Agenda

Multithreaded Programming

Transactional Memory (TM)

- TM Introduction
- TM Implementation Overview
- Hardware TM Techniques
- Software TM Techniques

Q&A

Transactional Memory Introduction

Ali-Reza Adl-Tabatabai
Programming Systems Lab
Intel Corporation

Transactional memory definition

Memory transaction: A sequence of memory operations that either execute completely (commit) or have no effect (abort)

An “all or nothing” sequence of operations

- On commit, all memory operations appear to take effect as a unit (all at once)
- On abort, none of the stores appear to take effect

Transactions run in isolation

- Effects of stores are not visible until transaction commits
- No concurrent conflicting accesses by other transactions

Similar to database ACID properties

Transactional memory language construct

The basic atomic construct:

lock(L); x++; unlock(L); → atomic {x++;}

Declarative – user simply specifies, system implements "under the hood"

Basic atomic construct universally proposed
- HPCS languages (Fortress, X10, Chapel) provide atomic in lieu of locks
- Research extensions to languages – Java, C#, Atomos, CaML, Haskell, …

Lots of recent research activity
- Transactional memory language constructs
- Compiling & optimizing atomic
- Hardware & software implementations of transactional memory

Example: Java 1.4 HashMap

Fundamental data structure

- Map: Key → Value

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

Not thread safe: don't pay lock overhead if you don't need it

Synchronized HashMap

Java 1.4 solution: Synchronized layer

- Convert any map to thread-safe variant
- Explicit locking – user specifies concurrency

```
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

Transactional HashMap

Transactional layer via an ‘atomic’ construct

- Ensure all operations are atomic
- Implicit atomic directive – system discovers concurrency

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

Transactional HashMap:

- Thread-safe, easy to program
- Good scalability

Transactions: Scalability

Concurrent read operations

- Basic locks do not permit multiple readers
  - Reader-writer locks
- Transactions automatically allow multiple concurrent readers

Concurrent access to disjoint data

- Programmers have to manually perform fine-grain locking
  - Difficult and error prone
  - Not modular
- Transactions automatically provide fine-grain locking

ConcurrentHashMap

Java 5 solution: Complete redesign

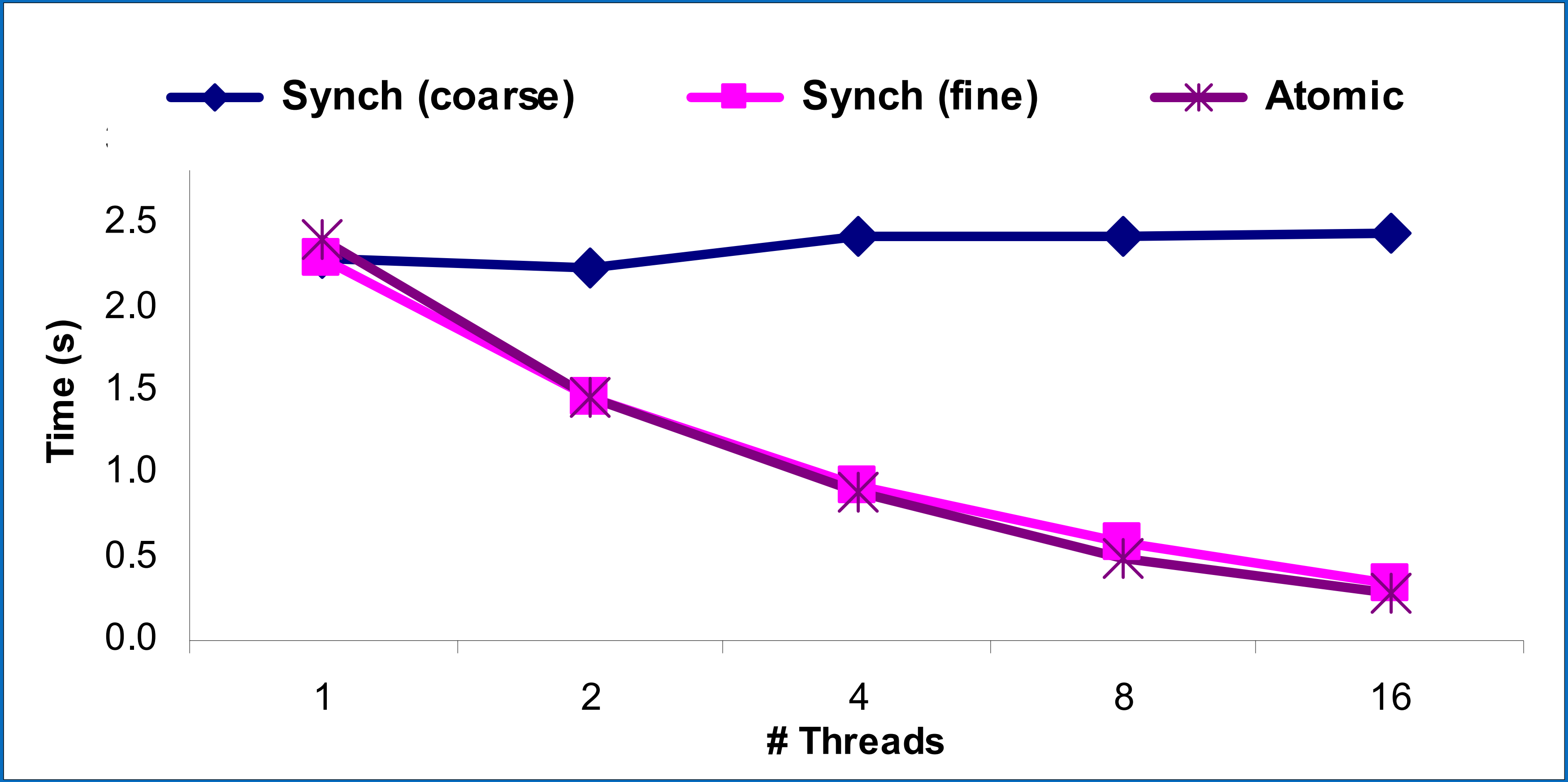
```
public Object get(Object key) {
   int hash = hash(key);
   // Try first without locking...
   Entry[] tab = table;
   int index = hash & (tab.length - 1);
   Entry first = tab[index];
   Entry e;

   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

HashMap performance



Transactions scales as well as fine-grained locks

Transactional memory benefits

As easy to use as coarse-grain locks

Scale as well as fine-grain locks

Composition:

- Safe & scalable composition of software modules

Example: A bank application

Bank accounts with names and balances

- HashMap is natural fit as building block

```
class Bank {
  ConcurrentHashMap accounts;
  …
  void deposit(String name, int amount) {
    int balance = accounts.get(name);          // Get the current balance
    balance = balance + amount;                // Increment it
    accounts.put(name, balance);               // Set the new balance
  }
  …
}
```

Not thread-safe – Even with ConcurrentHashMap

Thread safety

Suppose Fred has $100

T0: deposit("Fred", 10)

- bal = acc.get("Fred") <- 100
- bal = bal + 10
- acc.put("Fred", bal) -> 110

T1: deposit("Fred", 20)

- bal = acc.get("Fred") <- 100
- bal = bal + 20
- acc.put("Fred", bal) -> 120

Fred has $120. $10 lost.

Traditional solution: Locks

```
class Bank {
  ConcurrentHashMap accounts;
  …
  void deposit(String name, int amount) {
    synchronized(accounts) {
      int balance = accounts.get(name);          // Get the current balance
      balance = balance + amount;                // Increment it
      accounts.put(name, balance);               // Set the new balance
    }
  }
  …
}
```

Thread-safe – but no scaling

- ConcurrentHashMap does not help
- Performance requires redesign from scratch & fine-grain locking

Transactional solution

```
class Bank {
  HashMap accounts;
  …
  void deposit(String name, int amount) {
    atomic {
      int balance = accounts.get(name);          // Get the current balance
      balance = balance + amount;                 // Increment it
      accounts.put(name, balance);                // Set the new balance
    }
  }
  …
}
```

Thread-safe – and it scales!

Safe composition + performance

Transactional memory benefits

As easy to use as coarse-grain locks

Scale as well as fine-grain locks

Safe and scalable composition

Failure atomicity:

- Automatic recovery on errors

Traditional exception handling

```
class Bank {
 Accounts accounts;
 …
 void transfer(String name1, String name2, int amount) {
  synchronized(accounts) {
    try {
      accounts.put(name1, accounts.get(name1)-amount);
      accounts.put(name2, accounts.get(name2)+amount);
    }
    catch (Exception1) {..}
    catch (Exception2) {..}
  }
  …
}
```

Manually catch all exceptions and determine what needs to be undone

Side effects may be visible to other threads before they are undone

Failure recovery using transactions

```
class Bank {
  Accounts accounts;
  …
  void transfer(String name1, String name2, int amount) {
    atomic {
        accounts.put(name1, accounts.get(name1)-amount);
        accounts.put(name2, accounts.get(name2)+amount);
    }
  }
  …
}
```

System rolls back updates on an exception

Partial updates not visible to other threads

Challenges in parallel programming

Finding independent tasks

Mapping tasks to threads

Defining & implementing synchronization protocol

Race conditions

Deadlock avoidance

Memory model

Composing parallel tasks

Scalability

Portable & predictable performance

Recovering from errors

... Single thread issues

→ Transactions address a lot of parallel programming problems

Challenges in parallel programming

Finding independent tasks

Mapping tasks to threads

Defining & implementing synchronization protocol

Race conditions

Deadlock avoidance

Memory model

Composing parallel tasks

Scalability

Portable & predictable performance

Recovering from errors

... Single thread issues

→ But not a silver bullet

Summary

Transactions provide many benefits over locks

- Automatic fine-grain concurrency
- Automatic read concurrency
- Deadlock avoidance
- Eliminates locking protocols
- Automatic failure recovery

→ Safe & scalable composition of thread-safe software modules

Challenge: How to implement transactions efficiently?