Transactional Memory

Architectural Support for Practical Parallel Programming

The TCC Research Group

Computer Systems Lab
Stanford University
http://tcc.stanford.edu
The Era of Multi-core Chips

- Diminishing returns from single-core chips
  - Wire delays, memory latency, power consumption, complexity, ...

- Multi-core chips are the scalable alternative
  - Modular, fault-tolerant, memory-level parallelism ...
  - All processor vendors are building CMPs

- But, we how do we program them?
  - Correct & fast parallel programming is a black art

Urgent: make parallel programming the common case
What Makes Parallel Programming Hard?

1. Finding independent tasks
2. Mapping tasks to threads
3. Defining & implementing synchronization protocol
4. Race conditions & deadlock avoidance
5. Memory model
6. Portable & predictable performance
7. Scalability
8. Locality management
9. Composing parallel tasks
10. Recovering from errors

11. And, of course, all the single thread issues...
Example: Java 1.4 HashMap

- Fundamental data structure
  - Map: Key → Value

```java
public Object get(Object key) {
    int idx = hash(key); // Compute hash
    HashEntry e = buckets[idx]; // to find bucket
    while (e != null) {
        bucket
        if (equals(key, e.key))
            return e.value;
        e = e.next;
    }
    return null;
}
```

- Not thread safe (no lock overhead when not needed)
Synchronized HashMap

- Java 1.4 solution: synchronized layer
  - Convert any map to thread-safe variant
  - Explicit locking – user specifies blocking

```java
public Object get(Object key) {
    synchronized (mutex) { // mutex guards all accesses to map m
        return m.get(key);
    }
}
```

- Coarse-grain synchronized HashMap:
  - Thread-safe, easy to program
  - Limits concurrency → poor scalability
    - E.g., 2 threads can’t access disjoint hashtable elements
ConcurrentHashMap

- Java 5 solution: Complete redesign

```java
public Object get(Object key) {
    int hash = hash(key);
    // Try first without locking...
    Entry[] tab = table;
    int index = hash & (tab.length - 1);
    Entry first = tab[index];
    Entry e;

    for (e = first; e != null; e = e.next) {
        if (e.hash == hash && eq(key, e.key)) {
            Object value = e.value;
            if (value != null)
                return value;
            else
                break;
        }
    }
    ...

    // Recheck under synch if key not there or interference
    Segment seg = segments[hash & SEGMENT_MASK];
    synchronized(seg) {
        tab = table;
        index = hash & (tab.length - 1);
        Entry newFirst = tab[index];
        if (e != null || first != newFirst) {
            for (e = newFirst; e != null; e = e.next) {
                if (e.hash == hash && eq(key, e.key))
                    return e.value;
            }
            return null;
        }
    }
```

Fine-grain locking & concurrent reads: complicated & error prone
Quantitative Example

**Hash-Table**

- Execution Time
- Processors
- Coarse Locks
- Fine Locks

**Balanced Tree**

- Execution Time
- Processors
- Coarse Locks
- Fine Locks
Locks are Simply Broken

- Performance – correctness tradeoff
  - Coarse-grain locks: serialization
  - Fine-grain locks: deadlocks, livelocks, races, ...

- Cannot easily compose lock-based code

- No failure atomicity

- User’s specification ≠ implementation
  - Makes programming & tuning difficult
Outline

- Motivation

- Transactional Memory
  - Motivation, use & performance example

- Transactional Coherence & Consistency (TCC)
  - Architecture model, implementation, advanced features

- Performance evaluation
  - SpecJBB2000

- Conclusions & current work
Transactional Memory (TM)

- Programmer specifies large, atomic tasks [Herlihy’93]
  - atomic { some_work; }
  - Multiple objects, unstructured control-flow, ...
  - Declarative approach; system implements details

- Transactional memory provides
  - Atomicity: all or nothing
  - Isolation: writes not visible until transaction commits
  - Consistency: serializable commit order

- Performance through optimistic concurrency [Kung’81]
  - Execute in parallel assuming independent transactions
  - If conflicts detected, abort & re-execute one transaction
    - Conflict = two transactions read-write same data
Transactional HashMap

- Transactional layer via an ‘atomic’ construct
  - Ensure all operations are atomic
  - Implicit atomic directive – system finds concurrency

```java
public Object get(Object key) {
    atomic // System guarantees atomicity
    {
        return m.get(key);
    }
}
```

- Transactional HashMap
  - Thread-safe, easy to program, good performance
Transactional Memory: Performance

- Concurrent read operations
  - Basic locks do not permit multiple readers
    - Need reader-writer locks \(\Rightarrow\) more complex
  - Automatically allows multiple concurrent readers

- Concurrent access to disjoint data
  - Users have to manually perform fine-grain locking
    - Difficult and error prone
    - Not modular
  - Automatically provides fine-grain locking
Performance Revisited

HashMap

Execution Time vs Processors

Balanced Tree

Execution Time vs Processors
Transactional Memory Benefits

- As easy to use as coarse-grain locks
- Scale as well as fine-grain locks
  - No performance correctness tradeoff
  - Automatic read-read & fine-grain concurrency
- Composition:
  - Safe & scalable composition of software modules
- Failure atomicity & recovery
Does TM help with all the Parallel Programming Issues?

- Finding independent tasks
- Mapping tasks to threads
- Defining & implementing synchronization protocol
- Race conditions & deadlock avoidance
- Memory model
- Composing parallel tasks
- Portable & predictable performance
- Scalability
- Locality management
- Recovering from errors
Outline

- Motivation

- Transactional Memory
  - Motivation, use & performance example

- Transactional Coherence & Consistency (TCC)
  - Architecture model, implementation, advanced features

- Performance evaluation
  - SpecJBB2000

- Conclusions & current work
The Stanford Transactional Coherence/Consistency Project

- A hardware-assisted TM implementation
  - Avoids $\geq 2x$ overhead of software-only implementation
  - Semantically correct TM implementation
  - Does not require recompilation of base libraries

- A system that uses TM for coherence & consistency
  - All transactions, all the time
  - Use TM to replace MESI coherence
    - Other proposals build TM on top of MESI
  - Sequential consistency at the transaction level
    - Address the memory model challenge as well

- Research on applications, TM languages, TM system issues, TM architectures, TM prototypes,...
TCC Execution Model

Transactional coherence/consistency with non-blocking guarantees

See [ISCA’04] for details
TCC Implementation

- TM implementation requirements
  - Manage multiple data versions atomic commit or abort
  - Track read-set and write-set for conflict detection

- TCC implementation approach
  - Lazy version management using cache as write buffer
    - Transaction updates merge with memory at commit
    - Good fault tolerance & faster aborts compared to eager
  - Optimistic conflict detection
    - Detect conflicts when one transaction commits
    - Built-in forward progress guarantees
  - Commit also implements coherence/consistency

See [PACT’06] tutorial for alternatives
Example CMP Environment

Similar implementations for other CMP, SMP, cc-NUMA systems
CMP Architecture for TCC

Transactionally Read Bits:

\texttt{ld 0xdeadbeef}

Transactionally Written Bits:

\texttt{st 0xc0ffeebabe}

Commit:
Read pointers from Store Address FIFO, flush addresses with W bits set

Violation Detection:
Compare incoming address to R bits

Other implementations

- Write-back, multi-level caches, directory-based with 2-phase commit
- Hybrid HW/SW approach

See [PACT’05] for details
Virtualization of TCC Hardware [ASPLOS’06]

- Key observation: most transactions are small
  - They fit easily in L1 and L2 caches (see [HPCA’06])

- Space virtualization (cache overflow)
  - Switch to OS-based TM using virtual memory
    - Page-granularity, copies/diffs for versioning and conflicts
    - Transactions can use HW, OS, or both
  - Can handle overflows and paging

- Time virtualization (interrupts, quanta expiration)
  - Short transactions are aborted (faster than virtualization)
  - Interrupts deferred till next transaction commits
    - Otherwise, abort an transaction & reuse
Support for PL & OS Functionality [ISCA’06]

- Challenging issues
  - Interaction with library-based software, I/O, exceptions, & system calls within transactions, error handling, schedulers, conditional synchronization, memory allocators, ...

- Defined complete TM semantics at the ISA level
  - Two-phase commit
  - Transactional handlers for commit/abort/violations
    - All interesting events switch to software handlers
  - Nested transactions (closed and open)
    - Closed: independent rollback & restart for nested transactions
    - Open: independent atomicity and isolation for nested transactions

- Demonstrate TM interaction with rich PL & OS functionality
  - See [ISCA’06] and [PLDI’06] for details
TM Programming with TCC

- Basic approaches
  - Sequential algorithms: use TM for thread-level speculation
  - Parallel algorithm: use TM for non-blocking synchronization

- C-based programming
  - OpenMP extensions for transactional programming
  - Familiar, high-level model for C programmers

- Java-based programming with Atomos [PLDI’06]
  - Replaces synchronized and volatile with atomic
  - Transaction-based conditional waiting
    - Removed wait, notify, and notifyAll
    - Watch sets for efficient implementation
  - Nested transactions, violation handlers, ...

- Other work
  - Performance feedback & tuning environment [ICS’05]
  - TM programming with Python
ATLAS: the 1\textsuperscript{st} TM Hardware Prototype

- 8-core TCC system on BEE2 board (aka RAMP-Red)
  - 100MHz; runs Linux OS
  - 100x faster than a simulator
Outline

- Motivation

- Transactional Memory
  - Motivation, use & performance example

- Transactional Coherence & Consistency (TCC)
  - Architecture model, implementation, advanced features

- Performance evaluation
  - SpecJBB2000

- Conclusions & current work
TM Example: SPECjbb2000

- 3-tier Java benchmark
  - Shared data within and across warehouses
- Parallelized actions within one warehouse
  - Orders, payments, delivery updates, etc on shared data
Sequential Code for NewOrder

TransactionManager::go() {
    // 1. initialize a new order transaction
    newOrderTx.init();
    // 2. create unique order ID
    orderId = district.nextOrderId(); // newID++
    order = createOrder(orderId);
    // 3. retrieve items and stocks from warehouse
    warehouse = order.getSupplyWarehouse();
    item = warehouse.retrieveItem(); // B-tree search
    stock = warehouse.retrieveStock(); // B-tree search
    // 4. calculate cost and update node in stockTable
    process(item, stock);
    // 5. record the order for delivery
    district.addOrder(order); // B-tree update
    // 6. print the result of the process
    newOrderTx.display();
}

- Non-trivial code with complex data-structures
  - Fine-grain locking ➔ difficult to get right
  - Coarse-grain locking ➔ no concurrency
TM Code for NewOrder

TransactionManager::go() {
    atomic { // begin transaction
        // 1. initialize a new order transaction
        // 2. create a new order with unique order ID
        // 3. retrieve items and stocks from warehouse
        // 4. calculate cost and update warehouse
        // 5. record the order for delivery
        // 6. print the result of the process
    } // commit transaction
}

- Whole NewOrder as one atomic transaction
  - 2 lines of code changed

- Also tried nested transactional versions
  - To reduce frequency & cost of violations
  - 2 to 4 additional lines of code
TM Performance for SpecJBB2000

- Simulated TM CMP
  - Stanford’s TCC architecture

- Speedup over sequential
  - Flat transactions: 1.9x
    - Code similar to coarse locks
    - Frequent aborted transactions due to dependencies
  - Nested transactions: 3.9-4.2x
    - Reduced abort cost OR
    - Reduced abort frequency

- See [WTW’06] for details
Conclusions

- **Transactional Memory (TM)**
  - Simple code that scales well on parallel systems
  - Easy to compose, fault recovery, ...

- **Transactional Coherence and Consistency**
  - An efficient hardware-based TM
  - Uses TM to provide coherence & consistency model

- **Current research focus**
  - Hybrid & scalable TM implementations
  - Language and application development work
  - Operating system and error recovery support
  - System-level transactions
Questions?

- Further information and papers available at

http://tcc.stanford.edu