Programming with Transactional Memory

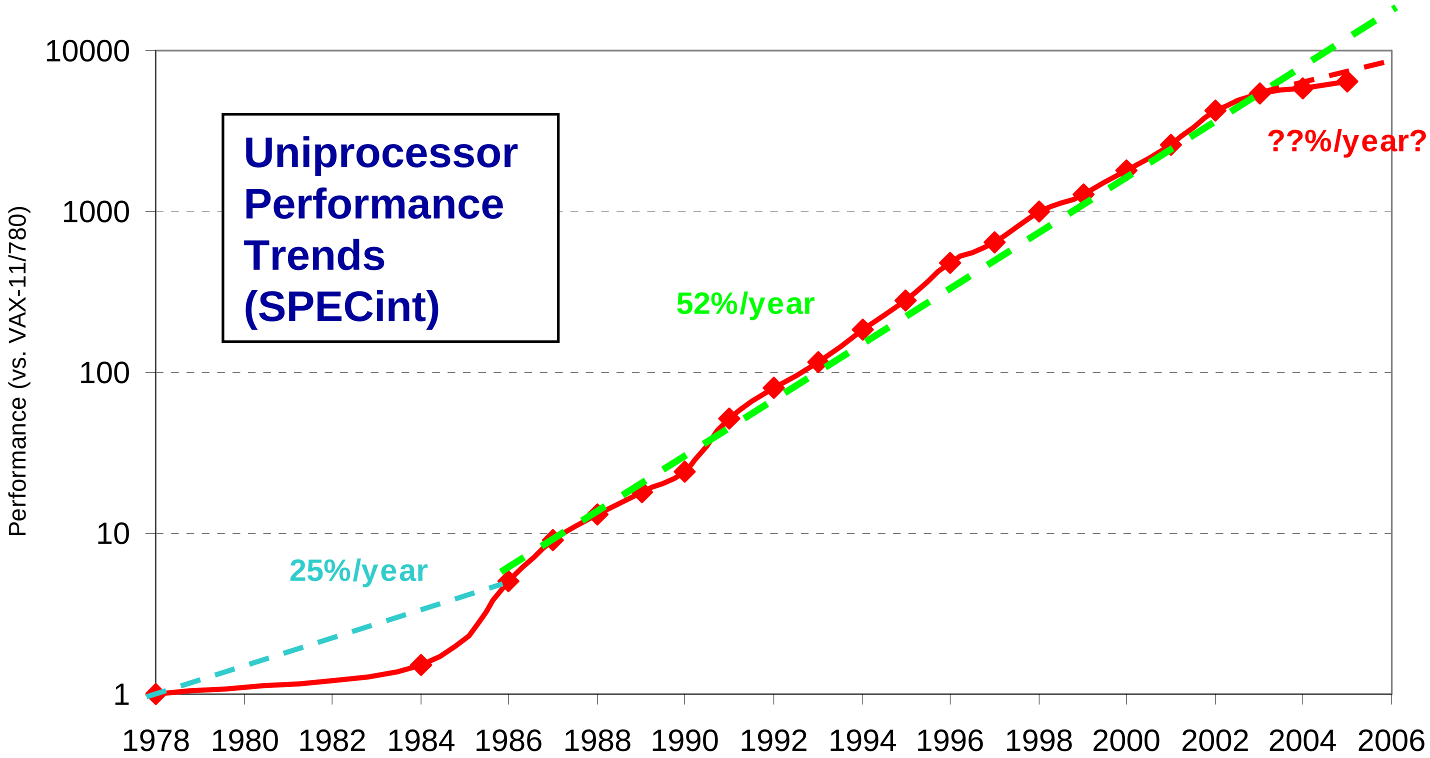
Brian D. Carlstrom

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

The Problem: “The free lunch is over”

Chip manufacturers have switched from making faster uniprocessors to adding more processor cores per chip

- Software developers can no longer just hope that the next generation of processor will make their program faster



From Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 4th edition, Sept. 15, 2006

Parallel Programming for the Masses?

Every programmer is now a parallel programmer

- The black arts now need to be taught to undergraduates

- IBM and Sun went multi-core first on the server side
- AMD/Intel now in core count race for laptops, desktops, and servers

Year	Microprocessor	Proc/chip	Thread/proc	Thread/chip
2004	IBM POWER5	2	2	4
2005	Azul Vega 1	24	1	24
2005	Sun Niagara 1	8	4	32
2005	AMD Opteron	2	1	2
2006	Intel Woodcrest	2	2	4
2006	Intel Barcelona	4	1	4
2006	Azul Vega 2	48	1	48
2007	AMD Barcelona	4	1	4
2007	Sun Niagara 2	8	8	64
2008	*Intel*	*4*	*2*	*8*
2009	*AMD*	*8*	*1*	*8*
2009	*Intel*	*8*	*2*	*16*

What Makes Parallel Programming Hard?

Typical parallel program

- Single memory shared by multiple program threads
- Need to coordinate access to memory shared b/w threads
- Locks allow temporary exclusive access to shared data

Lock granularity tradeoff

- Coarse grained locks - contention, lack of scaling, …
- Fine grained locks - excessive overhead, deadlock,…

Apparent tradeoff between correctness and performance

- Easier to reason about only a few locks…
- … but only a few locks can lead to contention

Transactional Memory to the Rescue?

Transactional Memory

- Replaces waiting for locks with concurrency
- Allows non-conflicting updates to shared data
- Shown to improve scalability of short critical regions

Promise of Transactional Memory

- Program with coarse transactions
- Performance like fine grained lock

Focus on correctness, tune for performance

- Easier to reason about only a few transactions…
- … only focus on areas with true contention

Thesis and Contributions

Thesis:

If transactional memory is to *make parallel programming easier*, rather than just more scalable, the programming interface requires *more than simple atomic transactions*

To support this thesis I will:

- Show why lock based programs cannot be simply translated to a transactional memory model
- Present the design of Atomos, a parallel programming language designed for transactional memory
- Show how Atomos can support semantic concurrency control, allowing programs with coarse transactions to perform competitively with fine-grained transactions.

Overview

- Motivation and Thesis
  - How to make parallel programming of chip multiprocessors easier using transactional memory
- Transactional Memory
  - Concepts, implementation, environment
- JavaT [SCP 2006]
  - Executing Java programs with Transactional Memory
- Atomos [PLDI 2006]
  - A transactional programming language
- Semantic concurrency control [PPoPP 2007]
  - Improving scalability of applications with long transactions

Locks versus Transactions

Lock

```
...
synchronized (lock) {
  x = x + y;
}
...
```

Transaction

```
...
atomic {
  x = x + y;
}
...
```

Mapping from lock to protected data

- **`lock`** protects **`x`**

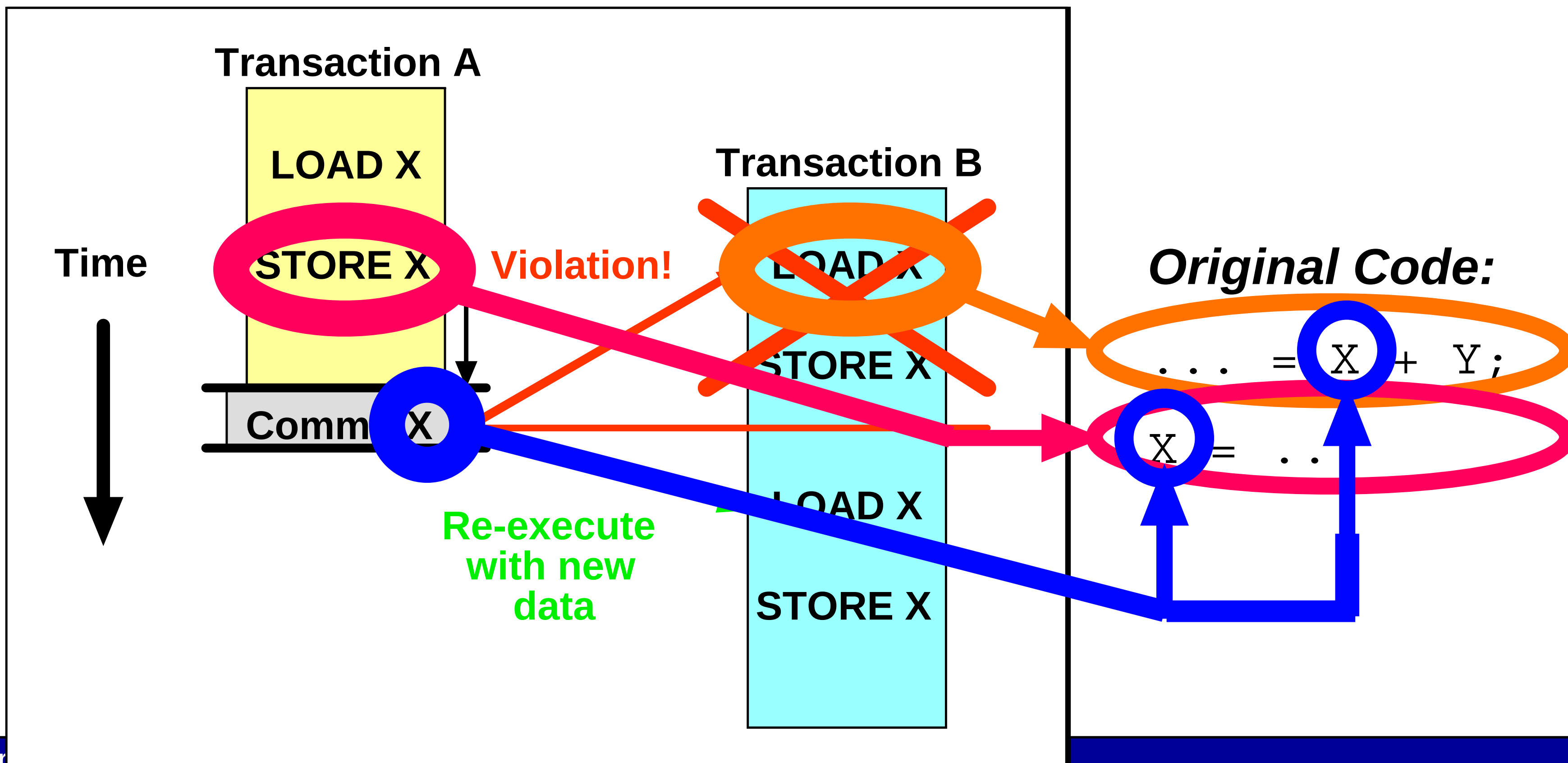
Transaction protects all data

- No need to worry if another lock is necessary to protect **`y`**

Transactional Memory at Runtime

What if transactions modify the same data?

- First commit causes other transactions to abort & restart
- Can provide programmer with useful feedback!



Transactional Memory Related Work

Transactional Memory

- Transactional Memory: Architectural Support for Lock-Free Data Structures [Herlihy & Moss 1993]
- Software Transactional Memory [Shavit & Touitou 1995]

Database

- Transaction Processing [Gray & Reuter 1993]

4.7)	Nested transactions	[Moss	1981]
4.9)	Multi-level transactions	[Weikum & Schek	1984]
4.10)	Open nesting	[Gray	1981]
16.7.3)	Commit and abort handlers	[Eppinger et al.	1991]

Recent Transactional Memory

- Language support for lightweight txs [Harris & Fraser 2003]
- Exceptions and side-effects in atomic blocks [Harris 2004]
- Open nesting in STM [Ni et al. 2007]

Hardware Environment

Chip Multiprocessor

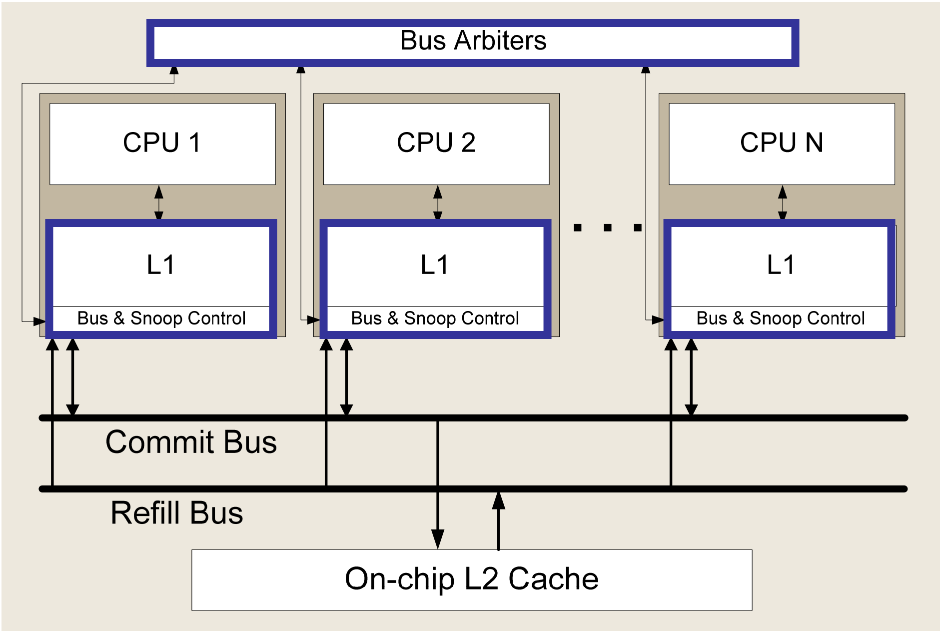
- up to 32 CPUs
- write-back L1
- shared L2
- x86 ISA

Lock evaluation

- MESI protocol

TM evaluation

- L1 buffers speculative data
- Bus snooping detects data dependency violations



Changes for TM support

Software Environment

Virtual Machine

- IBM's Jikes RVM (Research Virtual Machine) 2.4.2+CVS
- GNU Classpath 0.19

HTM extensions

- VM_Magic methods converted by JIT to HTM primitives

Polyglot

- Translate language extensions to VM_Magic calls

Overview

Motivation and Thesis

- How to make parallel programming of chip multiprocessors easier using transactional memory

Transactional Memory

- Concepts, implementation, environment

JavaT [SCP 2006]

- Executing Java programs with Transactional Memory

Atomos [PLDI 2006]

- A transactional programming language

Semantic concurrency control [PPoPP 2007]

- Improving scalability of applications with long transactions

JavaT: Transactional Execution of Java Programs

Goals

- Run existing Java programs using transactional memory
- Require no new language constructs
- Require minimal changes to program source
- Compare performance of locks and transactions

Non-Goals

- Create a new programming language
- Add new transactional extensions
- Run all Java programs correctly without modification

JavaT: Rules for Translating Java to TM

Three rules create transactions in Java programs

1. **`synchronized`** defines a transaction
2. **`volatile`** references define transactions
3. **`Object.wait`** performs a transaction commit

Allows supports execution of a variety of programs:

- Histogram based on our ASPLOS 2004 paper
- STM benchmarks from Harris & Fraser, OOPSLA 2003
- SPECjbb2000 benchmark
- All of Java Grande (5 kernels and 3 applications)

Performance comparable or better in almost all cases

Many developers already believe that `synchronized` means atomic, as opposed to mutual exclusion!

JavaT: Defining transactions with synchronized

`synchronized` blocks define transactions

```
public static void main (String args[]){
    a();
    synchronized (x){
        b();
    }
    c();
}
```

→

```
a();      // non-transactional
BeginNestedTX();
b();      //     transactional
EndNestedTX();
c();      // non-transactional
```

We use closed nesting for nested **`synchronized`** blocks

```
public static void main (String args[]){
    a();
    synchronized (x){
        b1();
        synchronized (y) {
            b2();
        }
        b3();
    }
    c();
}
```

→

```
a();      // non-transactional
BeginNestedTX();
b1();     // transaction at level 1
BeginNestedTX();
b2();     // transaction at level 2
EndNestedTX();
b3();     // transaction at level 1
EndNestedTX();
c();      // non-transactional
```

JavaT: Alternative to rollback on wait

JavaT rules say that **`Object.wait`** commits transaction

- Other proposals rollback on wait (or prohibit side effects)
  - C.A.R. Hoare's Conditional Critical Regions (CCRs)
  - Harris's **`retry`** keyword
  - Welc et al.'s Transactional Monitors

Rollback handles one common pattern of condition variables

```
sychronized (lock) {
    while (!condition)
        wait();
    ...
}
```

JavaT: Commiting on wait

- So why does JavaT commit on wait?
- Motivating example: A simple barrier implementation

```
synchronized (lock) {
    count++;
    if (count != thread_count) {
        lock.wait();
    } else {
        count = 0;
        lock.notifyAll();
    }
}
```

Code like this is found in Sun Java Tutorial, SPECjbb2000, and Java Grande

- With commit, barrier works as intended
- With rollback, all threads think they are first to barrier

JavaT: Commit on wait tradeoff

Major positive of commit on wait

- Allows transactional execution of existing Java code

Major negative of commit on wait

- Nested transaction problem
- We don’t want to commit value of “a” when we wait:

```
synchronized (x) {
    a = true;
    synchronized (y) {
        while (!b)
            y.wait();
        c = true;}}
```

- With locks, wait releases specific lock
- With transactions, wait commits all outstanding transactions
- In practice, nesting examples are very rare
  - It is bad to wait while holding a lock
  - wait and notify are usually used for unnested top level coordination

JavaT: Keeping Scalable Code Simple

TestCompound benchmark from Harris & Fraser, OOPSLA 2003

- Atomic swap of Map elements

Java HashMap,
Java Hashtable,
ConcurrentHashMap

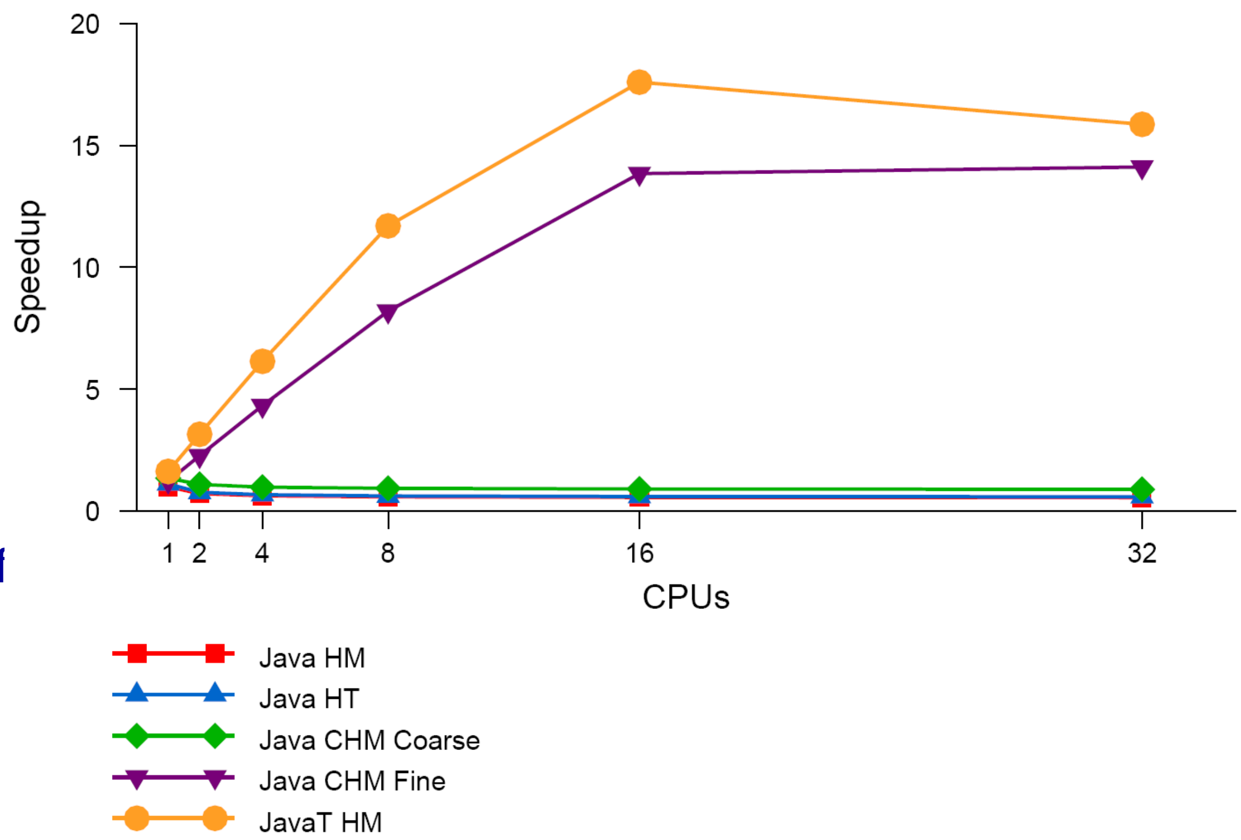
- Simple lock around swap does not scale

ConcurrentHM Fine

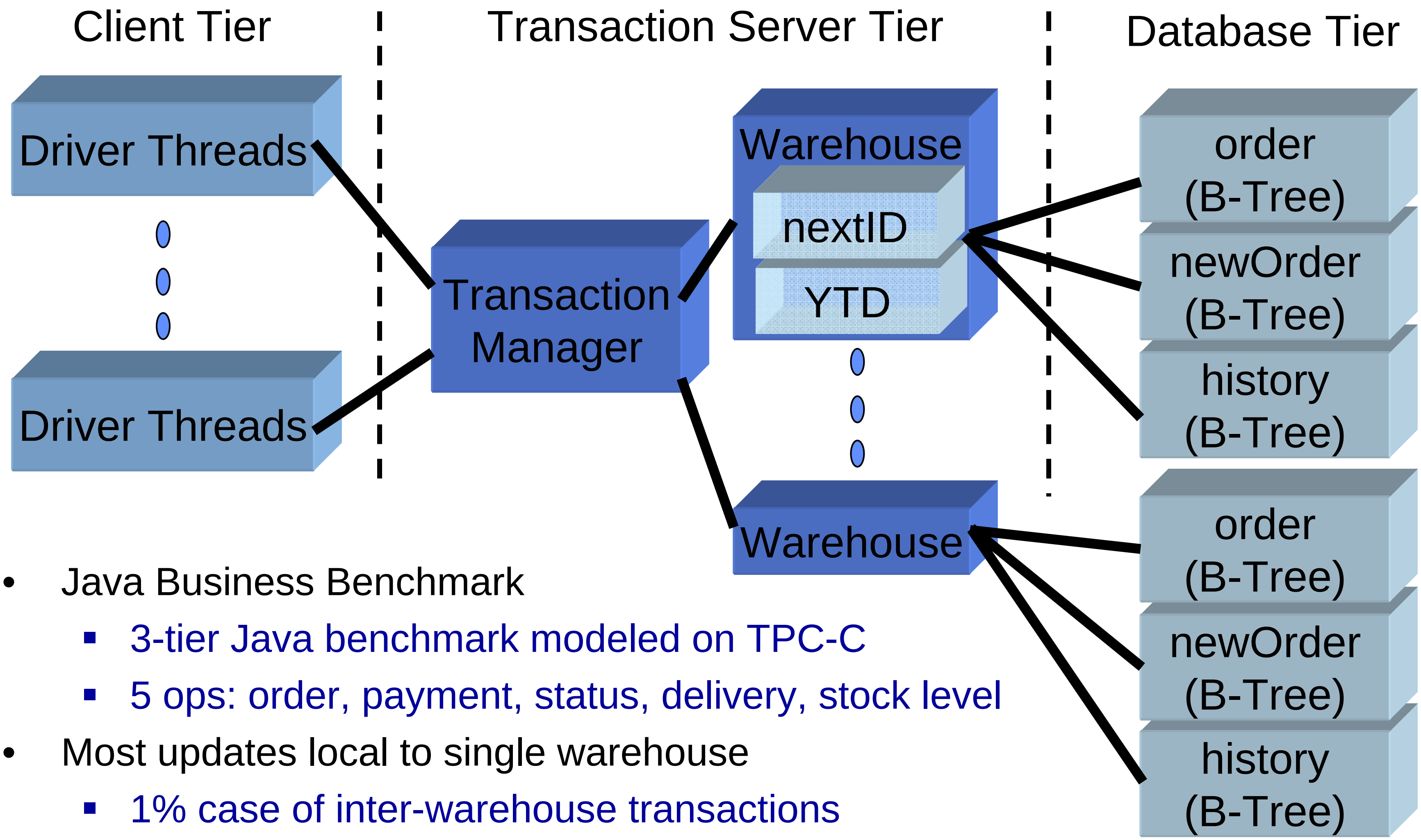
- Use ordered key locks to avoid deadlock

JavaT HashMap

- Use simplest code of Java HM, performs best of all!



SPECjbb2000 Overview

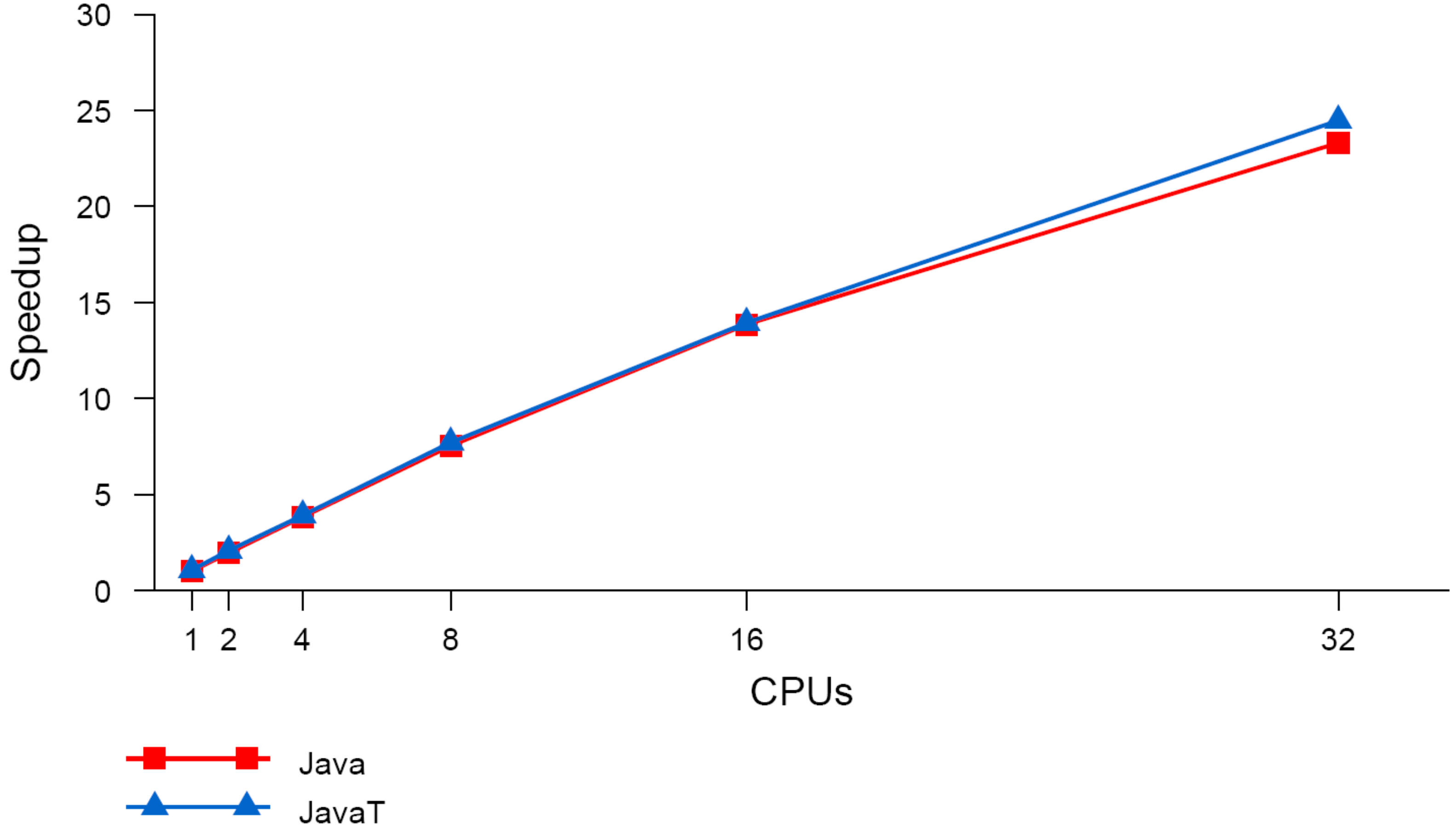


- Java Business Benchmark
  - 3-tier Java benchmark modeled on TPC-C
  - 5 ops: order, payment, status, delivery, stock level
- Most updates local to single warehouse
  - 1% case of inter-warehouse transactions

JavaT: SPECjbb2000 Results

SPECjbb2000

- Close to linear scaling for transactions and locks up to 32 CPUs
  - 32 CPU scale limited by bus in simulated CMP configuration



JavaT: Transactional Execution of Java Programs

Goals (revisited)

- Run existing Java programs using transactional memory
  - Can run a wide variety of existing benchmarks
- Require no new language constructs
  - Used existing **`synchronized`**, **`volatile,`** and **`Object.wait`**
- Require minimal changes to program source
  - No changes required for these programs
- Compare performance of locks and transactions
  - Generally better performance from transactions

Problem

- Conditional waiting semantics not right for all programs
- What can we do if we can change the language?

Overview

Motivation and Thesis

- How to make parallel programming of chip multiprocessors easier using transactional memory

Transactional Memory

- Concepts, implementation, environment

JavaT [SCP 2006]

- Executing Java programs with Transactional Memory

Atomos [PLDI 2006]

- A transactional programming language

Semantic concurrency control [PPoPP 2007]

- Improving scalability of applications with long transactions

The Atomos Programming Language

Atomos derived from Java

- `atomic` replaces `synchronized`
- `retry` replaces `wait/notify/notifyAll`

Atomos design features

- Open nested transactions
  - `open` blocks committing nested child transaction before parent
  - Useful for language implementation but also available for applications
- Commit and Abort handlers
  - Allow code to run dependant on transaction outcome
- Watch Sets
  - Extension to `retry` for efficient conditional waiting on HTM systems

Atomos: The counter problem

Application

```
atomic {
  ...
  id = nextId();
  ...
}
static long nextId() {
  atomic {
    nextID++;
}}
```

JIT Compiler

```
// method prolog
...
invocationCounter++;
...
// method body
...
// method epilogue
...
```

- Lower-level updates to global data can lead to violations
- General problem not confined to counters:
  - Application level caching
  - Cooperative scheduling in virtual machine

Atomos: Open nested counter solution

Solution

- Wrap counter update in open nested transaction

```
atomic {
  ...
  id = nextId();
  ...
}

static long nextID () {
  open {
    nextID++;
  }
}
```

Benefits

- Violation of counter just replays open nested transaction
- Open nested commit discards child's read-set preventing later violations

Issues

- What happens if parent rolls back after child commits?
- Okay for *statistical* counters and UID
- Not okay for SPECjbb2000 YTD (year-to-date) payment counters
  - Need to some way to coordinate with parent transaction

Atomos: Commit and Abort Handlers

Programs can specify callbacks at end of transaction

- Separate interfaces for commit and abort outcomes

```
public interface CommitHandler { boolean onCommit();}
public interface AbortHandler  { boolean onAbort ();}
```

Historical uses for commit and abort handlers

- DB technique for delaying non-transactional operations
- Harris brought the technique to STM for solving I/O problem
  - See *Exceptions and side-effects in atomic blocks*.
  - Buffer output until commit, rewind input on abort

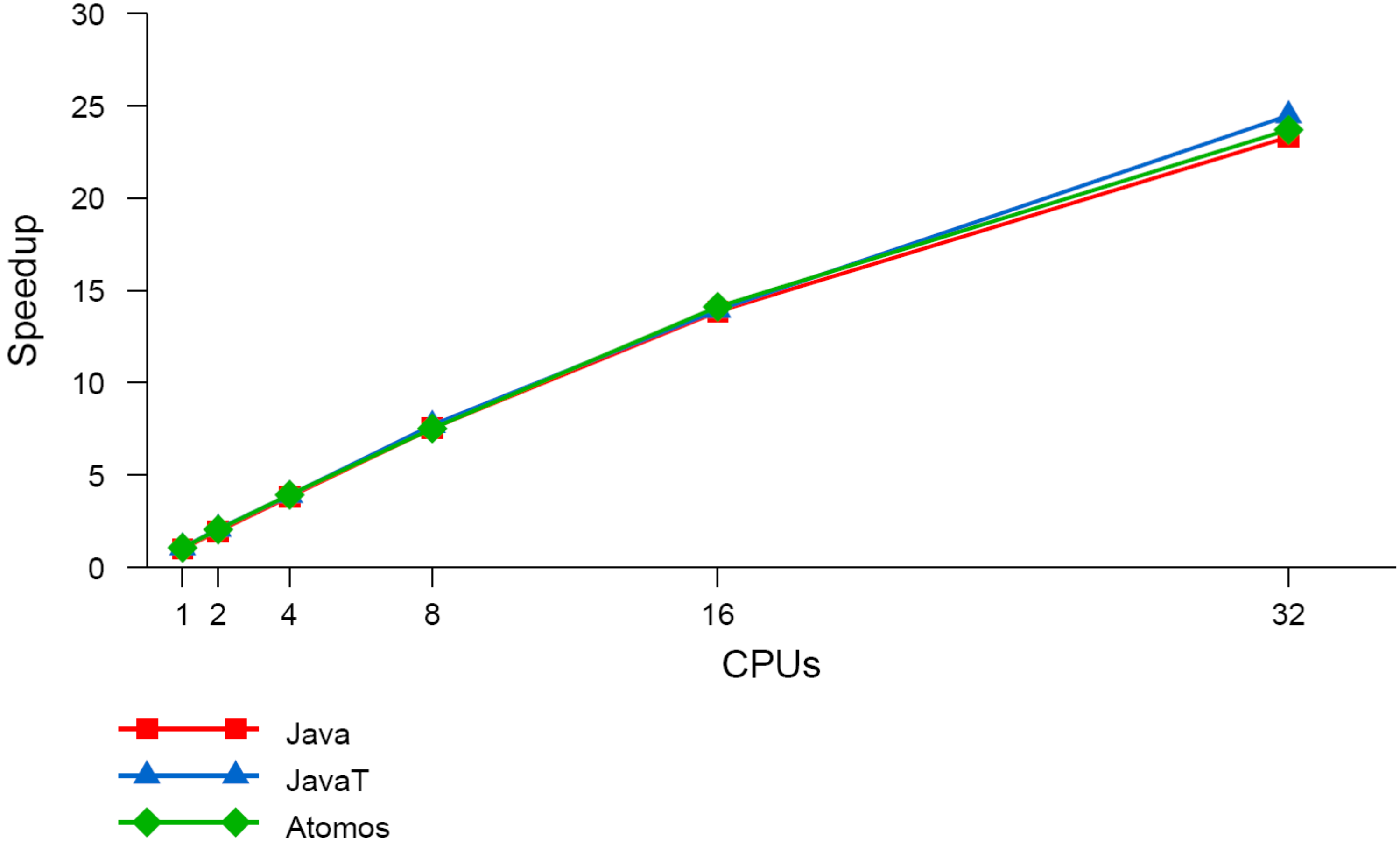
Atomos applications

- EITHER Delay updates to shared data until parent commits
  - Update YTD field only when parent is committing
- OR Provide compensation action to open nesting
  - Undo YTD update when parent is aborted

Atomos: SPECjbb2000 Results

SPECjbb2000

- Difference between JavaT and Atomos result is handler overhead
- Overhead has negligible impact, Atomos still outperforms Java



Atomos Summary

Atomos similarities to other proposals

- `atomic`, `retry`, and commit/abort handlers

Atomos differences

- Open nested transactions for reduced isolation
- `watch` allows for scalable HTM `retry` implementation

Open nested transactions controversial

- Some uses straight forward
- More sophisticated uses require proper handlers

Can we give programmers the benefits of open nesting without expecting them to use it directly?

Overview

Motivation and Thesis

- How to make parallel programming of chip multiprocessors easier using transactional memory

Transactional Memory

- Concepts, implementation, environment

JavaT [SCP 2006]

- Executing Java programs with Transactional Memory

Atomos [PLDI 2006]

- A transactional programming language

Semantic concurrency control [PPoPP 2007]

- Improving scalability of applications with long transactions

What happens to SPECjbb with long transactions?

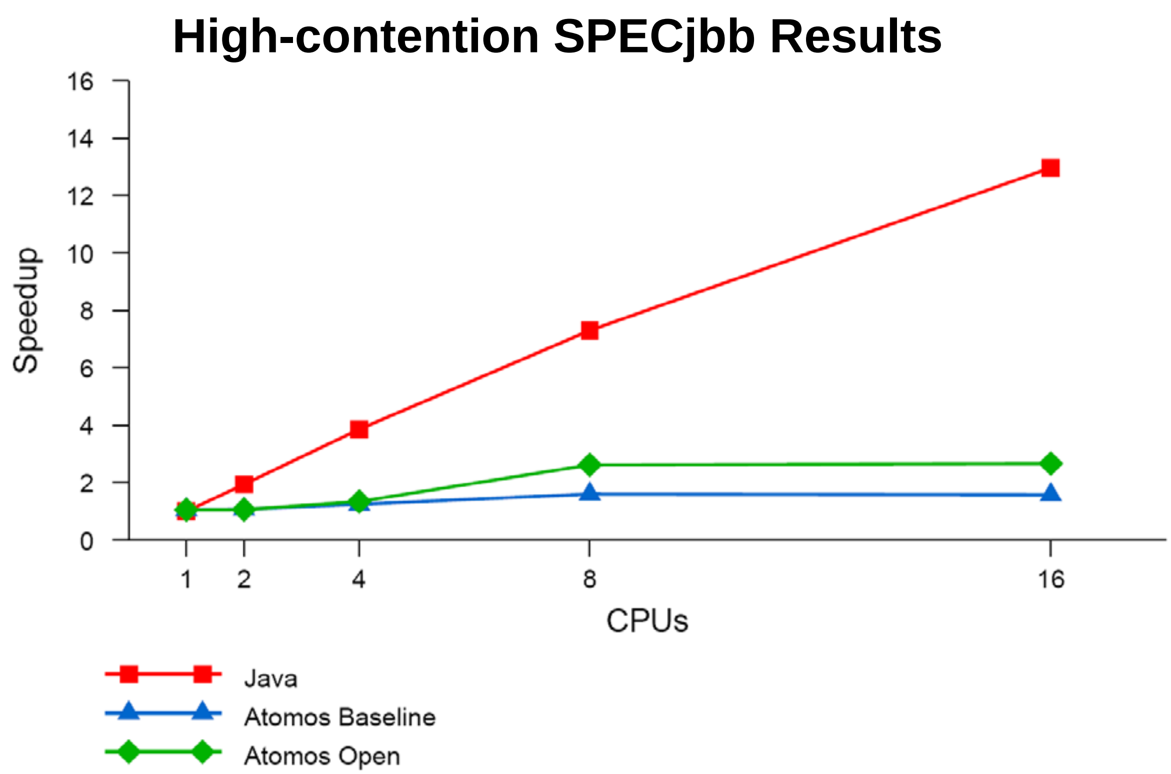
Old: SPECjbb could scale

- Open nesting addresses counters
- Only 1% of operations touch other warehouse data structures

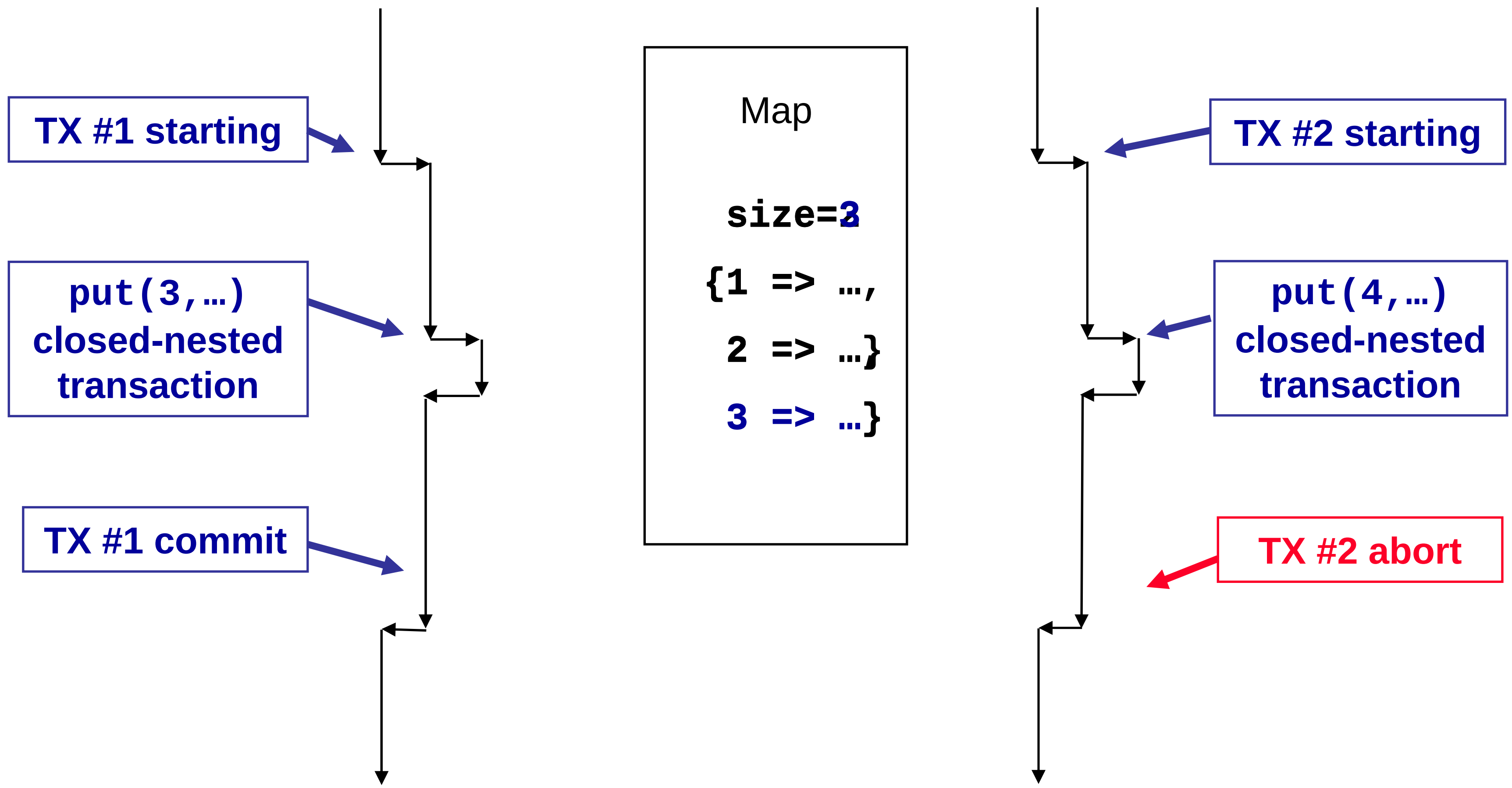
New: high-contention SPECjbb

- All threads in 1 warehouse
- All transactions touch some shared Map

Open nested results not much better than Baseline



Violations in logically independent operations



Unwanted data dependencies limit scaling

Data structure bookkeeping causing serialization

- Frequent HashMap and TreeMap violations updating size and modification counts

With short transactions

- Enough parallelism from operations that do not conflict to make up for the ones that do conflict

With long transactions

- Too much lost work from conflicting operations

How can we eliminate unwanted dependencies?

Reducing unwanted dependencies

Custom hash table

- Don't need size or modCount? Build stripped down Map
- Disadvantage: Do not want to custom build data structures

Open-nested transactions

- Allows a child transaction to commit before parent
- Disadvantage: Lose transactional atomicity

Segmented hash tables

- Use ConcurrentHashMap (or similar approaches)
    - Compiler and Runtime Support for Efficient STM, Intel, PLDI 2006
- Disadvantage:
  Reduces, but does not eliminate, unnecessary violations

Is this reduction of violations good enough?

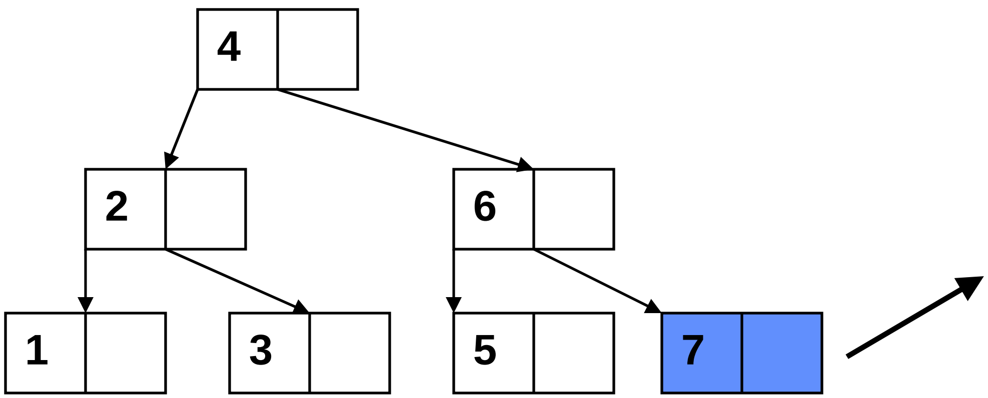
Semantic Concurrency Control

Database concept of multi-level transactions

- Release low-level locks on data after acquiring higher-level locks on semantic concepts such as keys and size

Example

- Before releasing lock on B-tree node containing key 7 record dependency on key 7 in lock table
- B-tree locks prevent races – lock table provides isolation



TX#	Key	Mode
…	…	…
#2317	7	Read
…	…	…

Semantic Concurrency Control

Applying Semantic Concurrency Control to TM

- Avoid retaining memory level dependencies
- Replace with semantic dependencies
- Add conflict detection on semantic properties

Transactional Collection Classes

- Avoid memory level dependencies on size field, …
- Replace with semantic dependencies on keys, size, …
- Only detect semantic conflicts that are necessary

No more memory conflicts on implementation details

Benefits of Transactional Collection Classes

Programmer just uses the usual collection interfaces

- Code change as simple as replacing

```
Map map = new HashMap();
```

- with

```
Map map = new TransactionalMap();
```

Similar interface coverage to util.concurrent

- Maps: TransactionalMap, TransactionalSortedMap
- Sets: TransactionalSet, TransactionalSortedSet
- Queue: TransactionalQueue

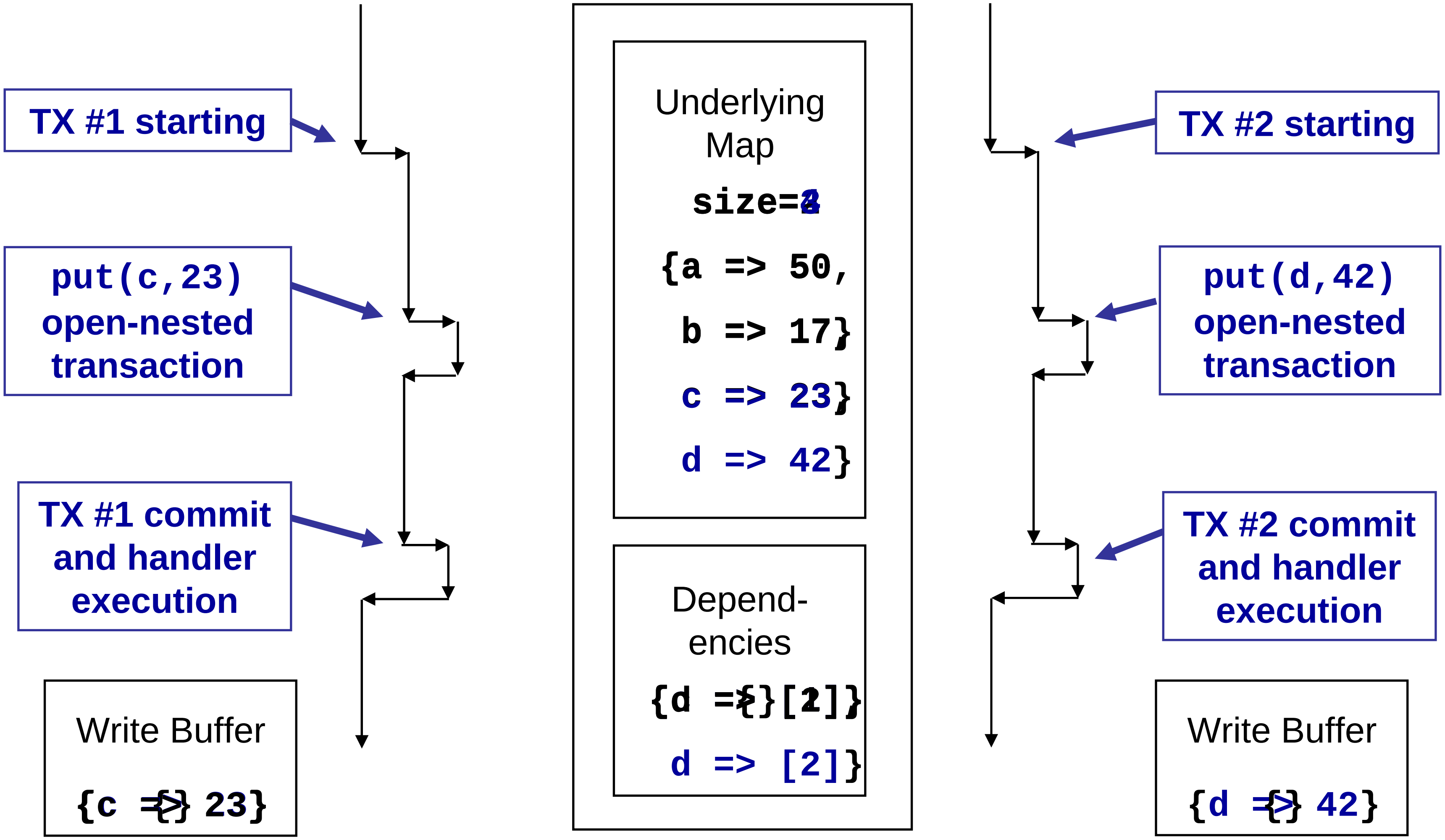
Only library writers deal directly with open nesting

- Similar to `java.util.concurrent.atomic`

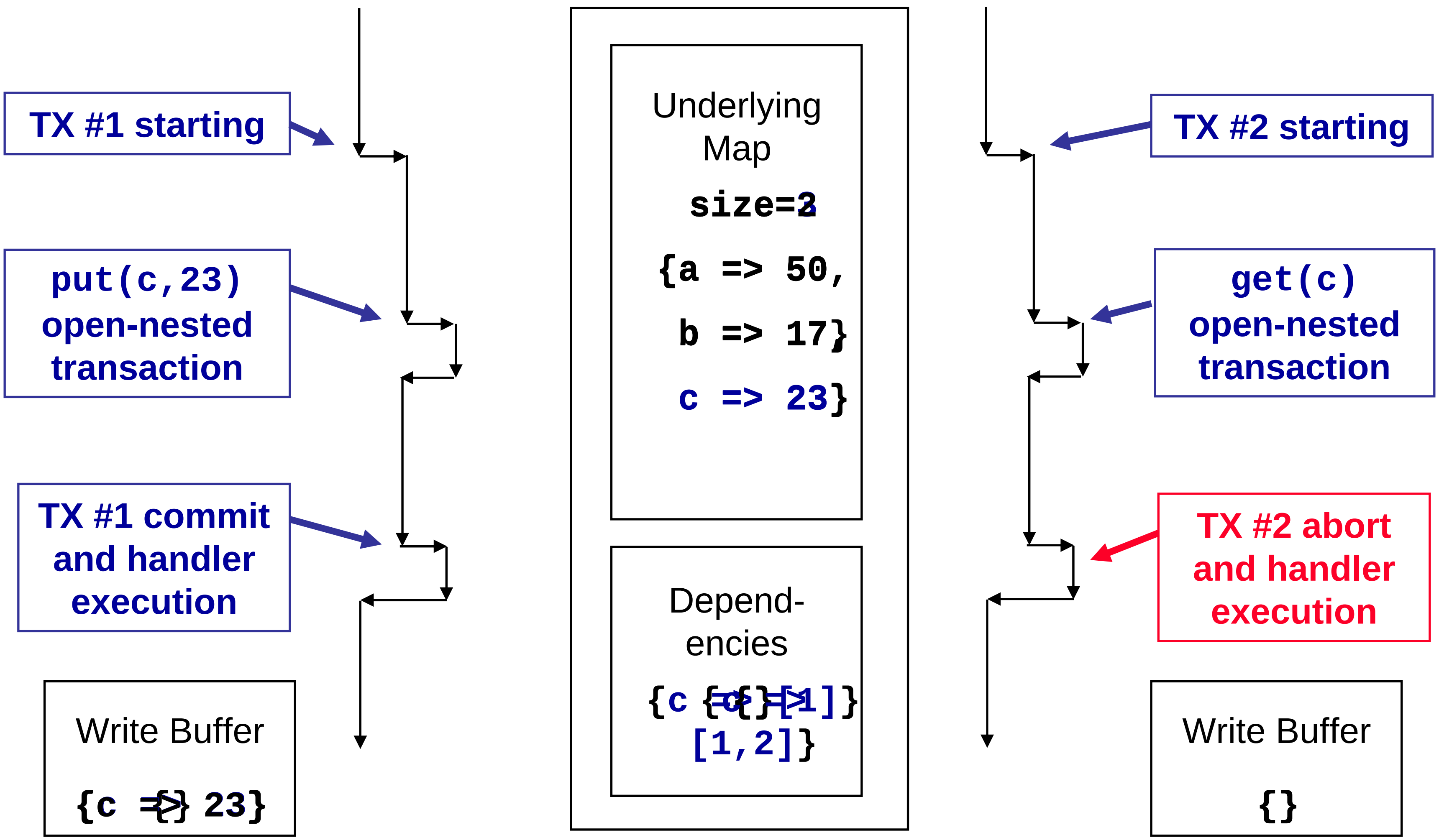
Implementing Transactional Collection Classes

General Approach	**Simplified Map example**
Read operations ▪Acquire semantic dependency ▪Open nesting reads underlying state	`get(key)` add dependencies on key returns value from underlying map
Write operations ▪Buffer changes until commit	`put(key,value)` writes to thread local buffer
On commit ▪Apply buffer to underlying state ▪Check for semantic conflicts	Apply buffer to underlying map, violate transactions that depended on the keys we are writing
On commit and abort ▪Release semantic dependencies	Remove key dependencies

Example of non-conflicting put operations



Example of conflicting put and get operations



Benefits of Semantic Concurrency Approach

Transactional Collection Class works with abstract type

- Can work with any conforming implementation
- HashMap, TreeMap, …

Avoids implementation specific violations

- Not just size and mod count
- HashTable resizing does not abort parent transactions
- TreeMap rotations invisible as well

High-contention SPECjbb2000 results

Java Locks

- Short critical sections

Atomos Baseline

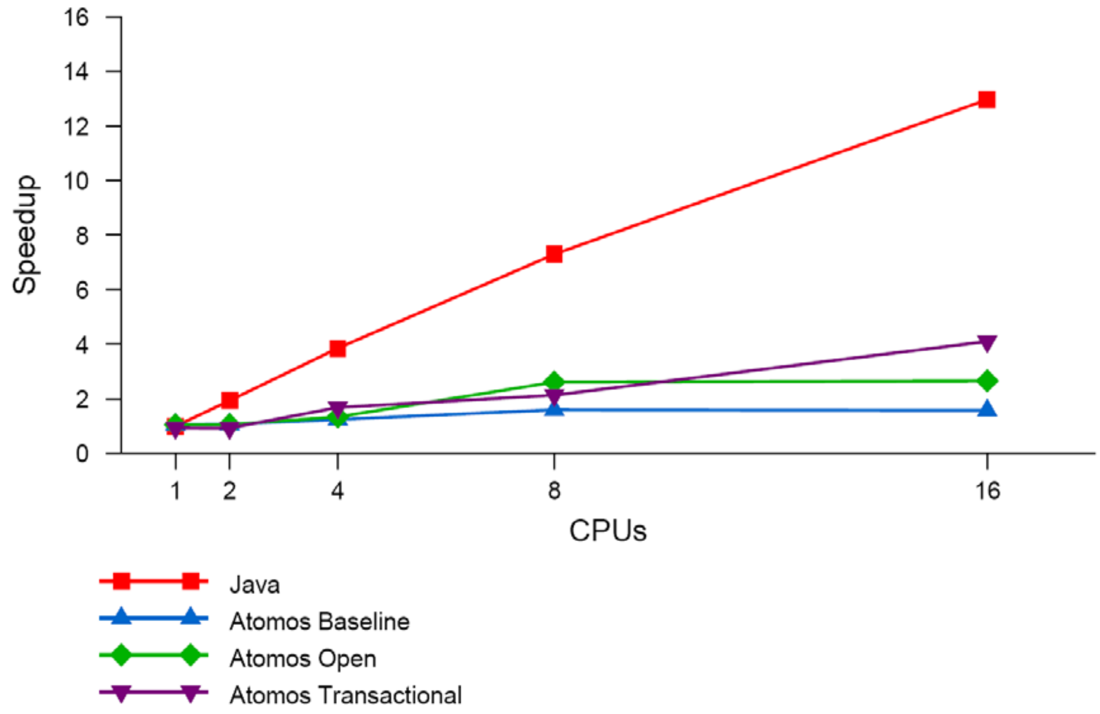
- Full protection of logical ops

Atomos Open

- Use simple open-nesting for UID generation

Atomos Transactional

- Change to Transactional Collection Classes



Performance Limit?

- Semantic violations from calls to SortedMap.firstKey()

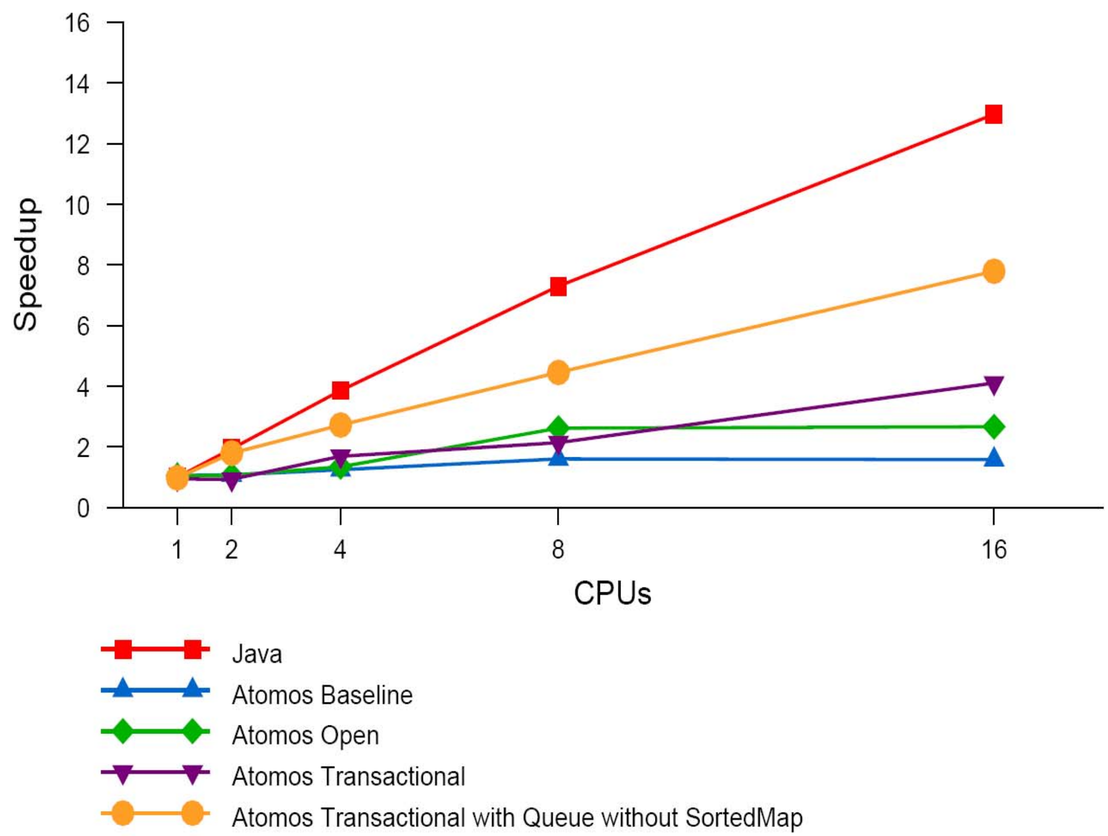
High-contention SPECjbb2000 results

SortedMap dependency

- SortedMap use overloaded
  1. Lookup by ID
  2. Get oldest ID for deletion

Replace with Map and Queue

1. Use Map for lookup by ID
2. Use Queue to find oldest



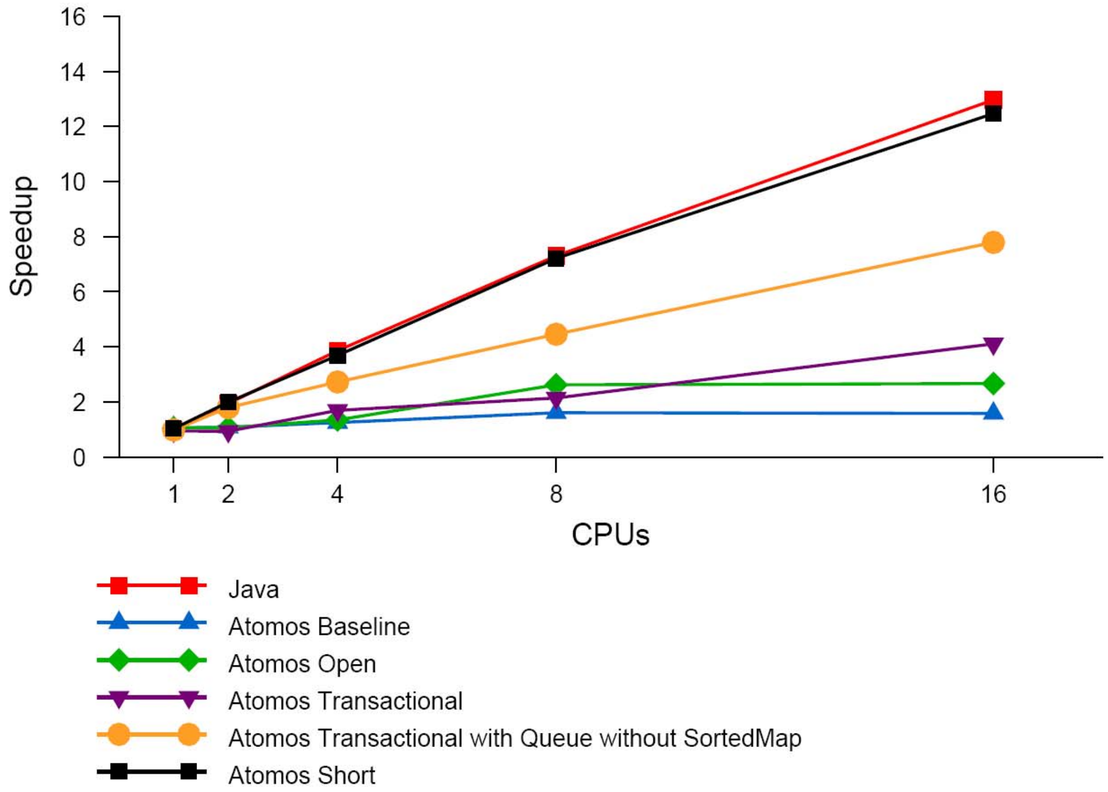
High-contention SPECjbb2000 results

What else could we do?

- Split larger transactions into smaller ones
- In the limit, we can end up with transactions matching the short critical regions of Java

Return on investment

- Coarse grained transactional version is giving almost 8x on 16 processors
- Coarse grained lock version would not have scaled at all



Focus on correctness

tune for performance

SPECjbb2000 Return on Investment

Version	Speedup on 16 CPUs	Effort	***Atomos 14 changes 7.8x*** ***Java 272 changes 13x***
Baseline	1.6	1 atomic statement	
Open	2.7	4 open statements	
Transactional	4.1	2 Transactional Map, 1 TxnSortedMap 2 transactional counters	
Queue	7.8	Change TxnSortedMap to TxnMap/TxnQueue (2 new calls: Queue.add & Queue.remove)	
Short	12.5	272 atomic statements	
Java	13.0	272 synchronized statements	

Semantic Concurrency Control Summary

Transactional memory promises to ease parallelization

- Need to support coarse grained transactions

Need to access shared data from within transactions

- While composing operations atomically
- While avoiding unnecessary data dependency violations
- While still having reasonable performance!

Transactional Collection Classes

- Provides needed scalability through familiar library interfaces of Map, SortedMap, Set, SortedSet, and Queue
- Removes need for direct use of open nested transactions

Overview

Motivation and Thesis

- How to make parallel programming of chip multiprocessors easier using transactional memory

Transactional Memory

- Concepts, implementation, environment

JavaT [SCP 2006]

- Executing Java programs with Transactional Memory

Atomos [PLDI 2006]

- A transactional programming language

Semantic concurrency control [PPoPP 2007]

- Improving scalability of applications with long transactions

Summary

Thesis:

If transactional memory is to *make parallel programming easier*, rather than just more scalable, the programming interface requires *more than simple atomic transactions*

JavaT

- Transactions alone cannot run all existing Java programs due to incompatibility of monitor conditional waiting

Atomos Programming Language

- Features to support reduced isolation and integration non-transactional operations through handlers

Transactional Collection Classes

- Using semantic concurrency control to improve scalability of applications using long transactions

Future Work

Transaction-aware I/O libraries

- Semantic concurrency control for structured files such as b-trees
- Support for automatically buffering OutputStreams and Writers
- Support for application logging within transactions

Integrating with other transactional systems (distributed transactions)

- Treat TM as resource manager like DB or transactional file system

Programming Language

- Language support for loop based parallelism
- Task-based, rather than thread-based, models

Virtual Machines

- Garbage Collector

Acknowledgements

- My wife Jennifer and kids Michael, Daniel, and Bethany
- My parents David and Elizabeth
- My advisors Kunle Olukotun and Christos Kozyrakis
- My committee Dawson Engler, Margot Gerritsen, John Mitchell
- Jared Casper, Hassan Chafi, JaeWoong Chung, Austen McDonald and the rest of TCC group for the simulator and everything else
- Andrew Selle and Jacob Leverich for all those cycles
- Normans Adams, Marc Brown, and John Ellis for encouraging me to go back to school
- Everyone at Ariba that made it possible to go back to school
- Olin Shivers and Tom Knight and the MIT UROP program for inspiring me to do research as an undergraduate
- Intel for my PhD fellowship
- DARPA, not just for supporting me for the last five years, but for employing my father for my first five years…