited usage only
  - Unreasonable for high level languages
  - How do you architect it in an evolvable manner?
Virtualize for Completeness

Core 1

- Timer interrupts,
  - Context switches,
  - Exceptions,…

Virtual TM

- Limited buffers

virtual address space

Log-buffer space

- Out-of-band concurrency control

- Overflow management
  - Using virtual memory
  - Software libs. and microcode

- Programmer transparent
  - Performance isolation
  - Suspendable/swappable
Recent TM Research

- Recently, focus on solving the harder problem of TM
  - Making the model immune to cache buffer size limitations, scheduling limitations, etc.
- TCC (Stanford) (2004)
  - Same limitation as Herlihy/Moss for TM (size limited to local caches)
- LTM (MIT), VTM (Intel), LogTM (Wisconsin) (2005)
  - Assume hardware TM support
  - Add support to allow transactions to be immune to resource limitations
  - Goals of each similar, approaches very different
    - LTM: only resource overflow
    - VTM: complete virtualization
    - LogTM: only resource overflow
Some Research Challenges

- Large transactions
- Language extensions
- IO, loophole, escape hatches, …
- Interaction and co-existence with
  - Other synchronization schemes: locks, flags, …
  - Other transactions
    - Database transaction
    - System transaction (Microsoft)
  - Other libraries, system software, operating system, …
- Performance monitor, tuning, debugging, …
- Open vs Closed Nesting
- Interaction between transaction & non-transaction
- Usage & Workload
  - PLDI workshop
Backup
**TM – First Decade**

- **IBM 801 Database Storage (1980s)**
  - Lock attribute bits on *virtual memory* (via TLBs, PTEs) at 128 byte granularity

- **Load-Linked/Store Conditional (Jensen et al. 1987)**
  - Optimistic update of single cache line (Alpha, MIPS, PowerPC)

- **Transactional Memory (Herlihy&Moss 1993)**
  - Coined term; TM generalization of LL/SC
  - Instructions explicitly identify transactional loads and stores
  - Used dedicated transaction cache
    - Size of transactions limited to transaction cache

- **Oklahoma Update (Stone et al./IBM 1993)**
  - Similar to TM, concurrently proposed
  - Didn’t use cache but **dedicated monitored registers** to operate upon