Transactional Collection Classes

Brian D. Carlstrom, Austen McDonald, Michael Carbin
Christos Kozyrakis, Kunle Olukotun

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

Transactional Memory

Promise of Transactional Memory (TM)

1. Make parallel programming easier
2. Better performance through concurrent execution

How does TM make parallel programming easier?

- Program with large atomic regions
- Keep the performance of fine-grained locking

Transactional Collection Classes

- Transactional versions of Map, SortedMap, Queue, …
- Avoid unnecessary data dependency violations
- Provide scalability while allowing access to shared data

Evaluating Transactional Memory

Past evaluations

- Convert fine-grained locks to fine-grained transactions
- Convert barrier style applications with little communication

Past results

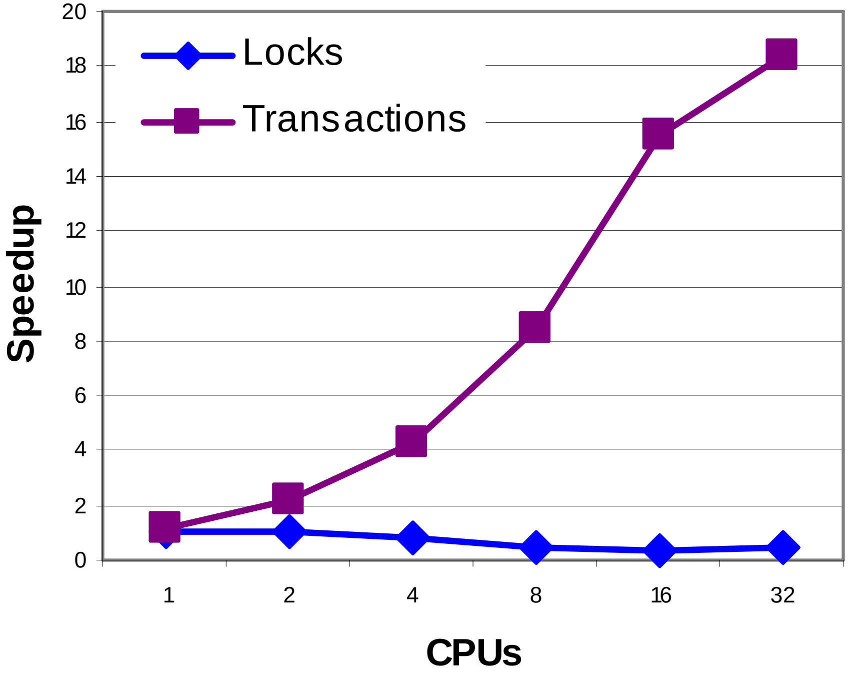
- TM can compete given similar programmer effort

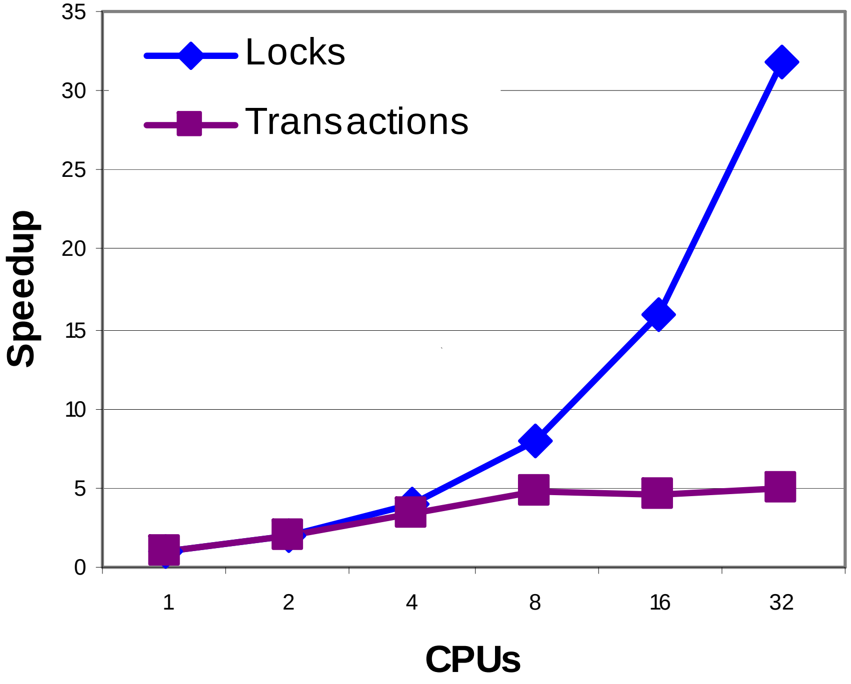
What happens when we use longer transactions?

TM hash table micro-benchmark comparison

Old: Many short transactions that each do only one Map operation

New: Long transactions containing one or more Map operations

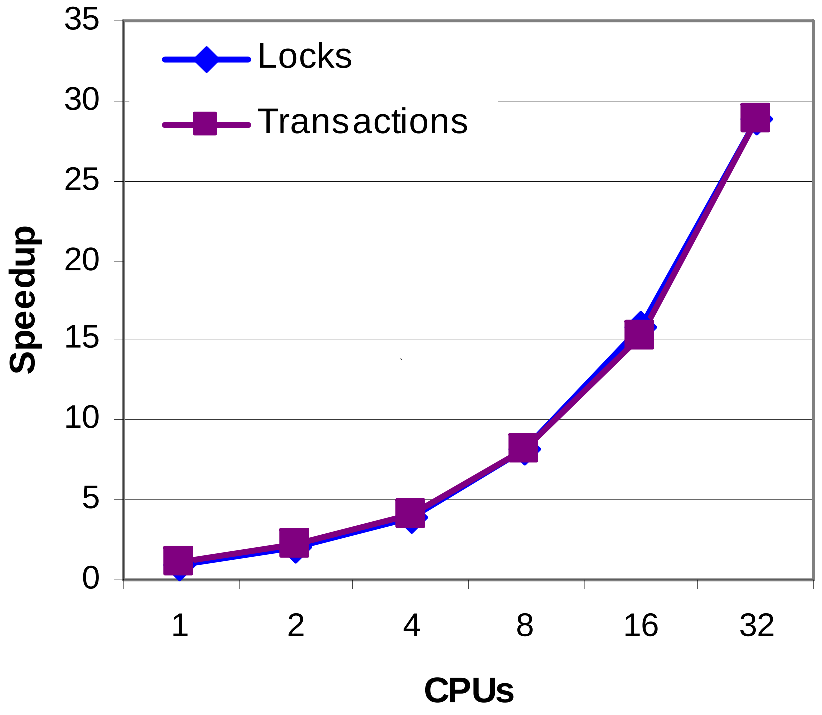




TM SPECjbb2000 benchmark comparison

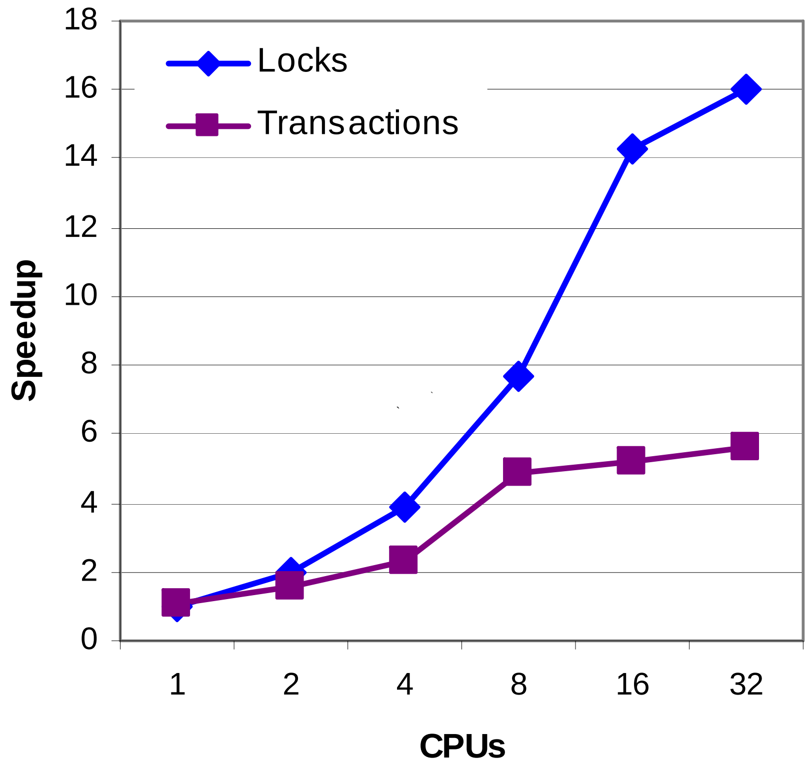
Old: Measures JVM scalability, but app rarely has communication

- 1 thread per warehouse, 1% inter-warehouse transactions



New: High contention - All threads in 1 warehouse

- All transactions touch some shared Map



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?

Composing Map operations

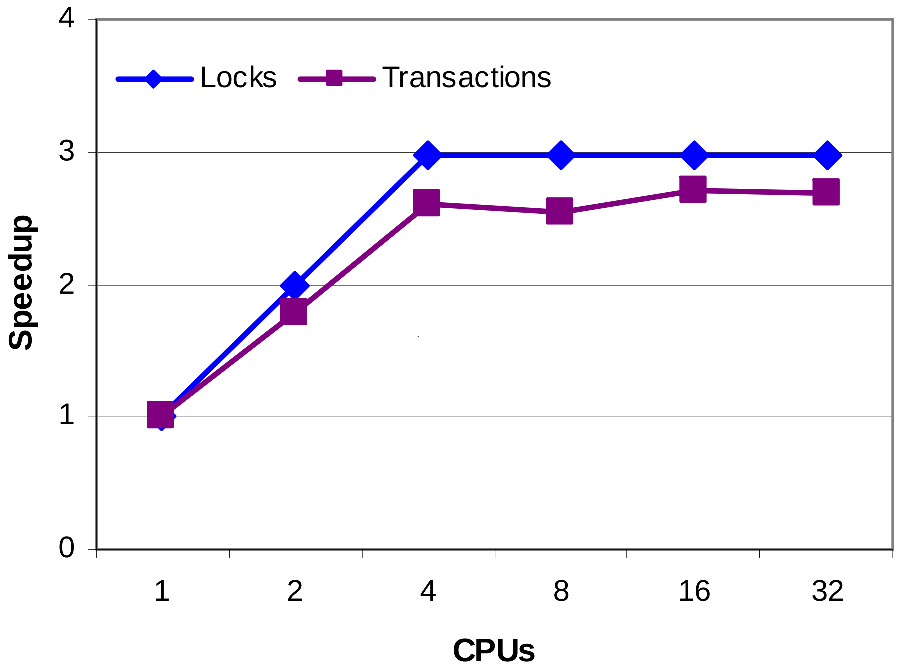
Suppose we want to perform two Map operations atomically

- With locks: take a lock on Map and hold it for duration
- With transactions: one big atomic block
- Both lousy performance

Use ConcurrentHashMap?

- Won’t help lock version
- Probabilistic approach hurts as number of operations per transaction increases

Can we do better?



Example compound operation:

```
atomic {
  int balance = map.get(acct);
  balance += deposit;
  map.put(acct, balance);}
```

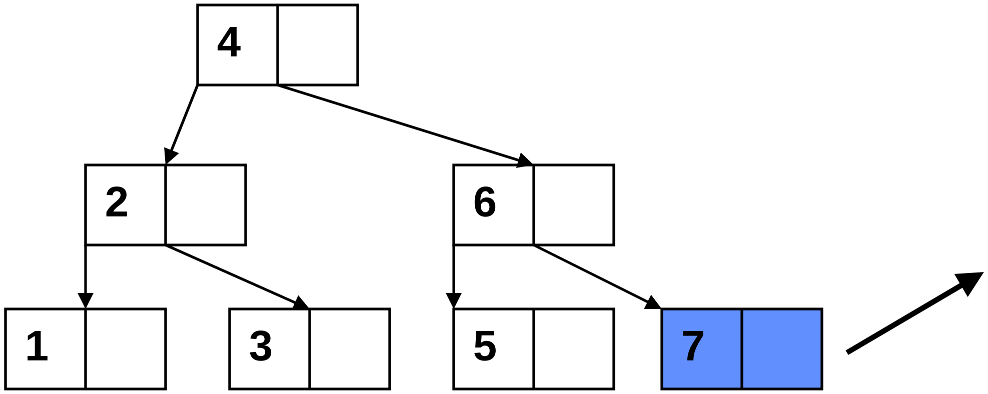
Semantic Concurrency Control

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

Transactional Collection Classes

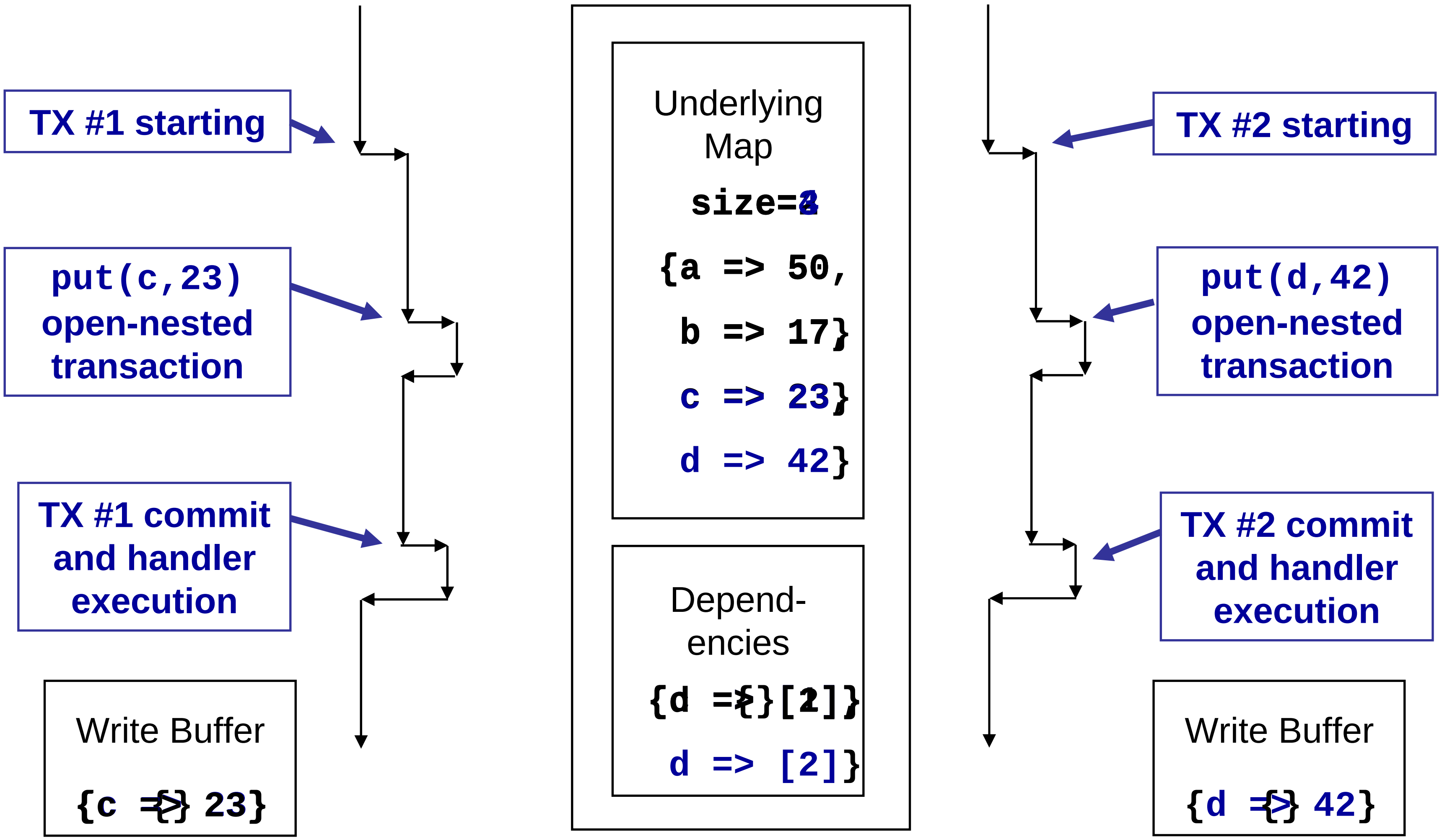
Our general approach

- Read operations acquire semantic dependency
  - Open nesting used to read class state
- Writes buffered until commit
- Check for semantic conflicts on commit
- Release dependencies on commit and abort

Simplified Map example

- Read operations add dependencies on keys
- Write operations buffer inserts and updates
- On commit we applied buffered changes, violating transactions that read values from keys that are changing
- On commit and abort we remove dependencies on the keys we have read

Example of non-conflicting put operations



TX #1 starting

`put(c,23)` open-nested transaction

TX #1 commit and handler execution

Write Buffer

`{c => 23}`

Underlying Map

`size=4`

`{a => 50,`

`b => 17}`

`c => 23}`

`d => 42}`

Depend-encies

`{d =>{}[2]}`

`d => [2]}`

TX #2 starting

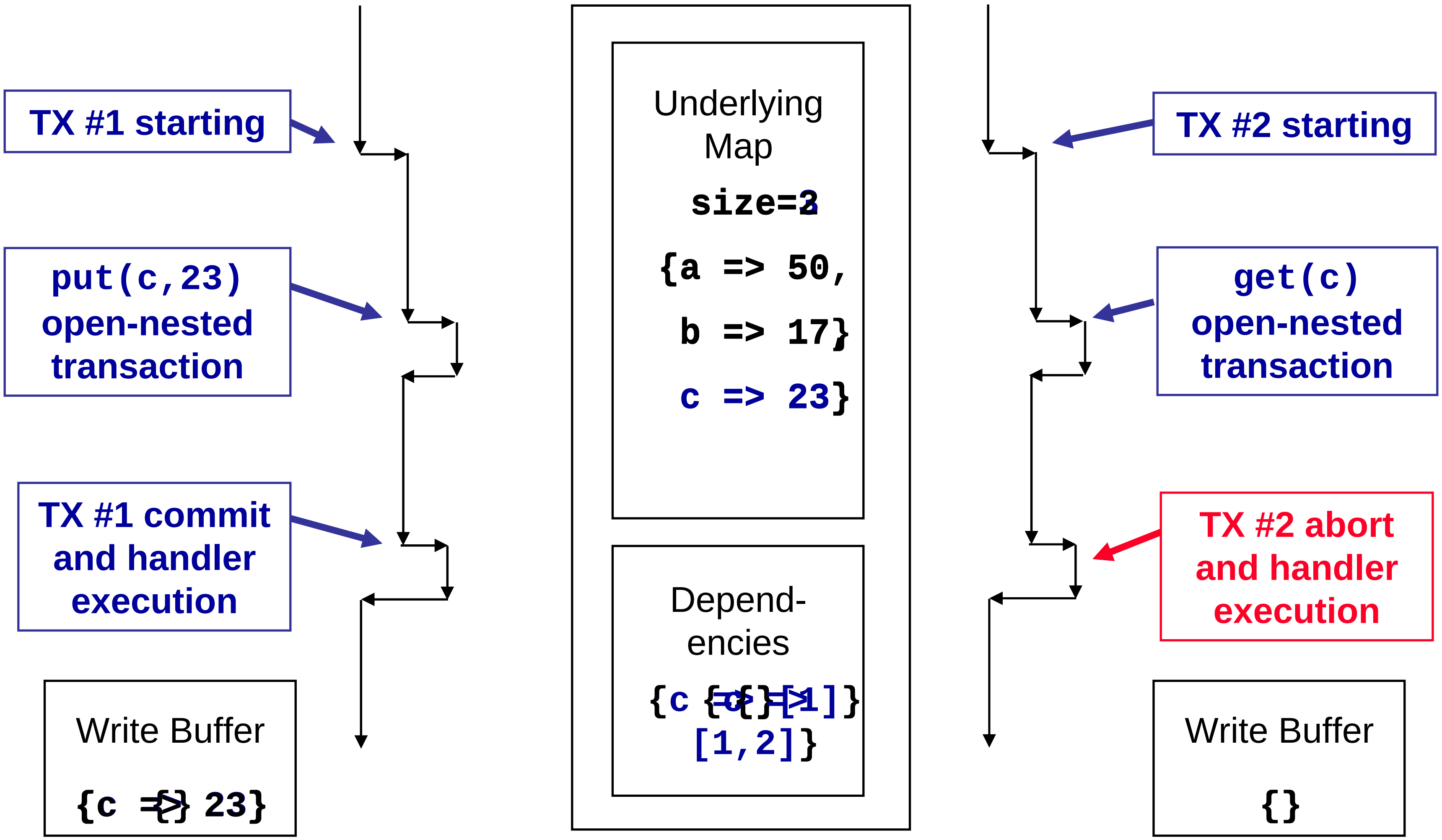
`put(d,42)` open-nested transaction

TX #2 commit and handler execution

Write Buffer

`{d => 42}`

Example of conflicting put and get operations



TX #1 starting

`put(c,23)` open-nested transaction

TX #1 commit and handler execution

Write Buffer

`{c =>{} 23}`

Underlying Map

`size=2` **`3`**

`{a => 50,`

`b => 17}`

`c => 23}`

Depend-encies

`{c {c{}=[>1]}`

`[1,2]}`

TX #2 starting

`get(c)` open-nested transaction

TX #2 abort and handler execution

Write Buffer

`{}`

Benefits of Semantic Concurrency Approach

Works 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

Making a Transactional Class

1. Categorize primitive versus derivative methods
   - Derivative methods such as isEmpty can be ignored
   - Often only a small fraction of methods are primitive
2. Categorize read versus write methods
   - Read methods do not conflict with each other
   - Need to focus on how write operations cause conflicts
3. Define semantic dependencies
   - Most difficult step, although still not rocket science
   - For Map, this involved deciding to track keys and size
4. Implement!

Making a Transactional Class

4. Implementation
    1. Derivative methods call primitive methods
    2. Read operations use open nesting
        - Avoid memory dependencies on committed state
        - Record semantic dependencies in shared state
        - Consult buffered state for local changes of our own write operations
    3. Write operations record changes in local state
    4. Commit handler
        - Transfers local state to committed state
        - Abort other transactions with conflicting dependencies
        - Releases dependencies
    5. Abort handler
        - Cleans up local state
        - Releases dependencies

Library focused solution

Programmer just uses the usual collection interfaces

- Code change as simple as replacing

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

- with

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

We provide similar interface coverage to util.concurrent

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

Primarily only library writers need to master implementation

- Seems more manageable work than util.concurrent effort

Paper details…

TransactionalMap

- Discussion of full interface including dealing with iteration

TransactionalSortedMap

- Adds tracking of range dependencies

TransactionalQueue

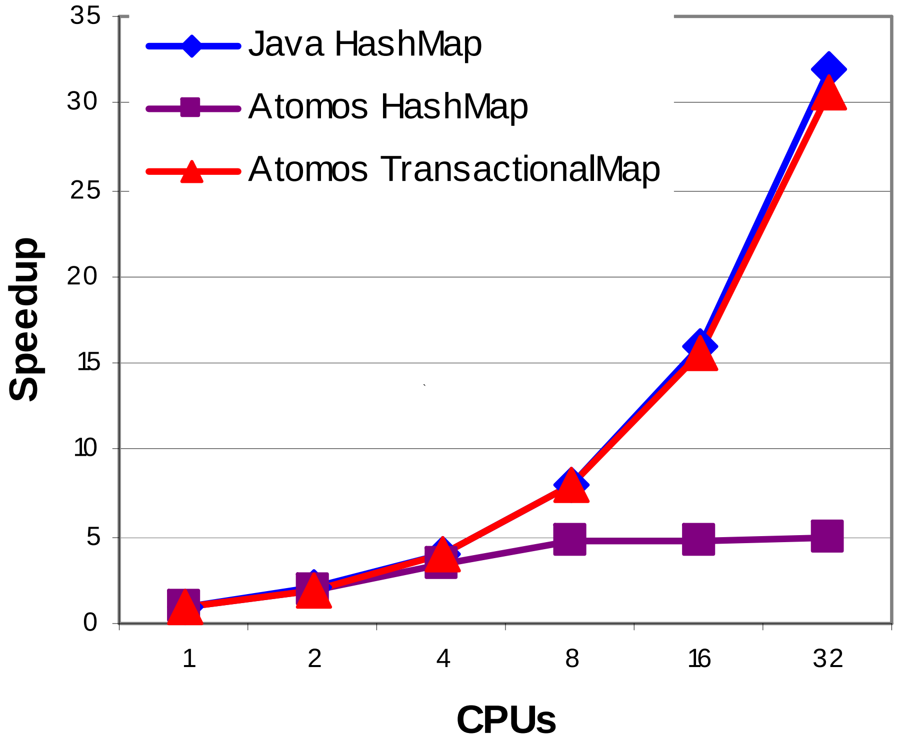
- Reduces serialization requirements
- Mostly FIFO, but if abort after remove, simple pushback

Evaluation Environment

- The Atomos Transactional Programming Language
  - Java - locks + transactions = Atomos
  - Implementation based on Jikes RVM 2.4.2+CVS
  - GNU Classpath 0.19
- Hardware is simulated PowerPC chip multiprocessor
  - 1-32 processors with private L1 and shared L2
- For details about the Atomos programming language
  - See PLDI 2006
- For details on hardware for open nesting, handlers, etc.
  - See ISCA 2006
- For details on simulated chip multiprocessor
  - See PACT 2005

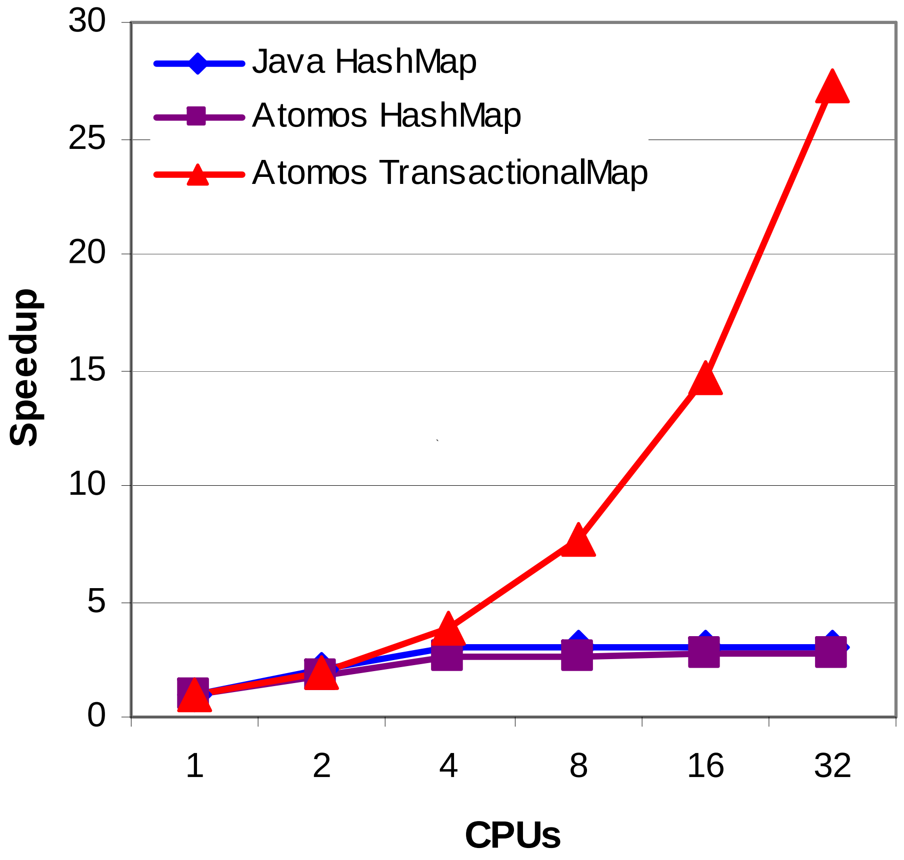
TestMap results

- TestMap is a long operation containing a single map operation
- Java HashMap with single lock scales because lock region is small compared to long operation
- TransactionalMap with semantic concurrency control returns scalability lost to memory level violations



TestCompound results

- TestCompound is a long operation containing two map operations
- Java HashMap protects the compound operations with a lock, limiting scalability
- TransactionalMap preserves scalability of TestMap



High-contention SPECjbb2000 results

Java Locks

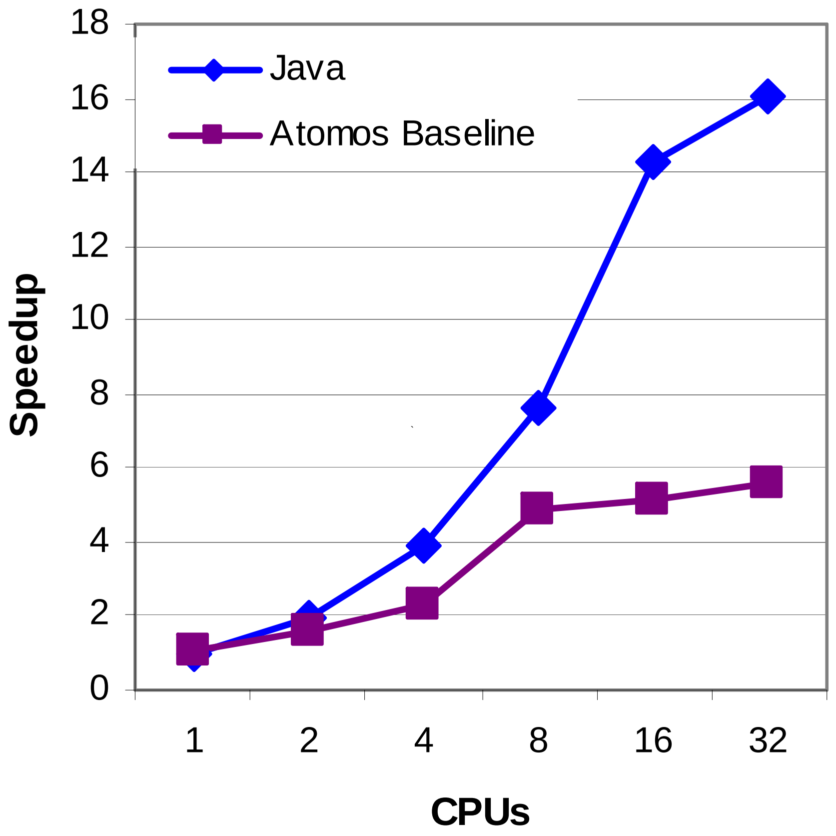
- Short critical sections

Atomos Baseline

- Full protection of logical ops

Performance Limit?

- Data dependency violations on unique ID generator for new order objects



High-contention SPECjbb2000 results

Java Locks

- Short critical sections

Atomos Baseline

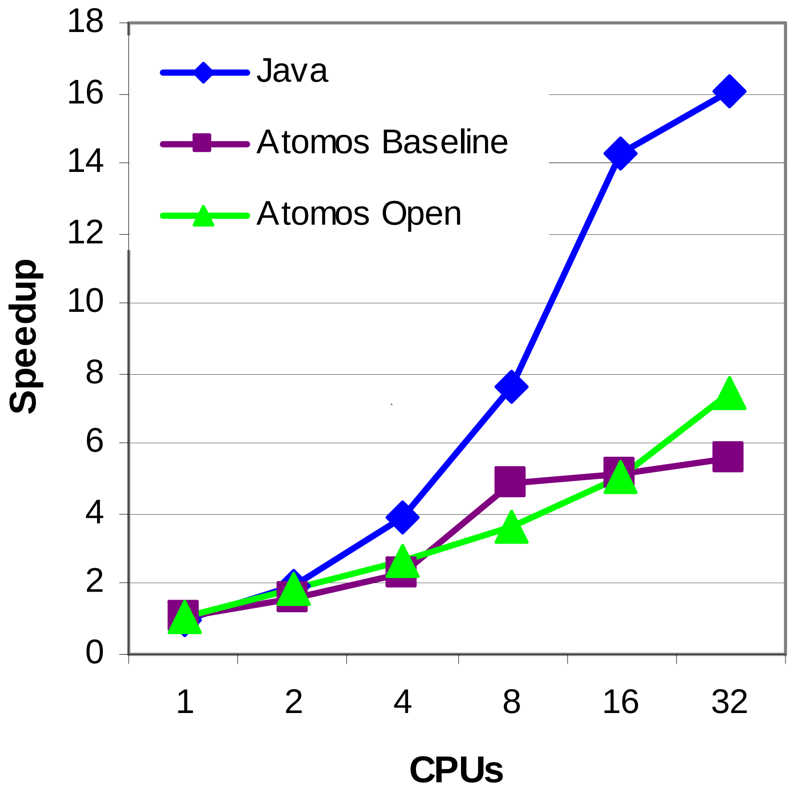
- Full protection of logical ops

Atomos Open

- Use simple open-nesting for UID generation

Performance Limit?

- Data dependency violations on TreeMap and HashMap



High-contention SPECjbb2000 results

Java Locks

- Short critical sections

Atomos Baseline

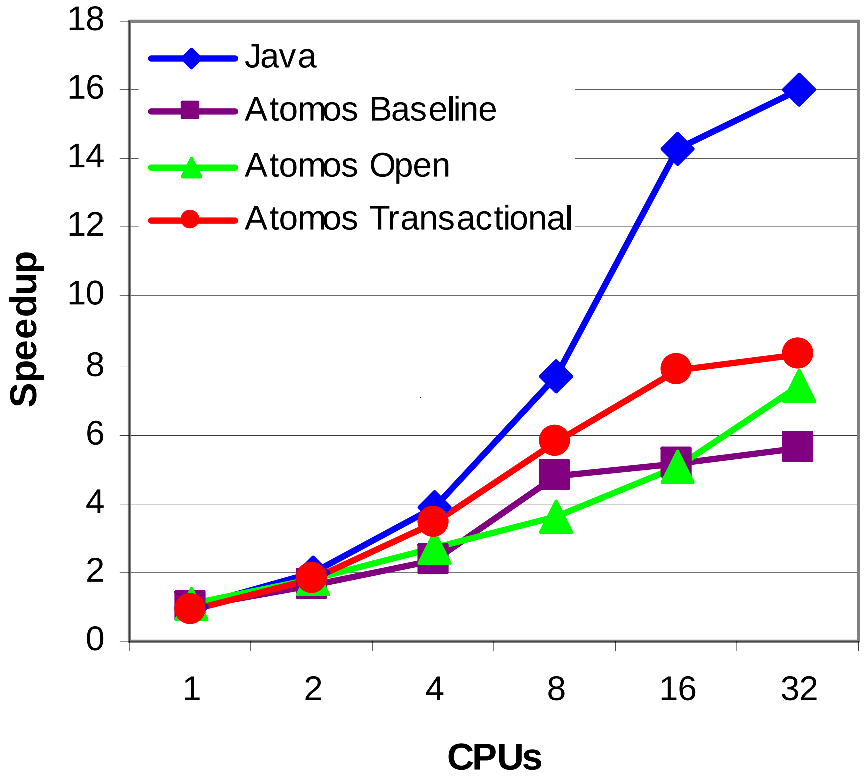
- Full protection of logical ops

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

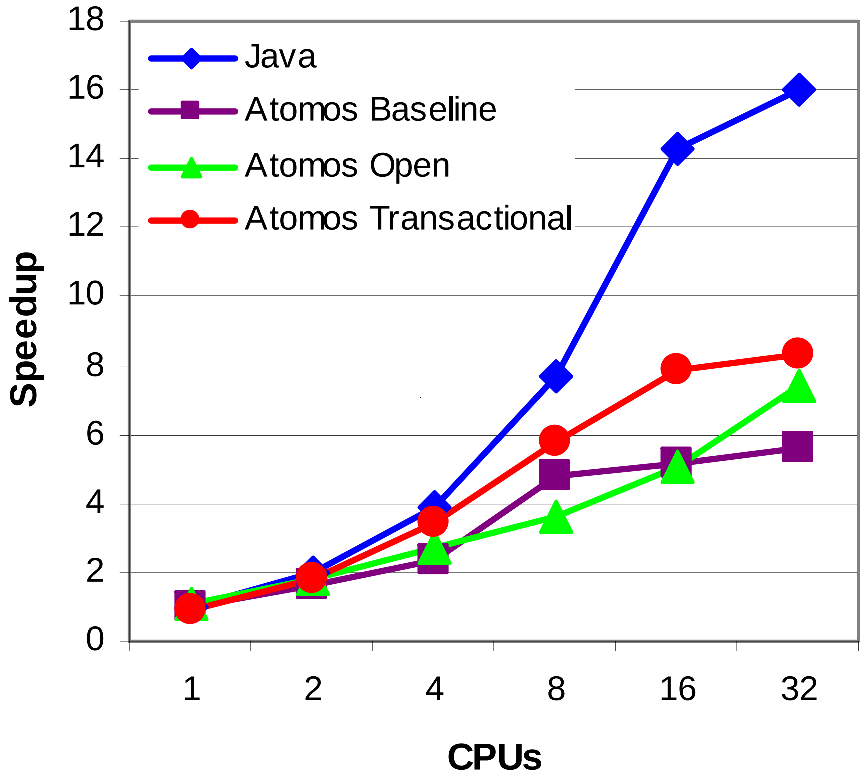
SortedMap dependency

- SortedMap use overloaded

1. Lookup by ID
2. Get oldest ID for deletion

Replace with Map and Queue

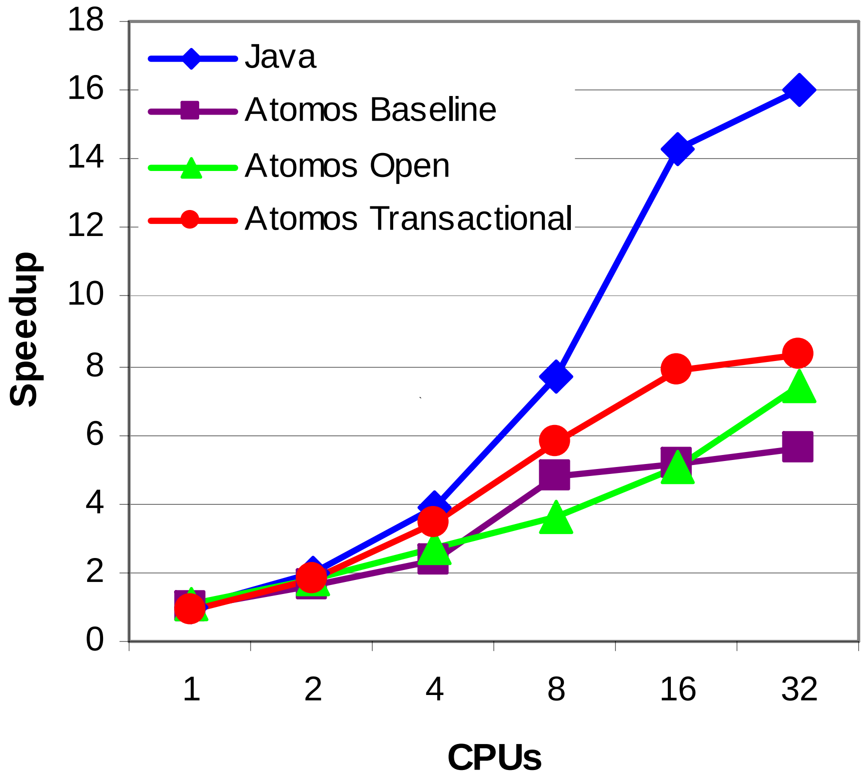
1. Use Map for lookup by ID
2. Use Queue to find oldest



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 8x on 32 processors
- Coarse grained lock version would not have scaled at all

Conclusions

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 dependency violations
- While still having reasonable performance!

Transactional Collection Classes

- Provides needed scalability through familiar library interfaces of Map, SortedMap, Set, SortedSet, and Queue