Transactional Programming In A Multi-core Environment

Ali-Reza Adl-Tabatabai *Intel Corp.*
Christos Kozyrakis *Stanford U.*
Bratin Saha *Intel Corp.*

PACT 2007 Tutorial, September16th, 2007

Presenters

- **Ali-Reza Adl-Tabatabai**, Intel
  - Principal Engineer, Programming Systems Lab at Intel
  - Compilers & runtimes for future Intel architectures
  - PhD. CMU
- **Christos Kozyrakis**, Stanford University
  - Assistant Professor, Electrical Eng. & Computer Science
  - Architectures & programming models for transactional memory
  - Ph.D. UC Berkeley
- **Bratin Saha** , Intel **(absent)**
  - Senior Staff Researcher, Programming Systems Lab
  - Design of highly scalable runtimes for multi-core processors
  - Ph.D. Yale

Tutorial Motivation & Goals

- Motivation
  - Transactions are a good synchronization abstraction
  - How can transactions be used & implemented?

- Goals
  - Introduction to transactional memory
    - A research technology for easier parallel programming
  - Overview, uses, and implementation

Agenda

- Transactional Memory (TM)
  - TM Introduction
  - TM Implementation Overview
  - Hardware TM Techniques
  - Software TM Techniques
- Q&A

Tutorial Slides

- Available on-line at

  http://csl.stanford.edu/~christos/pact07_tm.pdf

TM Bibliography

- Active, online bibliography at
  http://www.cs.wisc.edu/trans-memory

- "Transactional Memory" book by Jim Larus and Ravi Rajwar

- A select list of key papers provided in the following slides

STM & TM Languages References

1. N. Shavit and S. Touitou. Software Transactional Memory.In *Proc. of the 14th Symposium on Principles of Distributed Computing*, Aug. 1995.
2. M. Herlihy et al. Software Transactional Memory for Dynamic-sized Data Structures. In *Proc of the 22nd Symposium on Principles of Distributed Computing*, July 2003.
3. T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *Proc. of the 18th Conference on Object-oriented Programming, Systems, Languages, and Applications*, 2003.
4. V. Marathe, W. Scherer, and M. Scott. Adaptive Software Transactional Memory. In *Proc. of the 19th Intl. Symposium on Distributed Computing*, Sept. 2005.
5. P. Charles et al. X10: An Object-oriented Approach to Nonuniform Cluster Computing. *Proc. of the 20th Conference on Object-oriented Programing, Systems, Languages, and Applications*. Oct. 2005.
6. E. Allen et al. *The Fortress Language Specification*. Sun Microsystems, 2005.
7. *Chapel Specification*. Cray, February 2005.
8. Hudson et al. McRT-Malloc: A Scalable Transaction Aware Memory Allocator. ISMM 2006
9. B. D. Carlstrom et al. The Atomos Transactional Programming Language. In *Proc. of the Conference on Programming Language Design and Implementation*, June 2006.
10. N. Shavit and D. Dice, What Really Makes Transactions Faster. Workshop on Languages, Compilers, and Hardware Support for Transactional Computing, June 2006.
11. B. Saha et al. Implementing a high performance software transactional memory. In Proc. of the Conference on Principles and Practices of Parallel Processing, March, 2006
12. A. Adl-Tabatabai et al, Compiler and runtime support for efficient software transactional memory. In *Proc of the Conference on Programming Language Design and Implementation*, June 2006.
13. T. Harris et al. Optimizing Memory Transactions. In *Proc of the Conference on Programming Language Design and Implementation*, June 2006.

STM & TM Languages References

1. B. Carlstrom et al. Transactional Collection Classes. In PPoPP 2007.
2. Y. Ni et al. Open Nesting in Software Transactional Memory. In PPoPP 2007.
3. C. Wang et. al. Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language. In CGO 2007.
4. T. Shpeisman et al. Enforcing Isolation and Ordering in STM. In PLDI 2007.

HTM & Hybrid-TM References

1. T. Knight. An Architecture for Mostly Functional Languages. In *Proc. of the ACM Conference on LISP and Functional Programming*, 1986.
2. M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proc. of the 20th Annual Intl. Symp. on Computer Architecture*,, May 1993
3. R. Rajwar and J. R. Goodman. Transactional Lock-Free Execution of Lock-Based Programs. In *Proc. of the 10th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.
4. L. Hammond, et al. Transactional Memory Coherence and Consistency. In *Proc. Of the 31st Annual Intl. Symp. on Computer Architecture*, June 2004.
5. L. Hammond, et. al. Programming with transactional coherence and consistency. In *Proc. of the 11th Intl. Conference on Architecture Support for Programming Languages and Operating Systems*, Oct. 2004.
6. S. Ananian et al. Unbounded Transactional Memory. In *Proc. of the 11th Intl. Symposium on High Performance Computer Architecture*, Feb. 2005.
7. R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *Proc. of the 32nd Annual Intl. Symp. On Computer Architecture*, Jun. 2005.
8. C. Blundell, et al. Deconstructing Transactional Semantics: The Subtleties of Atomicity. In *ISCA Workshop on Duplicating, Deconstructing, and Debunking*, June 2005.
9. B. Saha, et. al. Architecture Support for Software Transactional Memory. Micro 2006.

HTM & Hybrid-TM References

9. A. McDonald et al. Characterization of TCC on Chip-Multiprocessors. In *Proc. of the 14th Intl. Conference on Parallel Architectures and Compilation Techniques*, Sept. 2005.
10. K. Moore et al. LogTM: Log-Based Transactional Memory. In *Proc. of the 12th Intl. Conference on High Performance Computer Architecture*, Feb. 2006.
11. S. Kumar et al. Hybrid Transactional Memory. In Proc. of the Conference on Principles and Practices of Parallel Processing, March, 2006
12. J. Chung et al. The Common Case Transactional Behavior of Multithreaded Programs. In Proc. of the 12th Intl. Conference on High Performance Computer Architecture, Feb. 2006.
13. A. Sriraman et al. Hardware Acceleration of hardware Transactional Memory, Workshop on Languages, Compilers, and Hardware Support for Transactional Computing, June 2006.
14. L. Ceze et al. Bulk Disambiguation of Speculative Threads in Multiprocessors, In Proc.of the 32nd Intl. Symposium on Computer Architecture, June 2006.
15. A. McDonald et al. Architectural Semantics for Practical Transactional Memory, In Proc.of the 32nd Intl. Symposium on Computer Architecture, June 2006.
16. P. Damron et al. Hybrid Transactional Memory, In Proc. Of the 12th Intl. *Conference on Architecture Support for Programming Languages and Operating Systems*, Oct. 2006.
17. J. Chung et al. Tradeoffs in Transactional Memory Virtualization , In Proc. of the 12th Intl. *Conference on Architecture Support for Programming Languages and Operating Systems*, Oct. 2006.
18. C. Minh et al. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In ISCA 2007.
19. W. Chuang et al. Unbounded Page-Based Transactional Memory. In ASPLOS 2006.
20. J. Bobba et al. Performance Pathologies in Hardware Transactional Memory. In ISCA 2007

Agenda

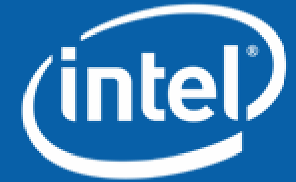
Transactional Memory (TM)

- TM Introduction

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

Q&A

© Ali-Reza Adl-Tabatabai & Bratin Saha
intel



Transactional Memory Introduction

Ali-Reza Adl-Tabatabai
Programming Systems Lab
Intel Corporation

Multi-core: An inflection point in SW

Multi-core architectures: an inflection point in mainstream SW development

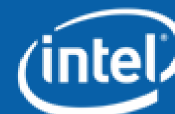
Writing parallel SW is hard

- Mainstream developers not used to thinking in parallel
- Mainstream languages force the use of low-level concurrency features

Navigating through this inflection point requires better concurrency abstractions

Transactional memory: an alternative to locks for concurrency control

© Ali-Reza Adl-Tabatabai & Bratin Saha



Transactional memory definition

Memory transaction: A sequence of memory operations that execute atomically and in isolation

Atomic: 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

Execute as if in a single step with respect to other threads

© Ali-Reza Adl-Tabatabai & Bratin Saha

Transactional memory language construct

The basic atomic construct:

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

User simply specifies, system implements “under the hood”

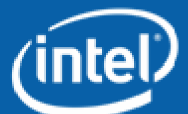
Basic atomic construct universally proposed

- 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 and software implementations of transactional memory

© Ali-Reza Adl-Tabatabai & Bratin Saha



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

© Ali-Reza Adl-Tabatabai & Bratin Saha

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

© Ali-Reza Adl-Tabatabai & Bratin Saha
intel

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

© Ali-Reza Adl-Tabatabai & Bratin Saha

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

© Ali-Reza Adl-Tabatabai & Bratin Saha

ConcurrentHashMap

Java 5 solution: Complete redesign

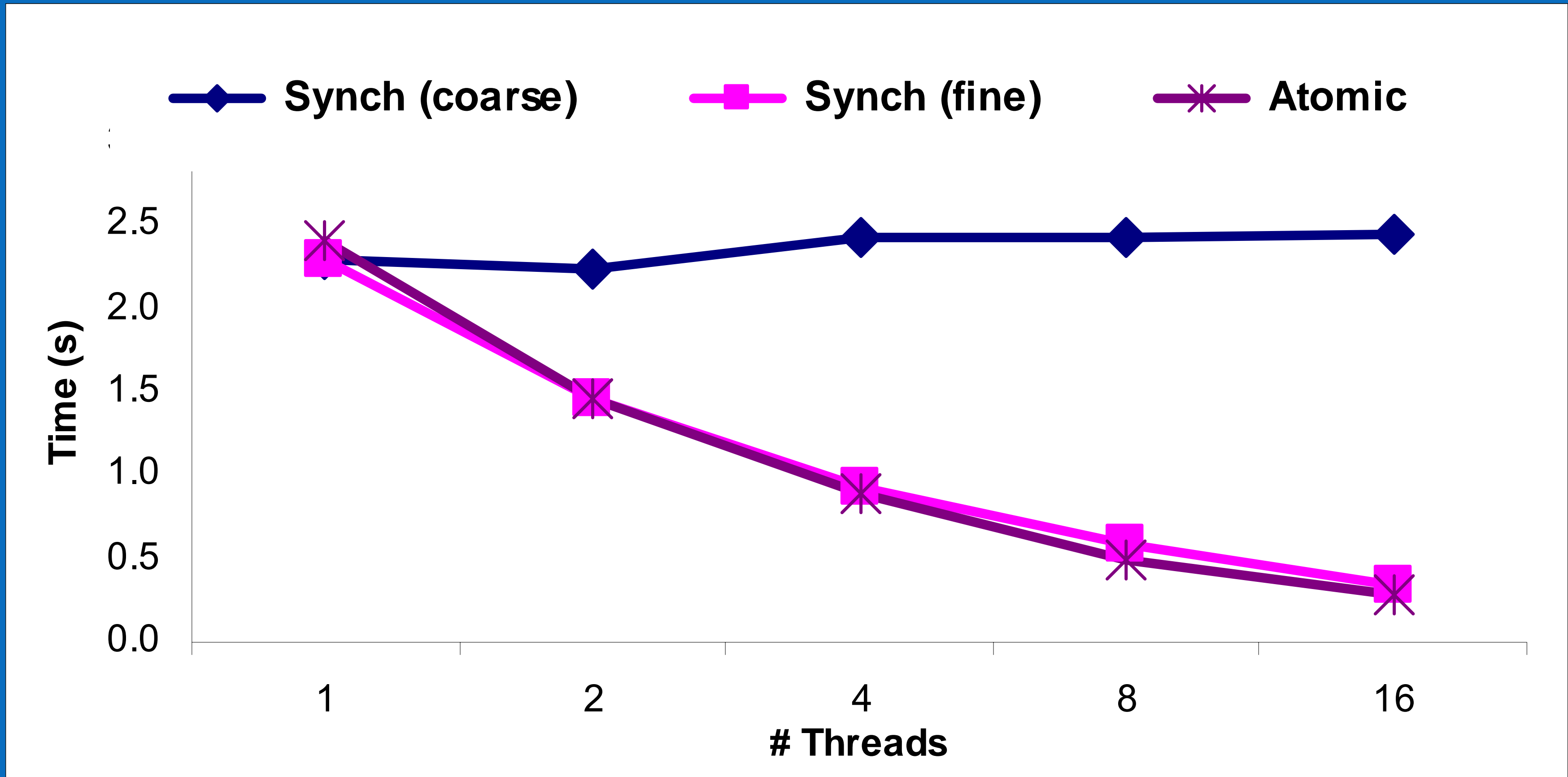
```
public Object get(Object key) {
  int hash = hash(key);
  // Try first without locking...
  Entry[] tab = table;
  int index = hash & (tab.length - 1);
  Entry first = tab[index];
  Entry e;

  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

© Ali-Reza Adl-Tabatabai & Bratin Saha

HashMap performance



Transactions scales as well as fine-grained locks

© Ali-Reza Adl-Tabatabai & Bratin Saha

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

© Ali-Reza Adl-Tabatabai & Bratin Saha
intel

Example: A bank application

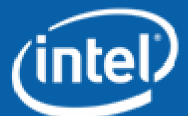
Bank accounts with names and balances

- 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

Transactional memory tutorial
© Ali-Reza Adl-Tabatabai & Bratin Saha



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.

© Ali-Reza Adl-Tabatabai & Bratin Saha

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

Fine-grain locking does not compose

© Ali-Reza Adl-Tabatabai & Bratin Saha

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

© Ali-Reza Adl-Tabatabai & Bratin Saha

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

© Ali-Reza Adl-Tabatabai & Bratin Saha

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

© Ali-Reza Adl-Tabatabai & Bratin Saha

intel

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

© Ali-Reza Adl-Tabatabai & Bratin Saha

Condition synchronization using locks

```
Object blockingDequeue(...) {
    synchronized (this) {
        // Block until queue has item
        while (isEmpty()) {
            try {
              this.wait();
            } catch(InterruptedException ie) { }
        }
        return dequeue();
} }
```

Lock-based condition synchronization uses **wait** & **notify**

Enqueue() must explicitly **notify** to wake up blocking thread

Forgetting the notify causes a **lost wakeup bug**

Recheck isEmpty() in a loop because of **spurious wakeups**

Transactional memory tutorial
© Ali-Reza Adl-Tabatabai & Bratin Saha
intel

Condition synchronization with transactions

```
Object blockingDequeue(...) {
  // Block until queue has item
   atomic {
      if (isEmpty())
         retry;
      return dequeue();
   }
}
```

retry

- Rolls back (nested) transaction
- Waits for change in memory state
- Store by another thread implicitly signals blocked thread
  → No lost wakeups
- See Harris et al PPoPP 2005 & Adl-Tabatabai et al PLDI 2006

© Ali-Reza Adl-Tabatabai & Bratin Saha

Conditional atomic regions

```
Object blockingDequeue(...) {
  // Block until queue has item
    when (!isEmpty())
        return dequeue();
}
```

when

- Blocks until condition holds
- See Harris & Fraser OOPSLA 2003 and IBM X10 paper in OOPSLA 2005

© Ali-Reza Adl-Tabatabai & Bratin Saha

Composing alternatives

```
atomic {
  q1.blockingDequeue();
} orelse {
  q2.blockingDequeue();
} orelse {
  q3.blockingDequeue();
}
```

orelse

- Execute exactly one clause atomically
- Left-bias: Try in order
- User retry: Try next alternative

→ Allows composition of alternatives

- See Harris et al PPoPP 2005 & Adl-Tabatabai et al PLDI 2006

© Ali-Reza Adl-Tabatabai & Bratin Saha

Summary

Multicore: an inflection point in mainstream SW development

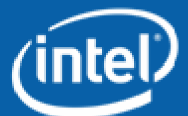
Navigating inflection requires new language abstractions

- Safety
- Scalability & performance
- Modularity

Transactional memory enables safe & scalable composition of software modules

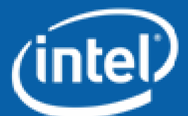
- Automatic fine-grained & read concurrency
- Avoids deadlock
- Automatic failure recovery
- Avoids lost wakeups, allows composition of alternatives

© Ali-Reza Adl-Tabatabai & Bratin Saha



Questions?

© Ali-Reza Adl-Tabatabai & Bratin Saha



Agenda

- Transactional Memory (TM)
  - TM Introduction
  - TM Implementation Overview
  - Hardware TM Techniques
  - Software TM Techniques
- Q&A

Transactional Memory Implementation Overview

Christos Kozyrakis

Computer Systems Laboratory
Stanford University
http://csl.stanford.edu/~christos

TM Implementation Requirements

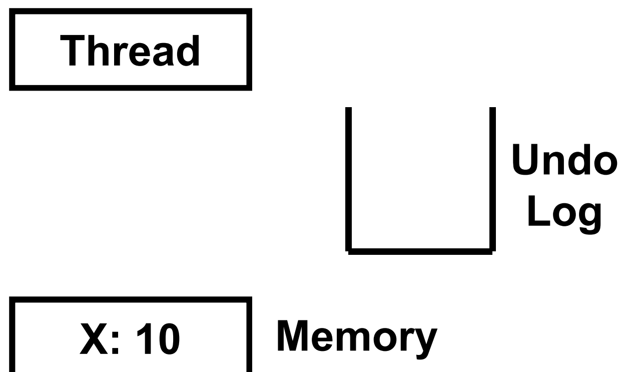
- TM implementation must provide atomicity and isolation
  - Without sacrificing concurrency
- Basic implementation requirements
  - Data versioning
  - Conflict detection & resolution
- Implementation options
  - Hardware transactional memory (HTM)
  - Software transactional memory (STM)
  - Hybrid transactional memory

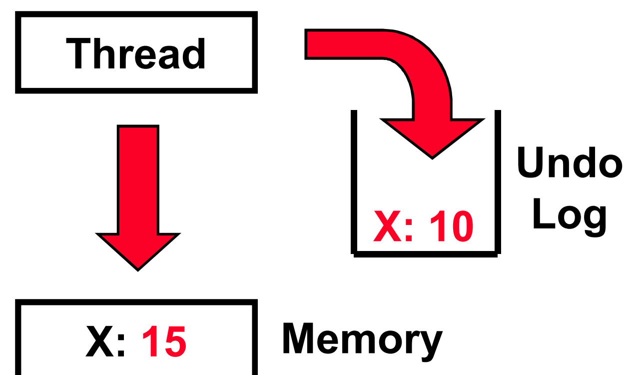
Data Versioning

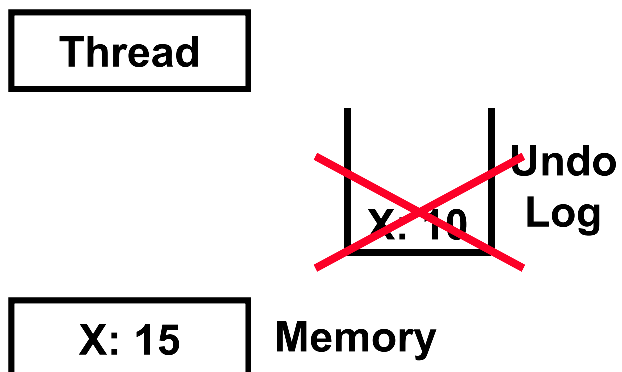
- Manage uncommited (new) and commited (old) versions of data for concurrent transactions

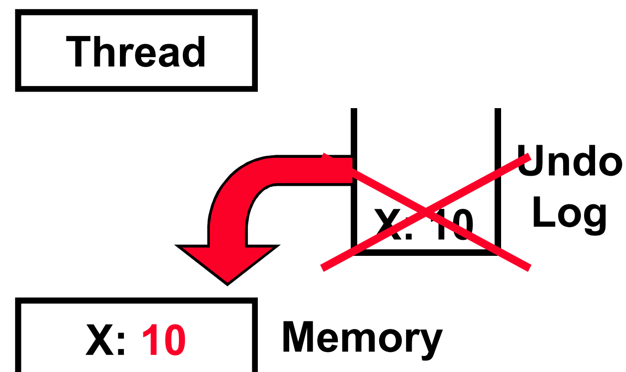
1. Eager (undo-log based)
   - Update memory location directly; maintain undo info in a log
   - \+ Faster commit, direct reads (SW)
   - – Slower aborts, fault tolerance issues
2. Lazy (write-buffer based)
   - Buffer writes until commit; update memory location on commit
   - \+ Faster abort, no fault tolerance issues
   - – Slower commits, indirect reads (SW)

Eager Versioning Illustration

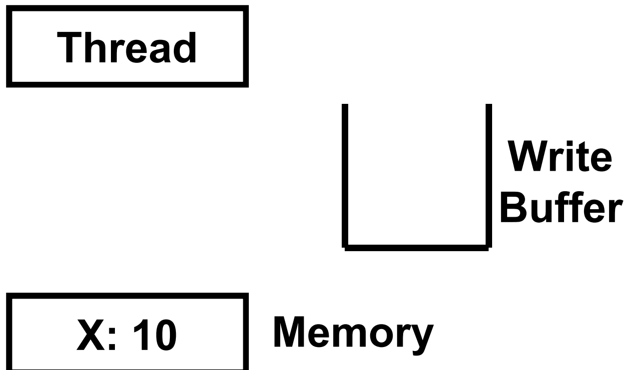


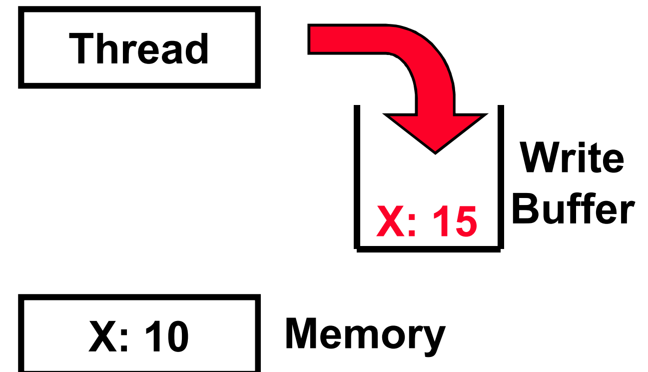


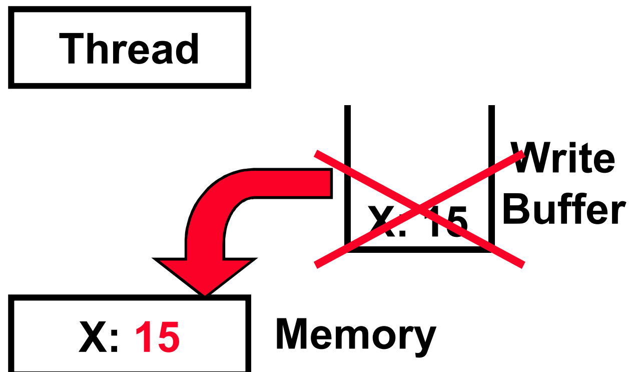


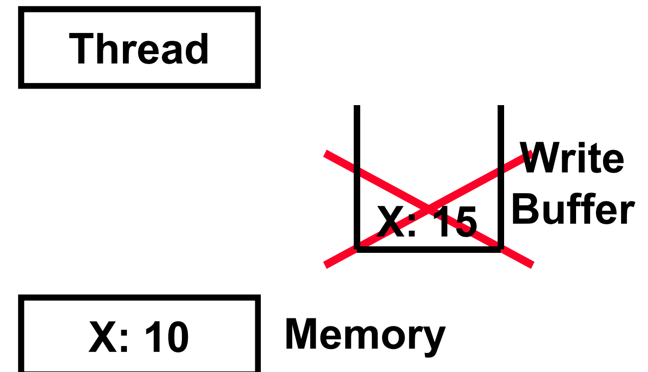


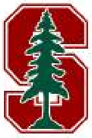
Lazy Versioning Illustration











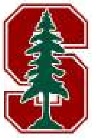
Conflict Detection

- Detect and handle conflicts between transaction
  - Read-Write and (often) Write-Write conflicts
  - For detection, a transactions tracks its read-set and write-set

1. Pessimistic detection
   - Check for conflicts during loads or stores
     - HW: check through coherence lookups
     - SW: checks SW barriers using locks and/or version numbers
   - Use contention manager to decide to stall or abort
     - Various priority policies to handle common case fast
2. Optimistic detection
   - Detect conflicts when a transaction attempts to commit
     - HW: write-set of committing transaction compared to read-set of others
       - Committing transaction succeeds; others may abort
     - SW: validate write-set and read-set using locks and version numbers

- Can use separate mechanism for loads & stores (SW)

Pessimistic Detection Illustration



Optimistic Detection Illustration

TIME →

Case 1	Case 2	Case 3	Case 4
X0: rd A, wr C, commit → check X1: wr B, commit → check	X0: wr A, commit → check X1: rd A, restart, rd A, commit → check	X0: rd A, commit → check X1: wr A, commit → check	X0: rd A, wr A, restart, rd A, wr A, commit → check X1: rd A, wr A, commit → check
Success	**Abort**	**Success**	**Forward progress**

© Christos Kozyrakis

Conflict Detection Tradeoffs

1. Pessimistic conflict detection (aka encounter or eager)
   - \+ Detect conflicts early
     - Undo less work, turn some aborts to stalls
   - – No forward progress guarantees, more aborts in some cases
   - – Locking issues (SW), fine-grain communication (HW)

2. Optimistic conflict detection (aka commit or lazy)
   - \+ Forward progress guarantees
   - \+ Potentially less conflicts, no locking (SW), bulk communication (HW)
   - – Detects conflicts late, still has fairness problems

- Contention management important with both approaches

TM Performance Pathologies

- Pessimistic conflict detection
  - Starving writer: abort writer due to frequent readers
  - Friendly fire: abort due to an xaction that later aborts
  - Futile stall: stall due to an xaction that later aborts
  - Dueling upgrades: concurrent read-modify-writes
- Optimistic conflict detection
  - Starving elder: long xaction aborted by small xactions
  - Restart convoy: convoying due to dependencies to one xaction
- See details Bobba's paper in [ISCA'07]
- Pathologies dealt with contention management
  - Some cases are simple
  - E.g., prioritize starving elder or back off on convoying

Implementation Space

Conflict Detection	Version Management: Eager	Version Management: Lazy
Pessimistic	HW: UW LogTM SW: Intel McRT, MS-STM	HW: MIT LTM, Intel VTM SW: MS-OSTM
Optimistic		HW: Stanford TCC SW: Sun TL/2

[This is just a subset of proposed implementations]

- No convergence yet
- Decision will depend on
  - Application characteristics
  - Importance of fault tolerance, complexity
  - Success of contention managers
- May have different approaches for HW, SW, and hybrid
  - It may not even matter…

Conflict Detection Granularity

- Object granularity (SW/hybrid)
  - + Reduced overhead (time/space)
  - + Close to programmer's reasoning
  - – False sharing on large objects (e.g. arrays)
    - – Unnecessary aborts
- Word granularity
  - + Minimize false sharing
  - – Increased overhead (time/space)
- Cache line granularity
  - + Compromise between object & word
  - + Works for both HW/SW

- Mix & match ➔ best of both words
  - Word-level for arrays, object-level for other data, …

Isolation to Non-Transactional Code

```
P1  . . .                      P2  . . .
    atomic {                       . . .
      write X';                    . . .
      . . .                        read X;
      write X";                    . . .
    }                              . . .
```

- Are transactions atomic with respect to non-transactional accesses?
  - Yes ➔ strong isolation; No ➔ weak atomicity
- More complicated in practice (see [PLDI'07])
  - Non-repeatable reads, lost updates, dirty reads, speculative lost updates, speculative dirty reads, overlapped writes, …
- Strong isolation is simpler from programmer's perspective
  - Otherwise there can be unexpected or unpredictable results
  - HTMs naturally build strong isolation on top of coherence events
  - STMs require additional barriers [PLDI'07] or HW filters [ISCA'07]

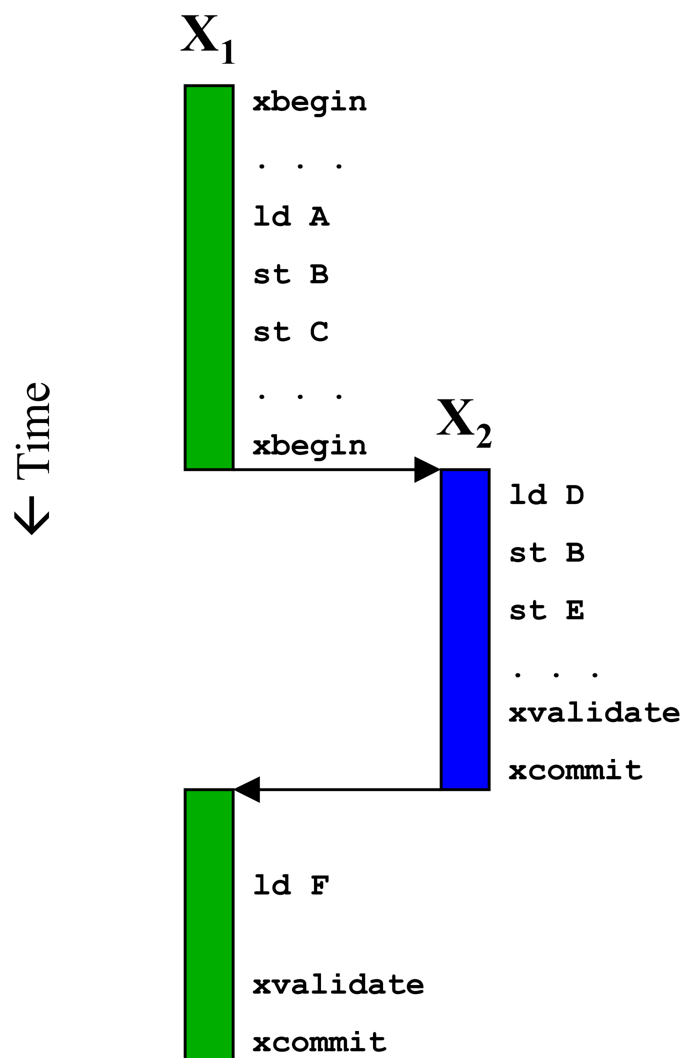
Interactions with PL & OS

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

- Necessary TM semantics
  1. Two-phase commit
     - Separate validation from commit
  2. Transactional handlers for commit/abort/conflict
     - All interesting events switch to software handlers
     - Mechanisms for registering software handlers
  3. Support for nested transactions
     - Closed: independent rollback & restart for nested transactions
     - Open: independent atomicity and isolation for nested transactions

- See McDonald's paper in [ISCA'06]

Closed Nested Transactions



X_1 State

Read-set	A, D, F
Write-Set	B_2, C_1 , E_2

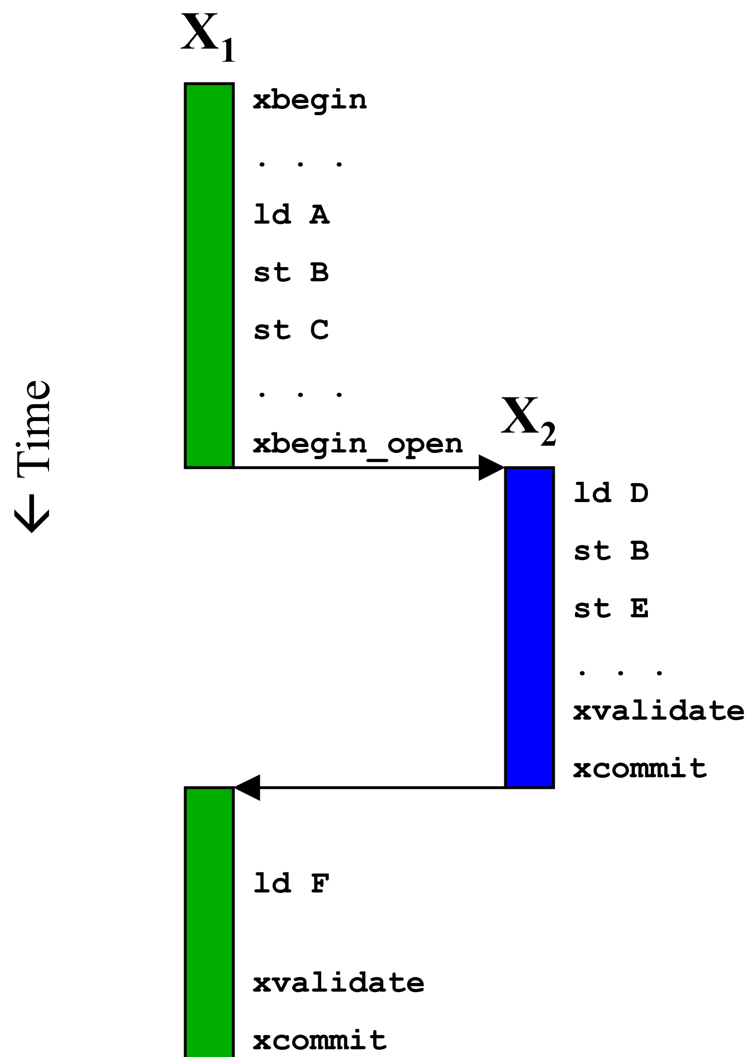
X_2 State

Read-set	D
Write-Set	B_2, E_2

Shared Memory

Address	A	B	C	D	E	F
Value	A_0	B_2	C_1	D_0	E_2	F_0

Open Nested Transactions



X_1 State

Read-set	A, F
Write-Set	B_2, C_1

X_2 State

Read-set	D
Write-Set	B_2, E_2

Shared Memory

Address	A	B	C	D	E	F
Value	A_0	B_2	C_0	D_0	E_2	F_0

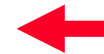
© Christos Kozyrakis

Nested Transactions Summary

- Closed nesting
  - Independent rollback and restart
    - Read-set and write-set tracked independently from parent
    - On inner conflict, abort inner transaction but not outer
    - On inner commit, merge with parent's read-set and write-set
  - Uses: reduce cost of conflict, allow alternate execution paths
- Open nesting
  - Independent atomicity and isolation for nested transactions
    - On inner commit, shared memory is updated immediately
    - Independent rollback similar to closed nesting
  - Uses: reduce frequency of conflicts, scalable & composable libraries, system and runtime code
    - See [ISCA'06], [PLDI'06], and two papers in [PPoPP'07]
    - But, may be too tricky for end programmers

Questions?

Agenda

- Transactional Memory (TM)
  - TM Introduction
  - TM Implementation Overview
  - Hardware TM Techniques 
  - Software TM Techniques
- Q&A

HTM: Hardware Transactional Memory Implementations

Christos Kozyrakis

Computer Systems Laboratory
Stanford University
http://csl.stanford.edu/~christos

Why Hardware Support for TM

- Performance
  - Software TM starts with a 40% to 2x overhead handicap
- Features
  - Strong isolation is there by default
  - Works for all binaries and libraries wo/ need to recompile
  - Depending on the implementation
    - Word-level conflict detection, forward progress guarantees, …

- How much HW support is needed?
  - This is the topic of ongoing research
  - All proposed HTMs are essentially hybrid
    - Add flexibility by switching to software on all interesting events

HTM Mechanisms Summary

- Data versioning in caches
  - Cache the write-buffer or the undo-log
  - Zero overhead for both loads and stores
    - The cache HW handles versioning and detection transparently
  - Can do with private, shared, and multi-level caches
- Conflict detection through some cache coherence protocol
  - Coherence lookups detect conflicts between transactions
  - Works with snooping & directory coherence
- Notes
  - Register checkpoint must be taken at transaction begin
  - Virtualization of hardware resources discussed later
  - HTM support similar to that for thread-level speculation (TLS)
    - Some HTMs support both TM and TLS

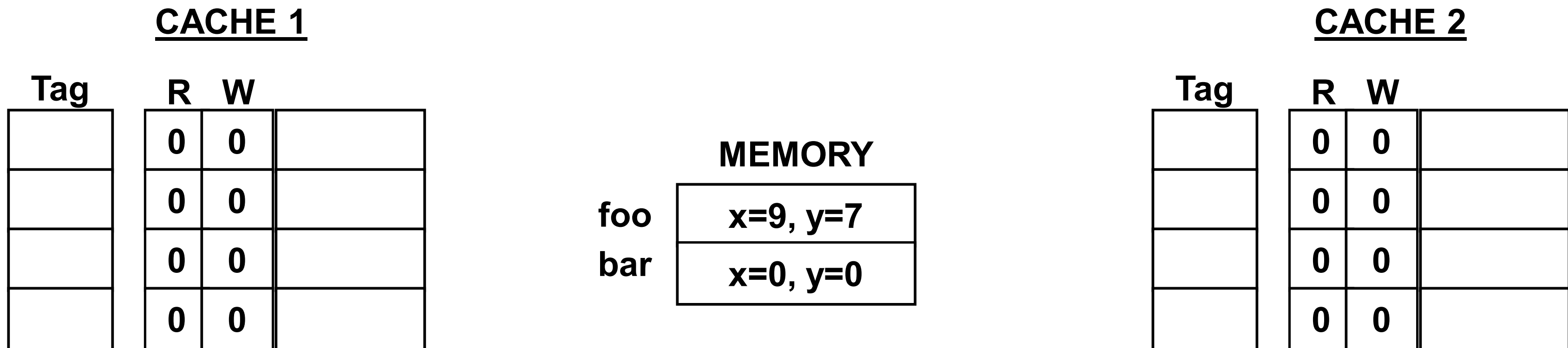
HTM Design

- Cache lines annotated to track read-set & write set
  - R bit: indicates data read by transaction; set on loads
  - W bit: indicates data written by transaction; set on stores
    - R/W bits can be at word or cache-line granularity
  - R/W bits gang-cleared on transaction commit or abort
  - For eager versioning, need a 2nd cache write for undo log

- Coherence requests check R/W bits to detect conflicts
  - Shared request to W-word is a read-write conflict
  - Exclusive request to R-word is a write-read conflict
  - Exclusive request to W-word is a write-write conflict (may be OK)

© Christos Kozyrakis

HTM Example (Lazy, Optimistic)



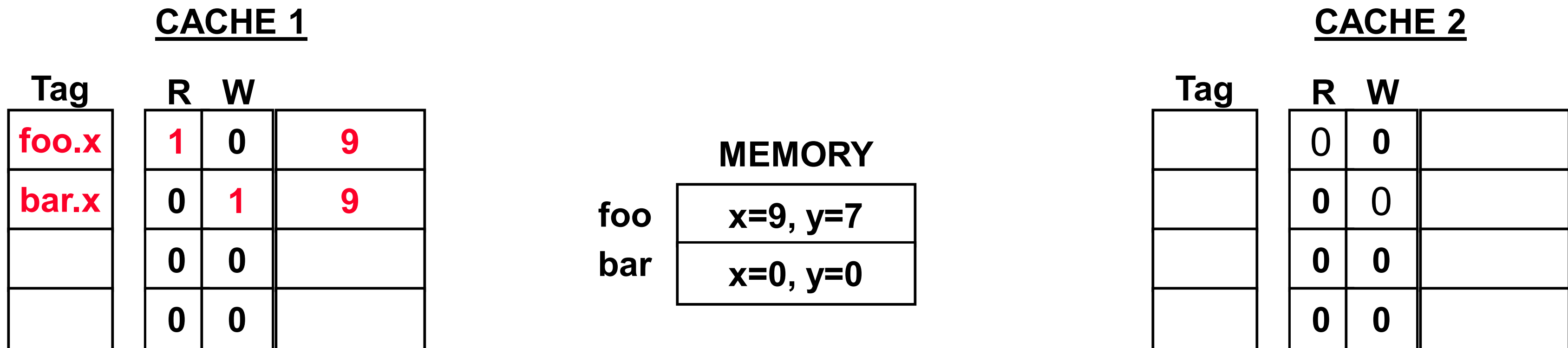
T1
```
atomic {
    bar.x = foo.x;
    bar.y = foo.y;
}
```

T2
```
atomic {
    t1 = bar.x;
    t2 = bar.y;
}
```

- T1 copies **foo** into **bar**
- T2 should read [0, 0] or should read [9,7]

© Christos Kozyrakis

HTM Example (1)



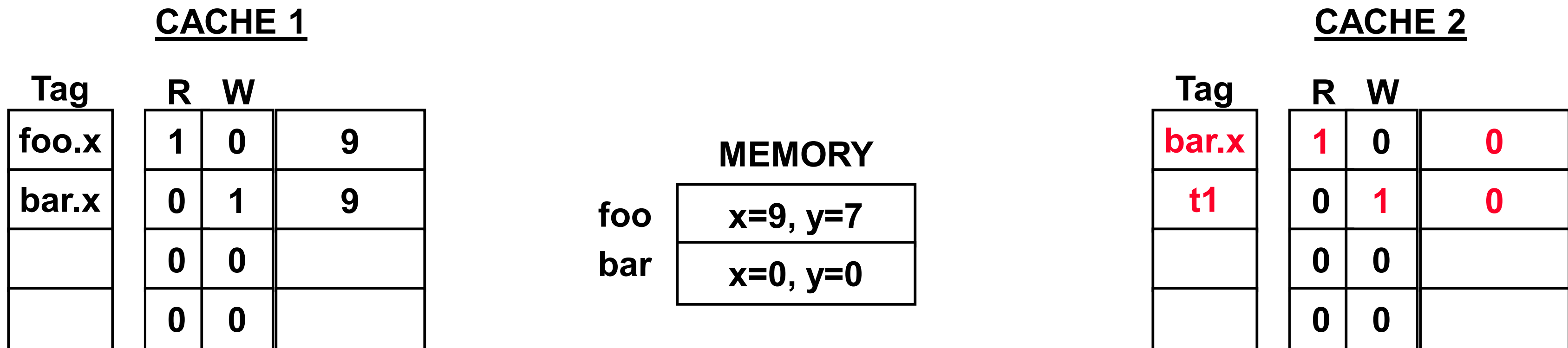

```
T1 atomic {
       bar.x = foo.x;
       bar.y = foo.y;
   }
```

```
T2 atomic {
       t1 = bar.x;
       t2 = bar.y;
   }
```

- Both transactions make progress independently

© Christos Kozyrakis

HTM Example (2)




```
T1 atomic {
       bar.x = foo.x;   ←
       bar.y = foo.y;
   }
```

```
T2 atomic {
       t1 = bar.x;   ←
       t2 = bar.y;
   }
```

- Both transactions make progress independently

HTM Example (3)

CACHE 1

Tag	R	W	
foo.x	1	0	9
bar.x	0	1	9
foo.y	1	0	7
bar.y	0	1	7

MEMORY

foo	x=9, y=7
bar	x=0, y=0

CACHE 1

Tag	R	W	
bar.x	1	0	0
t1	0	1	0
	0	0	
	0	0	

```
T1 atomic {
       bar.x = foo.x;
       bar.y = foo.y;   ⟵
   }
```

```
T2  atomic {
       t1 = bar.x;   ⟵
       t2 = bar.y;
    }
```

- Transaction T1 is now ready to commit

HTM Example (3)

CACHE 1

Tag	R	W	
foo.x	0	0	9
bar.x	0	0	9
foo.y	0	0	7
bar.y	0	0	7

Excl bar.x
Excl bar.y

MEMORY

foo	x=9, y=7
bar	x=9, y=7

CACHE 2

Tag	R	W	
bar.x	1	0	0
t1	0	1	0

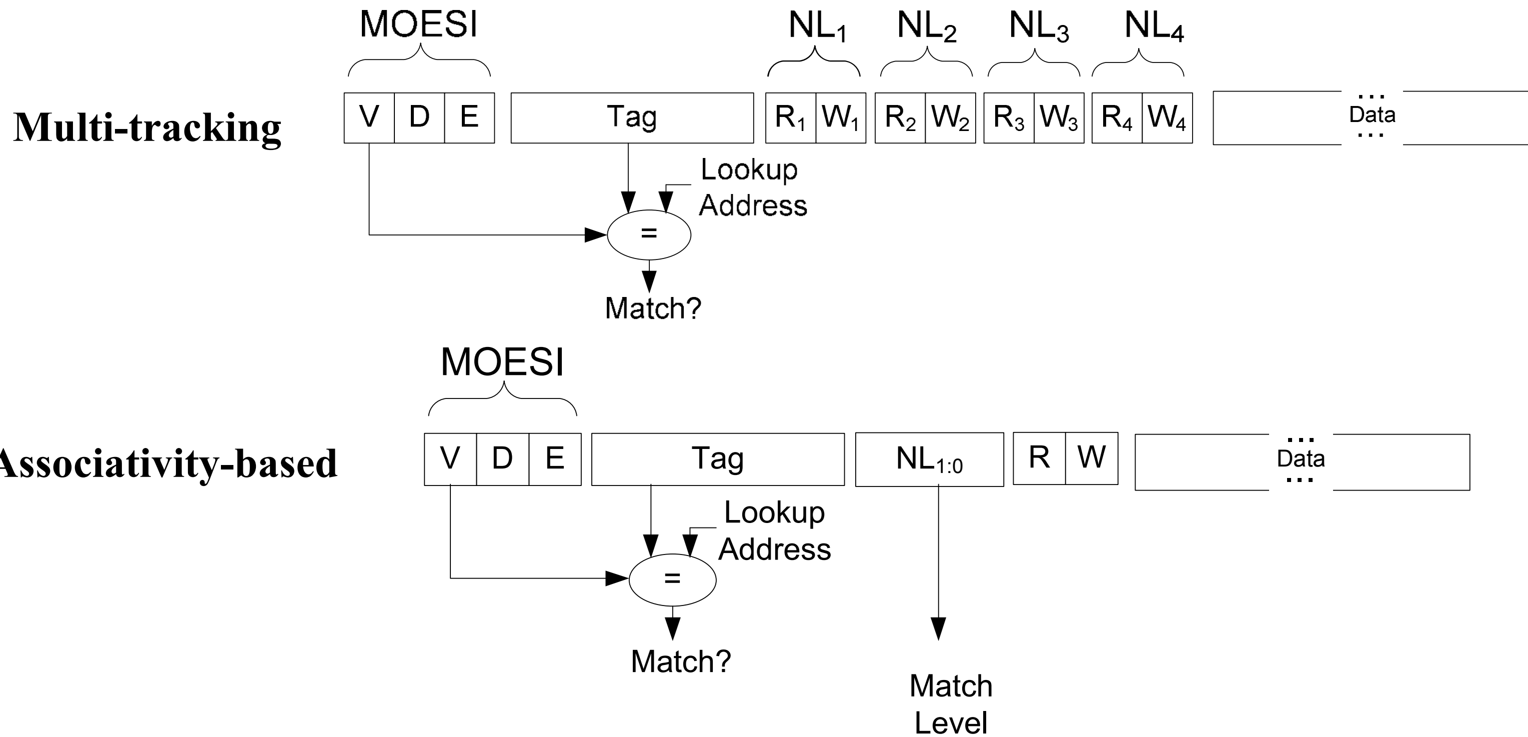
Conflict

```
T1 atomic {
      bar.x = foo.x;
      bar.y = foo.y;
   }
```

```
T2   atomic {
       t1 = bar.x;
       t2 = bar.y;
     }
```

- T1 updates shared memory
  - R/W bits are cleared
  - This is a logical update, data may stay in caches as dirty
- Exclusive request for bar.x reveals conflict with T2
  - T2 is aborted & restarted; all R/W cache lines are invalidated
  - When it reexecutes, it will read [9,7] without a conflict

Support for Nested Transactions



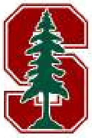
- Caches track read-sets & write-sets multiple transactions
  - Multi-tracking for eager versioning, associativity best for lazy
  - Gange-merge or lazy merge at inner commit
- See paper by McDonald at [ISCA'06] for details
  - Including HW and SW interactions around nesting

HTM Virtualization

- Space virtualization ➔ What if caches overflow?
  - Where is the write-buffer or log stored?
  - How are R & W bits stored and checked?
- Time virtualization ➔ What if time quanta expires?
  - Interrupts, paging, and thread migrations half-way through transactions
- Nesting virtualization ➔ What if nesting level exhausted?
- Observations: most transactions are currently small
  - Small read-sets & write-sets, short in terms of instructions, nesting is uncommon
  - See paper by Chung at [HPCA'06]

Time Virtualization

- Three-tier interrupt handling for low overhead
  1. Defer interrupt until next short transaction commits
     - Use that processor for interrupt handling
  2. If interrupt is critical, rollback youngest transaction
     - Most likely, the re-execution cost is very low
  3. If a transaction is repeatedly rolled back due to interrupts
     - Use space virtualization to swap out (typically higher overhead)
     - Only needed when most threads run very long transactions
- Key assumption
  - Rolling back a short transaction is cheaper than virtualizing it
- See paper by Chung at [ASPLOS'06]



Space Virtualization

- Virtualized TM (Rajwar @ [ISCA'05])
  - Map the write-buffer & read/write-set in virtual memory
    - They become unbounded; they can be at any physical location
  - Caches capture working set of write-buffer/undo-log
    - Hardware and firmware handle misses, relocation, etc
    - Bloom filters used to reduce lookups in virtual memory
- eXtended TM (Chung @ [ASPLOS'06])
  - Use OS virtualization capabilities (virtual memory)
    - On overflow, use page-based TM ➔ no HW/firmware needed
    - Similar to page-based DSM, but used only as a back up
    - Overflow either all transaction state or just a part of it
  - Works well when most transactions are small
- Page-based TM (Chuang @ [ASPLOS'06]
  - Similar to XTM but hardware manages overflow metadata
  - Requires new HW caches at the memory controller level

Hybrid TM Implementations

- Combine the best of both worlds
  - Performance of HTM; virtualization, cost, and flexibility of STM
- Dual TM implementations [PPoPP'06, ASPLOS'06]
  - Start transaction in HTM; switch to STM on overflow, abort, …
  - Typically requires 2 versions of the code
  - Carefully handle interactions between HTM & STM transactions
- HW accelerated STM (HASTM [Micro'06])
  - Provide key primitives for STM code to use
    - Add SW controlled mark bits to cache lines (private, non-persistent)
    - Focusing mostly on read/write-set tracking, not version management
  - Enables SW to build powerful filters for read/write barriers
    - Have I accessed this address before? Has anyone modified it?
    - If transaction fits in cache, this is close to HTM speed
  - There is still a SW path to guarantee correct operation in all cases

Bulk Disambiguation (Ceze @ [ISCA'06])

- HTM that tracks read-sets and write-sets using signatures
  - HW bloom filters replace R and W bits in caches
    - One filter for read-set, one for write-set, etc
    - Filters are updated on loads/stores, checked on coherence traffic
  - Filters can be swapped to memory, transmitted to other processors, …
    - Simple compression can reduce filter size significantly

- Tradeoffs
  - \+ Decouples cache from read-set/write-set tracking
    - Same cache design, non overflow for R and W bits
  - – Inexact operations can lead to false conflicts
    - May lead to degradation, depending on application behavior and HW details
  - – Still, there are virtualization challenges
    - Coherence messages must reach filter even if cache does not hold the line
    - Challenge for non-broadcast coherence schemes

Signature-based STM (Cao Minh @ [ISCA'07])

- Combines Bulk disambiguation + HASTM approaches
  - Based on an STM system with HW acceleration
  - HW filters to track read-set & write-set
    - No other changes to caches (write-buffer or log in SW)
  - Single code path (no fast path and slow path)
- SigTM benefits
  - Performance similar to HTM
    - 2x over STM, within 10% to 40% of HTM
  - Strong atomicity
    - Coherence requests are looked up in hardware filters
    - No modifications to non-transactions code
  - Simplified nesting support
    - Through saving/restoring the filters on nested begin and abort

Transactional Coherence

- Key observation
  - For well synchronized programs, coherence & consistency needed only at transaction boundaries
- Transactional Coherence & Consistency (TCC)
  - Eliminate MESI coherence protocol
  - Coherence using the R/W bits only
    - Fewer/simpler states; multiple writers are allowed
  - Communication logically only at commit points
- Characteristics
  - Sequential consistency at transaction boundaries
  - Coarser-grain communication
  - Bulk coherence creates hybrid between shared-memory and message passing
- See TCC papers at [ISCA'04], [ASPLOS'04], & [PACT'05]

```
foo() {
  work1();
  atomic {
     a.x = b.x;
     a.y = b.y;
  }
  work2();
}
```

Performance Example: SpecJBB2000

Client Tier | Transaction Server Tier | Database Tier

Driver Threads
Driver Threads
Transaction Manager
District
newID
Warehouse
Warehouse
orderTable (B-Tree)
itemTable (B-Tree)
stockTable (B-Tree)
itemTable (B-Tree)
stockTable (B-Tree)

- 3-tier Java benchmark
- Shared data within and across warehouses
  - B-trees for database tier
- Can we parallelize the actions within a warehouse?
  - Orders, payments, delivery updates, etc

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

Transactional Code for NewOrder

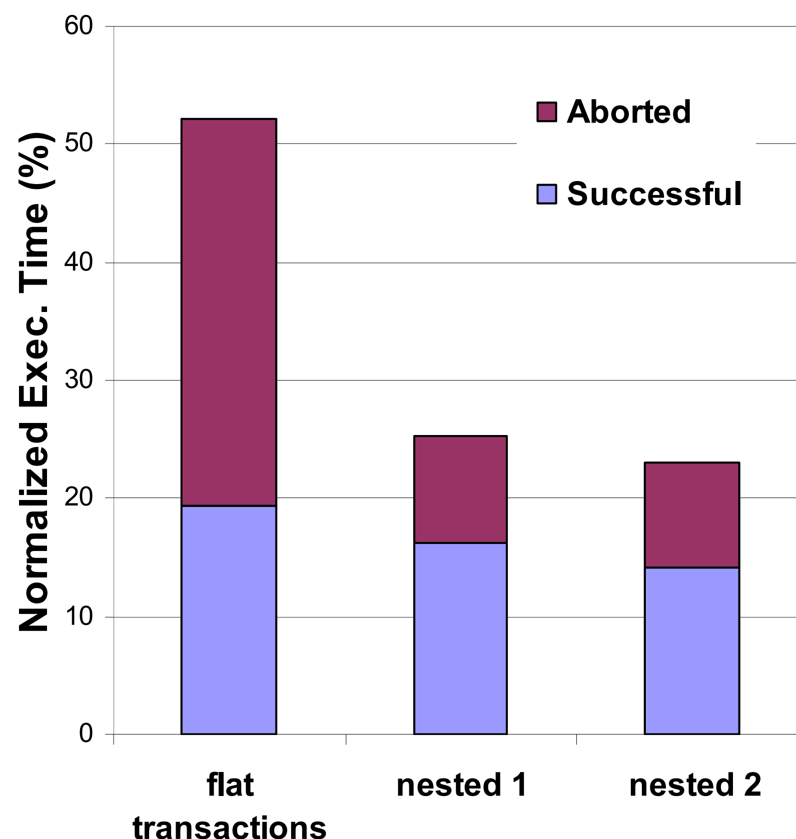
```
TransactionManager::go() {
    atomic { // begin transaction
      // 1. initialize a new order transaction
      // 2. create a new order with unique order ID
      // 3. retrieve items and stocks from warehouse
      // 4. calculate cost and update warehouse
      // 5. record the order for delivery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

© Christos Kozyrakis

HTM Performance

- Simulated 8-way CMP with TM support
  - Stanford's TCC architecture
  - Lazy versioning and optimistic conflict detection
- Speedup over sequential
  - Flat transactions: 1.9x
    - Code similar to coarse-grain locks
    - Frequent aborted transactions due to dependencies
  - Nested transactions: 3.9x to 4.2x
    - Reduced abort cost OR
    - Reduced abort frequency
- See paper in [WTW'06] for details
  - http://tcc.stanford.edu



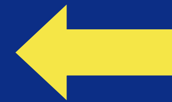
© Christos Kozyrakis

Hardware TM Summary

- High performance + compatibility with binary code
- Common characteristics
  - Data versioning in caches
  - Conflict detection through the coherence protocol
- Active research area; current research topics
  - Support for PL and OS development (see paper [ISCA'06])
    - Two-phase commit, transactional handlers, nested transactions
  - Development and comparison of various implementations
    - HTM vs STM vs Hybrid TMs
  - Long transactions & pervasive transactions
  - Scalability issues

Agenda

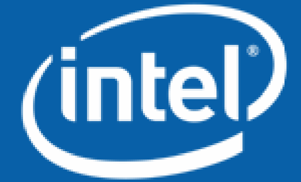
Transactional Memory (TM)

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

Q&A

© Ali-Reza Adl-Tabatabai & Bratin Saha

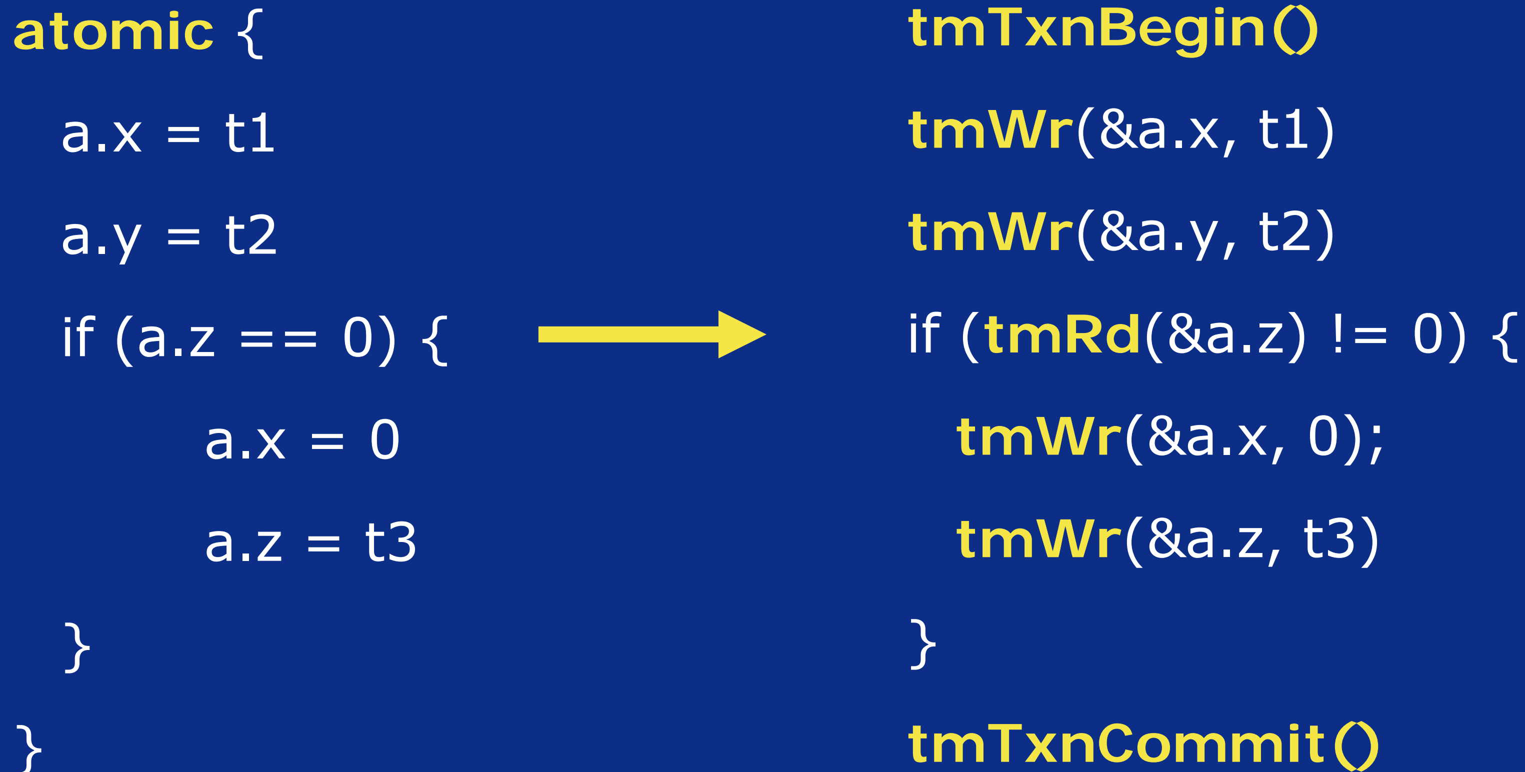
intel



Software Transactional Memory

Ali-Reza Adl-Tabatabai
Programming Systems Lab
Intel Corporation

Compiling transactions



Each shared memory access translated to a TM runtime call (a read or write barrier)

© Ali-Reza Adl-Tabatabai & Bratin Saha

TM runtime data structures

Transaction descriptor

- Read set, write set & undo log
- For conflict detection, rollback & commit

Transaction memento

- Checkpoint of transaction descriptor
- For nesting & partial rollback

Transaction record

- Pointer-sized record guarding shared data
- Tracks transactional state of data
  - Shared access by multiple readers (version number or shared reader lock)
  - Exclusive access by 1 writer (pointer to owner)
- Flexible mapping of data to transaction records

© Ali-Reza Adl-Tabatabai & Bratin Saha

intel

Mapping data to transaction records

Every data item has an associated transaction record

Java/C#

```
class Foo {
  int x;
  int y;
}
```

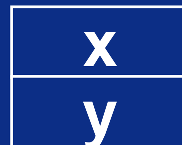
Embed in each object

Hash fields or array elements to global table

f(obj.hash, field.index)

C/C++

```
struct Foo {
  int x;
  int y;
}
```



Address-based hash into global table

Cache-line granularity

© Ali-Reza Adl-Tabatabai & Bratin Saha

intel

Conflict detection granularity

Object granularity

- Low overhead mapping operation
- Exposes optimization opportunities

Element/field granularity

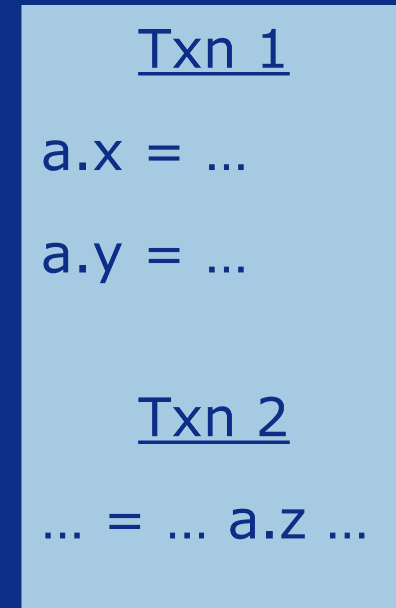
- Reduces false sharing
- Improves scalability

Cache line granularity

- Matches HW TM
- Hard for programmer & compiler to analyze

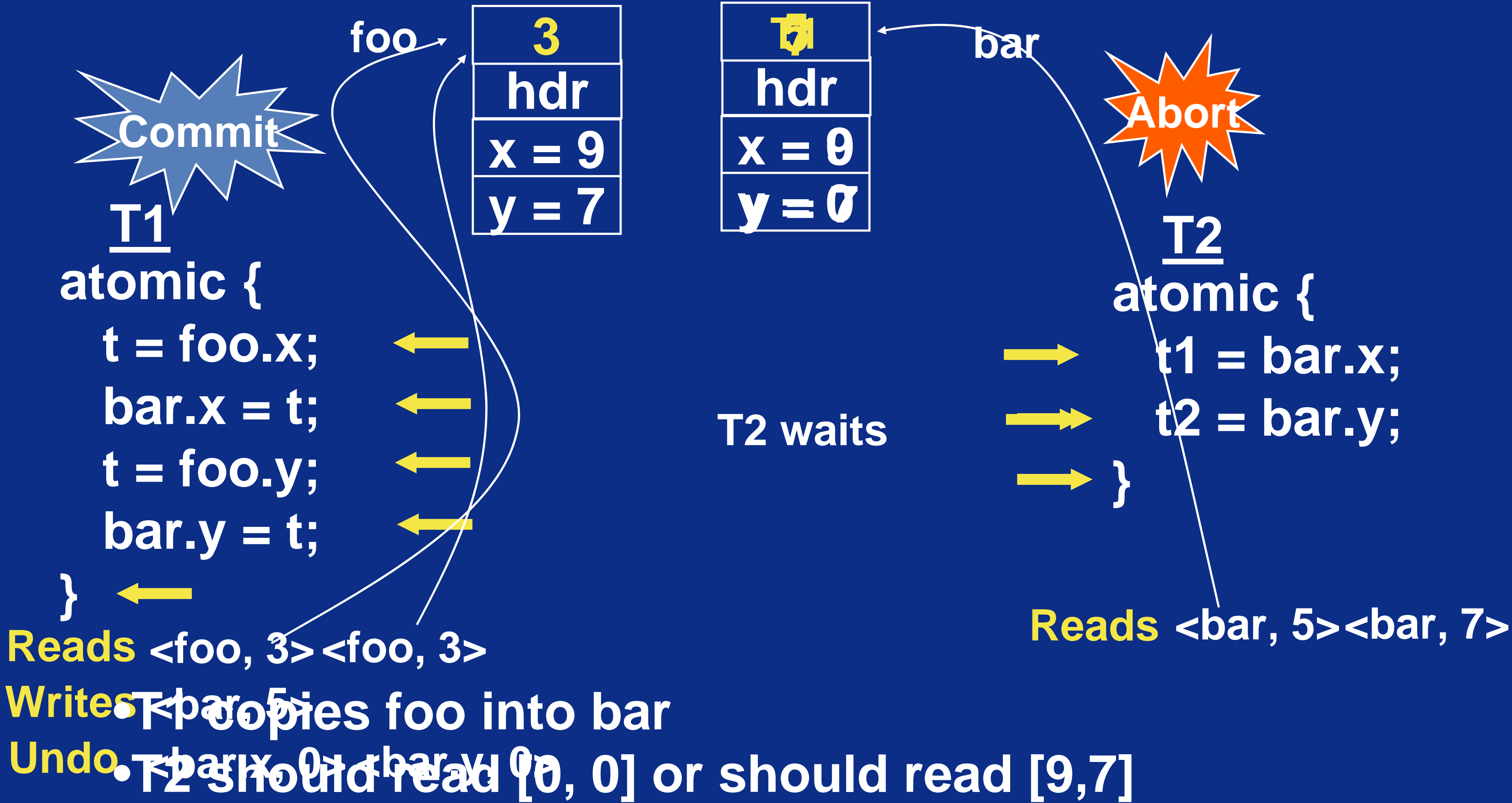
Mix & match object & element/field granularity

- Per type basis – e.g., element-level for arrays, object-level for non-arrays



© Ali-Reza Adl-Tabatabai & Bratin Saha
intel

Example: Eager versioning, optimistic STM



Transactional Memory Tutorial
© Ali-Reza Adl-Tabatabai & Bratin Saha

intel

Optimizing transactions

```
atomic {
  a.x = t1
  a.y = t2
  if (a.z == 0) {
      a.x = 0
      a.z = t3
  }
}
```

→

```
tmTxnBegin()
tmWr(&a.x, t1)
tmWr(&a.y, t2)
if (tmRd(&a.z) != 0) {
  tmWr(&a.x, 0);
  tmWr(&a.z, t3)
}
tmTxnCommit()
```

Coarse-grain barriers hide redundant logging & locking

© Ali-Reza Adl-Tabatabai & Bratin Saha

intel

Optimizing transactions

```
atomic {
  a.x = t1
  a.y = t2
  if (a.z == 0) {
      a.x = 0
      a.z = t3
  }
}
```

```
txnOpenForWrite(a)
txnLogObjectInt(&a.x, a)
a.x = t1
txnOpenForWrite(a)
txnLogObjectInt(&a.y, a)
a.y = t2
txnOpenForRead(a)
if(a.z != 0) {
  txnOpenForWrite(a)
  txnLogObjectInt(&a.x, a)
  a.x = 0
  txnOpenForWrite(a)
  txnLogObjectInt(&a.z, a)
  a.z = t3
}
```

RISC-like operations expose redundancies

© Ali-Reza Adl-Tabatabai & Bratin Saha

intel

Optimizing transactions

```
atomic {
  a.x = t1
  a.y = t2
  if (a.z == 0) {
      a.x = 0
      a.z = t3
  }
}
```

```
txnOpenForWrite(a)
txnLogObjectInt(&a.x, a)
a.x = t1
txnLogObjectInt(&a.y, a)
a.y = t2
if (a.z != 0) {
  a.x = 0
  txnLogObjectInt(&a.z, a)
  a.z = t3
}
```

Fewer & cheaper STM operations

Good compiler representation for TM is important

© Ali-Reza Adl-Tabatabai & Bratin Saha

intel

STM optimizations

- Standard compiler optimizations
  - CSE, PRE, dead-code elimination, …
  - Few modifications with proper IR for TM
  - Subtle in the presence of nesting
- STM-specific optimizations
  - Partial inlining of barrier fast paths
  - Barrier elimination
    - Immutable data (e.g., vtable, String)
    - Transaction-local data

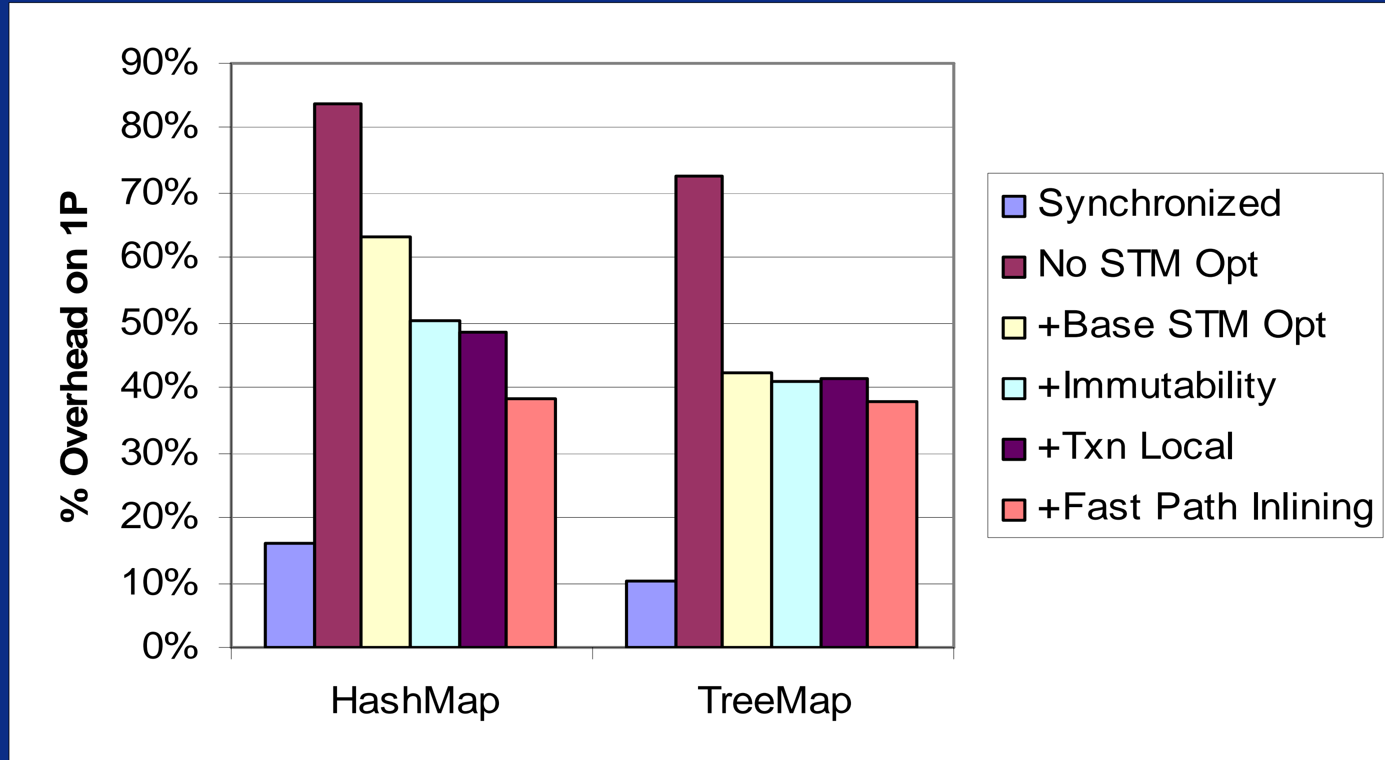
See Shpeisman et. al. PLDI 2006, Harris et. al. PLDI 2006 & Wang et. al. CGO 2007

© Ali-Reza Adl-Tabatabai & Bratin Saha

intel

Effect of compiler optimizations

1 thread overheads over thread-unsafe baseline



With compiler optimizations:

< 40% over no concurrency control

< 30% over synchronization

© Ali-Reza Adl-Tabatabai & Bratin Saha

intel

Java VM support for STM

- On-demand cloning of methods inside transactions
- Extra transaction field in each object
  - Supports mixing field & object granularity
- Garbage collection support
  - Enumeration of references in read set, write set & undo log
  - Garbage collector can move objects during a transaction
- Native methods invocation
  - Throw exception inside transaction
  - Some intrinsic functions allowed
- Runtime STM API
  - JIT compiler target
  - Wrapper around TM runtime API

See Harris & Fraser OOPSLA'03, Adl-Tabatabai et. al. PLDI 2006, and Harris et. al. PLDI 2006

© Ali-Reza Adl-Tabatabai & Bratin Saha

Transaction consistency

- In an STM with optimistic readers, a transaction may become inconsistent
  - Assuming validation done lazily
- In a managed environment, type safety and exception handling protects us
  - Validate the transaction when an exception is raised
  - Type safety ensures we don’t do wild pointer writes
- In an unmanaged environment, we can not leverage type-safety and exception handling

© Ali-Reza Adl-Tabatabai & Bratin Saha

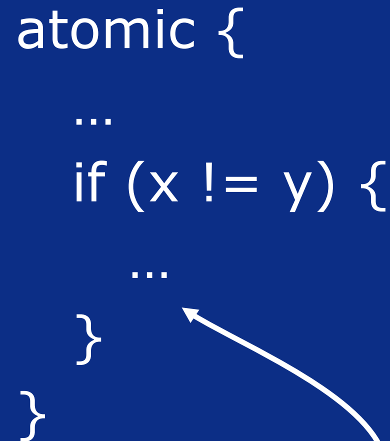
Transaction consistency

Processor 0

```
// initially x == y == 0

atomic {
   x++;
   y++;
}

// x & y are always in sync
```

Processor 1

```
atomic {
   …
   if (x != y) {
      …
   }
}
```



Execution should never get here

© Ali-Reza Adl-Tabatabai & Bratin Saha

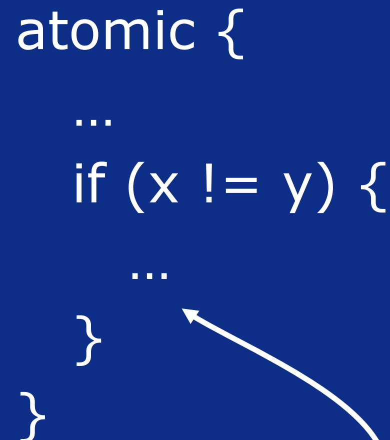
Transaction consistency

Processor 0

```
// initially x == y == 0

atomic {
   x++;
   y++;
}

// x & y are always in sync
```

Processor 1

```
atomic {
   …
   if (x != y) {
      …
   }
}
```



Lazy validation allows us to get here

© Ali-Reza Adl-Tabatabai & Bratin Saha

intel

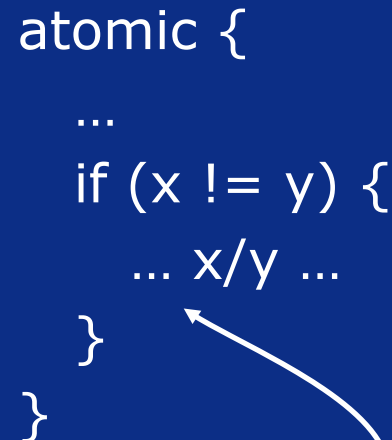
Transaction consistency

Processor 0

```
// initially x == y == 0

atomic {
   x++;
   y++;
}

// x & y are always in sync
```

Processor 1

```
atomic {
   …
   if (x != y) {
      … x/y …
   }
}
```



What if getting here raises an exception
or writes through a wild pointer?

Transactional Memory Tutorial
© Ali-Reza Adl-Tabatabai & Bratin Saha

STM in Java

Transactional Java

```
atomic {
  S;
}
```

Standard Java + STM API

```
while(true) {
  TxnHandle th = txnStart();
  try {
    S';
    break;
  } finally {
    if ( ! txnCommit(th) )
      continue;
  }
}
```

Transaction validated on uncaught exceptions
Language safety protects STM structures from corruption

© Ali-Reza Adl-Tabatabai & Bratin Saha

STM in C

- We can not rely on signal handlers in C
  - Application may override them
- An inconsistent transaction may write into STM data structures
  - Recovery becomes even more difficult
- We need to make sure that a transaction does not compute with inconsistent values
  - Get the effect of eager validation
- See Wang et. al. CGO 2007

© Ali-Reza Adl-Tabatabai & Bratin Saha

intel

Weak atomicity

Transactions isolated only from other transactions

- Most STMs behave this way
- Non-transactional accesses bypass STM access protocol

Weak atomicity exhibits subtle unintuitive behavior

- Can fail where locks work

Behavior depends on STM implementation

- See Shpeisman et. al. PLDI 2007 for taxonomy of issues

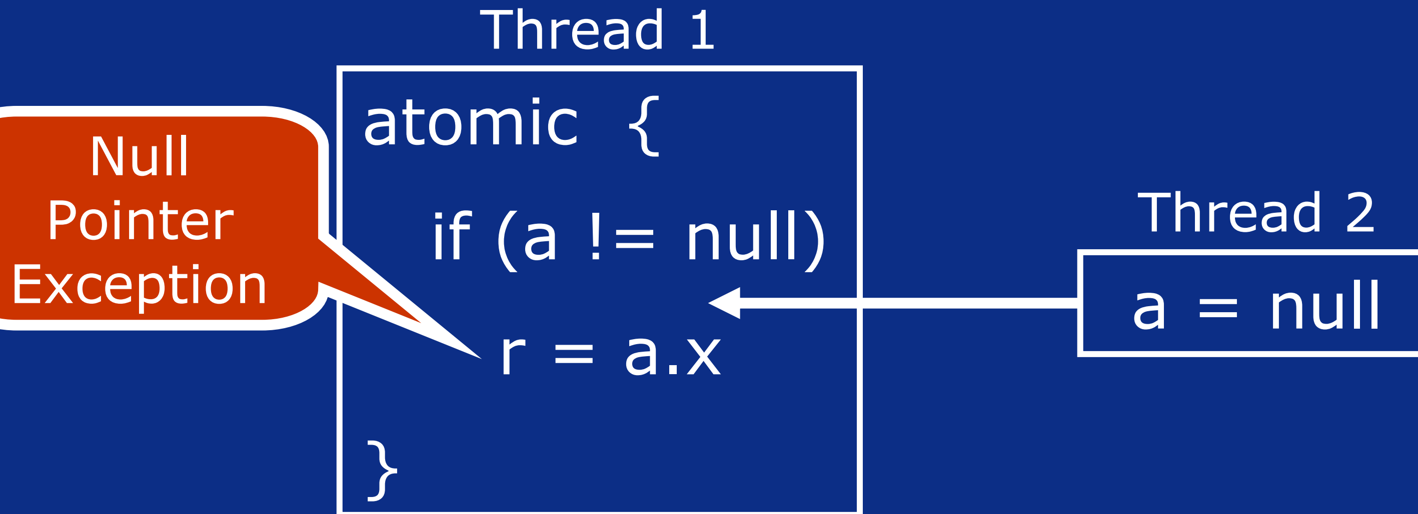
Requires segregation of transactional and non-transactional data

- Hard to enforce and error-prone
- Violations lead to subtle bugs

© Ali-Reza Adl-Tabatabai & Bratin Saha

intel

Unintuitive behavior



Locks behave the same way

Things get worse ...

© Ali-Reza Adl-Tabatabai & Bratin Saha

Privatization example: Locks

[Larus & Rajwar 2006, Hudson et. al. ISMM 2006]

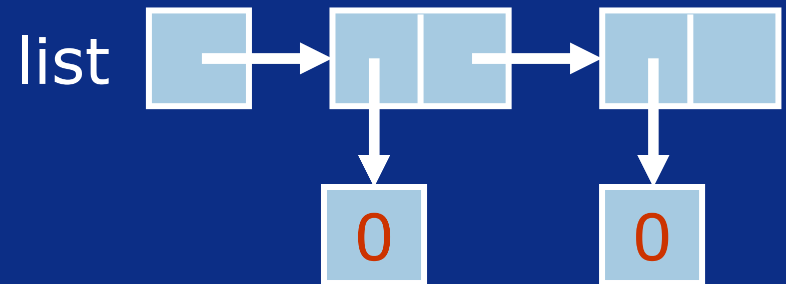
Thread 1

```
synchronized (list) {
 e = list.removeFirst();
}
r1 = e.x;
r2 = e.x;
Can r1 != r2?
```

No

Thread 2

```
synchronized (list) {
  if (!list.isEmpty()) {
      e = list.getFirst();
      e.x = 1;
}
```



This program is correctly synchronized

© Ali-Reza Adl-Tabatabai & Bratin Saha

intel

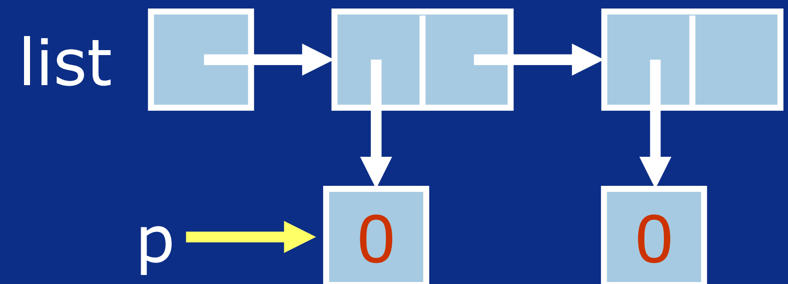
Privatization example: STM

Thread 1

```
atomic {
 e = list.removeFirst();
}
r1 = e.x;
r2 = e.x;
Can r1 != r2?
```

Thread 2

```
atomic {
  if (!list.isEmpty()) {
    p = list.getFirst();
    p.x = 1;
}
```



© Ali-Reza Adl-Tabatabai & Bratin Saha

intel

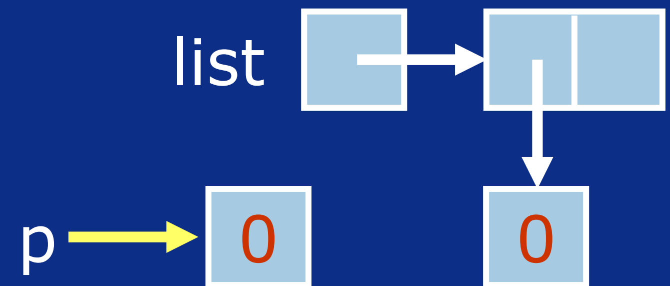
Privatization example: STM

Thread 1

```
atomic {
 e = list.removeFirst();
}
r1 = e.x;
r2 = e.x;
Can r1 != r2?
```

Thread 2

```
atomic {
  if (!list.isEmpty()) {
    p = list.getFirst();
    p.x = 1;
}
```



© Ali-Reza Adl-Tabatabai & Bratin Saha

Privatization example: STM

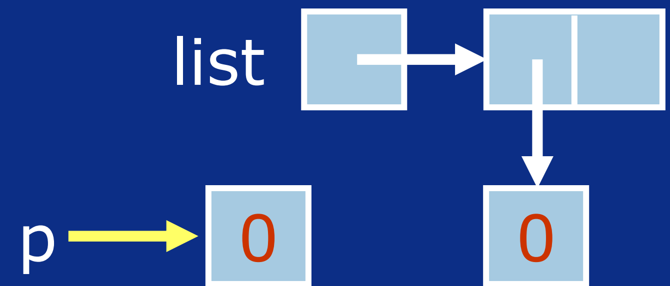
Thread 1

```
atomic {
 e = list.removeFirst();
}
r1 = e.x;
r2 = e.x;
Can r1 != r2?
```

Commit

Thread 2

```
atomic {
  if (!list.isEmpty()) {
    p = list.getFirst();
    p.x = 1;
}
```



© Ali-Reza Adl-Tabatabai & Bratin Saha

Privatization example: STM

Thread 1

```
atomic {
  e =
  list.remov   ();
}
r1 = e.x;
r2 = e.x;
Can r1 != r2?
```

Commit

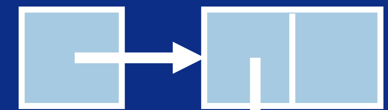
r1 = 1

Thread 2

```
atomic {
  if (!list.isEmpty()) {
     p = list.getFirst();
     p.x = 1;
}
```

Abort

list



p

1

0

© Ali-Reza Adl-Tabatabai & Bratin Saha

intel

Privatization example: STM

Thread 1

```
atomic {
  e = list.removeFirst();
}
r1 = e.x;
r2 = e.x;
Can r1 != r2?
```

r1 = 1

r2 = 0

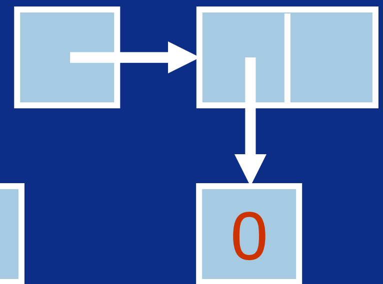
Yes!

Lazy-versioning? Yes!

Thread 2

```
atomic {
  if (!list.isEmpty()) {
      p = list.getFirst();
      p.x = 1;
}
```

Abort



© Ali-Reza Adl-Tabatabai & Bratin Saha

intel

Privatization

Use a commit time fence to avoid privatization problems

See Wang et. al. CGO 2007

- Solves both privatization and consistency issues

© Ali-Reza Adl-Tabatabai & Bratin Saha

intel

Strong atomicity

Isolates transactions from all memory accesses
- Aka strong isolation

Requires barriers for non-transactional accesses
- Affects performance of non-transactional code

JIT optimizations
- Barrier elimination for immutable & thread-local data
- Combine multiple accesses to same object into one transaction

Dynamic escape analysis
- Tracks thread-local objects at runtime
- Objects reachable from static fields are visible to multiple threads
- Shortens barrier fast path

© Ali-Reza Adl-Tabatabai & Bratin Saha

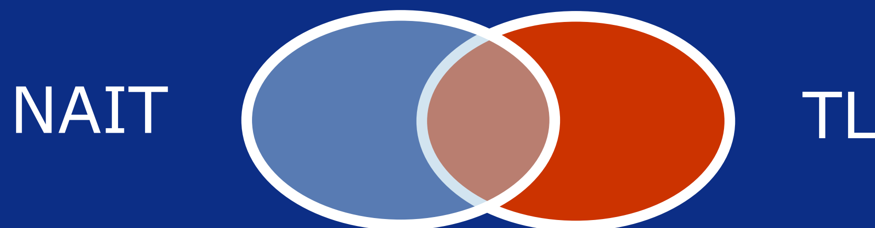
Whole-program optimizations for strong atomicity

Not-Accessed-In-Transaction Analysis (NAIT)

- Objects not accessed in transactions do not require barriers
- Object not written in transactions do not require read barriers

Thread-Local Analysis (TL)

- Classical optimization
- Eliminates barriers for thread-local objects



© Ali-Reza Adl-Tabatabai & Bratin Saha

Research challenges

Performance

- Compiler optimizations
- Right mix of hardware & software components
- Dealing with contention

Semantics

- Memory model
- Nested parallelism
- Integration with locks

System integration

- I/O
- Transactional OS
- Distributed transactions

Debugging & performance analysis tools

- Good diagnostics

Leveraging the TM infrastructure

- Speculative optimizations & multithreading

© Ali-Reza Adl-Tabatabai & Bratin Saha

intel

Conclusions

- Multi-core architectures: an inflection point in mainstream SW development
- Navigating inflection requires new parallel programming abstractions
- Transactions are a better synchronization abstraction than locks
  - Software engineering and performance benefits
- Lots of research on implementation and semantics issues
  - Great progress, but there are still open problems

© Ali-Reza Adl-Tabatabai & Bratin Saha

C STM now available

- Intel’s research STM for C now available for evaluation
- Based on Wang et. al. CGO 2007
- Supports C with transaction language extension
- Download binaries from whatif.intel.com
- Give us feedback

© Ali-Reza Adl-Tabatabai & Bratin Saha

Questions?

© Ali-Reza Adl-Tabatabai & Bratin Saha