Transactional Execution of Java Programs

Brian D. Carlstrom, JaeWoong Chung, Hassan Chafi, Austen McDonald
Chi Cao Minh, Lance Hammond, Christos Kozyrakis, Kunle Olukotun

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

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

TCC Transactional Memory

- Continuous Transactional Architecture
  - "all transactions, all the time"
  - Transactional Coherency and Consistency (TCC)
    - Replaces MESI Snoopy Cache Coherence (SCC) protocol
  - At hardware level, two classes of transactions
    1. *indivisible transactions* for programmer defined atomicity
    2. *divisible transactions* for outside critical regions
  - Divisible transactions can be split if convenient
    - For example, when hardware buffers overflow

Translating Java to Transactions

- Three rules create transactions in Java programs
  1. **`synchronized`** defines an indivisible transaction
  2. **`volatile`** references define indivisible transactions
  3. **`Object.wait`** performs a transaction commit
- Allows us to run:
  - Histogram based on our ASPLOS 2004 paper
  - Benchmarks described in Harris and Fraser OOPSLA 2003
  - SPECjbb2000 benchmark
  - All of Java Grande (5 kernels and 3 applications)
- Performance comparable or better in almost all cases

Defining indivisible transactions

- **`synchronized`** blocks define indivisible transactions

```
public static void main (String args[]){
    a();                              a();      // divisible transactions
    synchronized (x){                 COMMIT();
        b();           ------>        b();      // indivisible transaction
    }                                 COMMIT();
    c();                              c();      // divisible transactions
}                                     COMMIT();
```

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

```
public static void main (String args[]){
    a();                              a();      // divisible transactions
    synchronized (x){                 COMMIT();
        b1();                         b1();     //
        synchronized (y) {                      //
            b2();      ------>        b2();     // indivisible transaction
        }                                       //
        b3();                         b3();     //
    }                                 COMMIT();
    c();                              c();      // divisible transactions
}                                     COMMIT();
```

Coping with condition variables

- In our execution, **`Object.wait`** commits the transaction
- Why not rollback transaction on **`Object.wait`**?
  - This is the approach of Conditional Critical Regions (CCRs) as well as Harris's **`retry`** keyword
  - This does handle most common usage of condition variables

```
while (!condition) wait();
```

Coping with condition variables

- However, need **`Object.wait`** commit to run current code
- 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 rollback, all threads think they are first to barrier
- With commit, barrier works as intended

Coping with condition variables

- 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

Coping with condition variables

- Not happy with unclean semantics
  - Most existing Java programs work correctly
  - Unfortunately no guarantee
- Fortunately, if you prefer rollback…
  - Barrier code example can be rewritten to use rollback
  - Presumably this is generally true…

Hardware and Software Environment

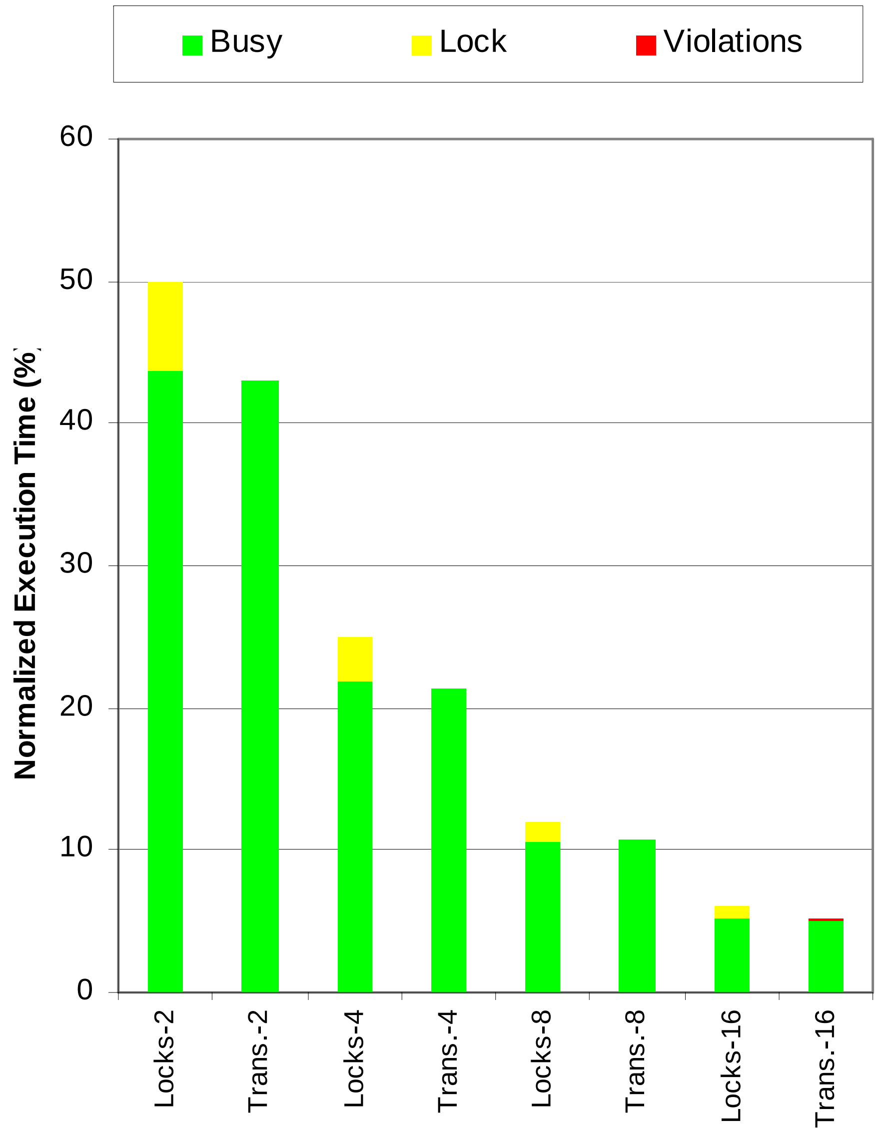
- The simulated chip multiprocessor TCC Hardware (See PACT 2005)

CPU	1-16 single issue PowerPC core
L1	64-KB, 32-byte cache line, 4-way associative, 1 cycle latency
Victim Cache	8 entries fully associative
Bus width	16 bytes
Bus arbitration	3 pipelined cycles
Transfer Latency	3 pipelined cycles
L2 Cache	8MB, 8-way, 16 cycles hit time
Main Memory	100 cycles latency, up to 8 outstanding transfers

- JikesRVM
  - Derived from release version 2.3.4
  - Scheduler pinned threads to avoid context switching
  - Garbage Collector disabled and 1GB heap used
  - All necessary code precompiled before measurement
  - Virtual machine startup excluded from measurement

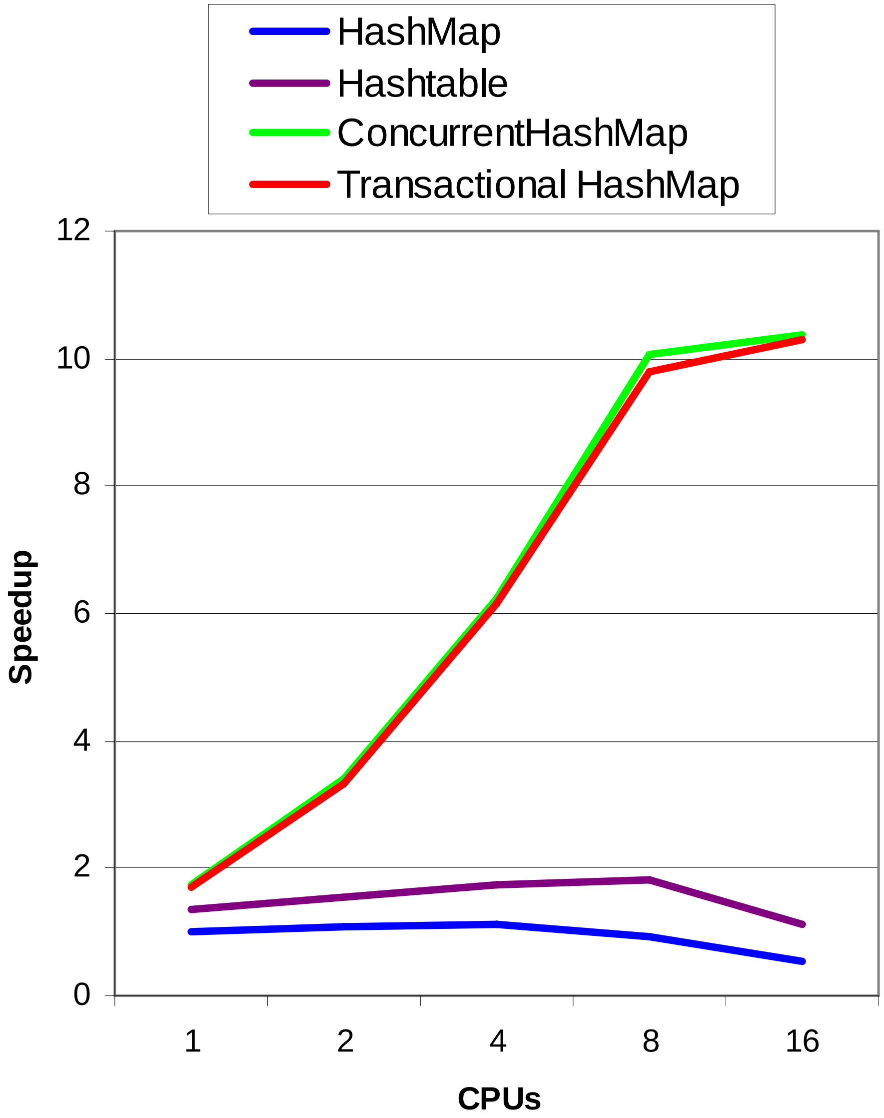
Transactions remove lock overhead

- SPECjbb2000 benchmark
- Problem
  - Locking is used because of 1% of operations than span two warehouses
  - Pay for lock overhead 100% of the time for 1% case.
- Solution
  - Transactions make the common case fast, time lost to violations not even visible in this example.



Transactions keep data structures simple

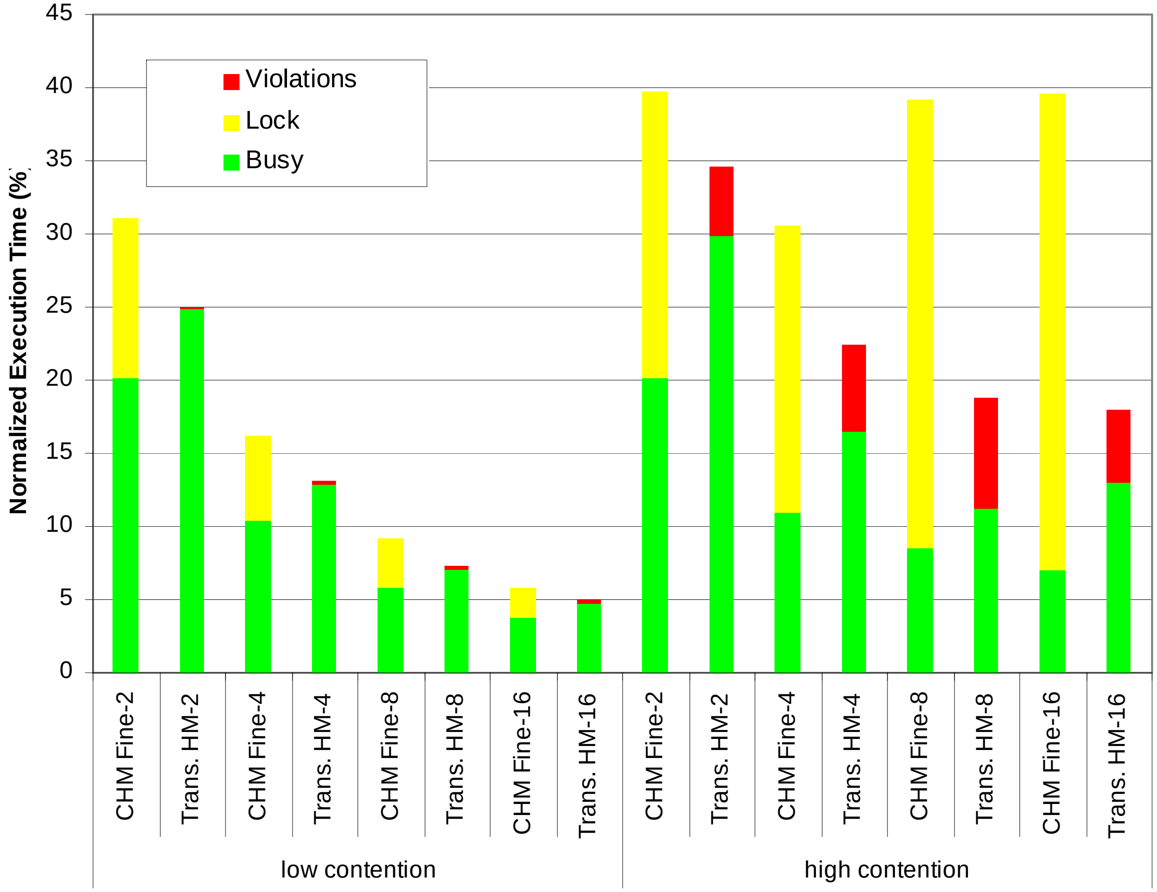
- TestHashtable
  - mix of read/writes to Map
- Problem
  - Java has 3 basic Map classes
  - Which to choose?
    - HashMap
      - No synchronization
    - Hashtable
      - Singe coarse lock
    - ConcurrentHashMap
      - Fine grained locking
- Solution
  - ConcurrentHashMap scales but has single CPU overhead
  - With transactions, just use HashMap and scale like CHM



Transactions can scale better with contention

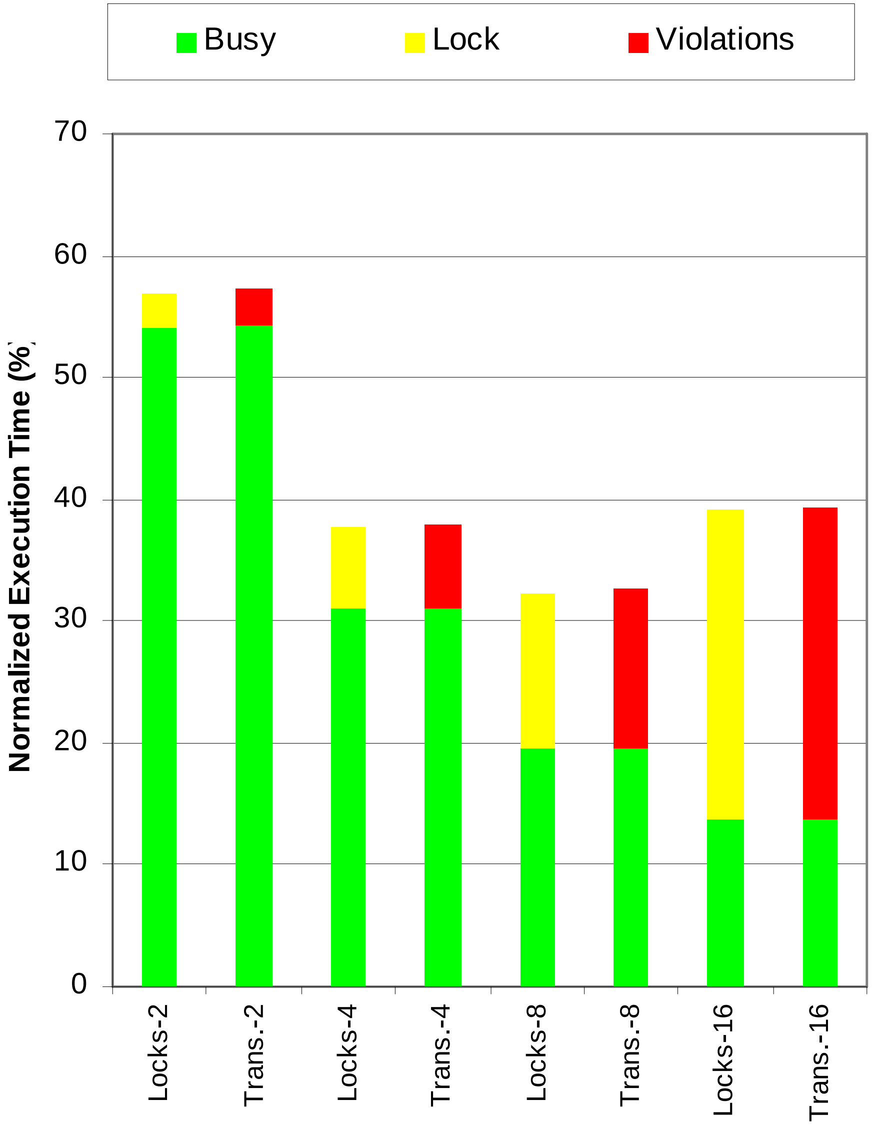
- TestCompound
  - Atomic swap of Map elements (low and high contention experiments)
  - Extra lock overhead compared to TestHashtable to lock keys

- Low Contention
  - Transactions have slight edge without lock overhead
- High Contention
  - CHM scales to 4 but then slows
  - Transactions scale to 16 cpus



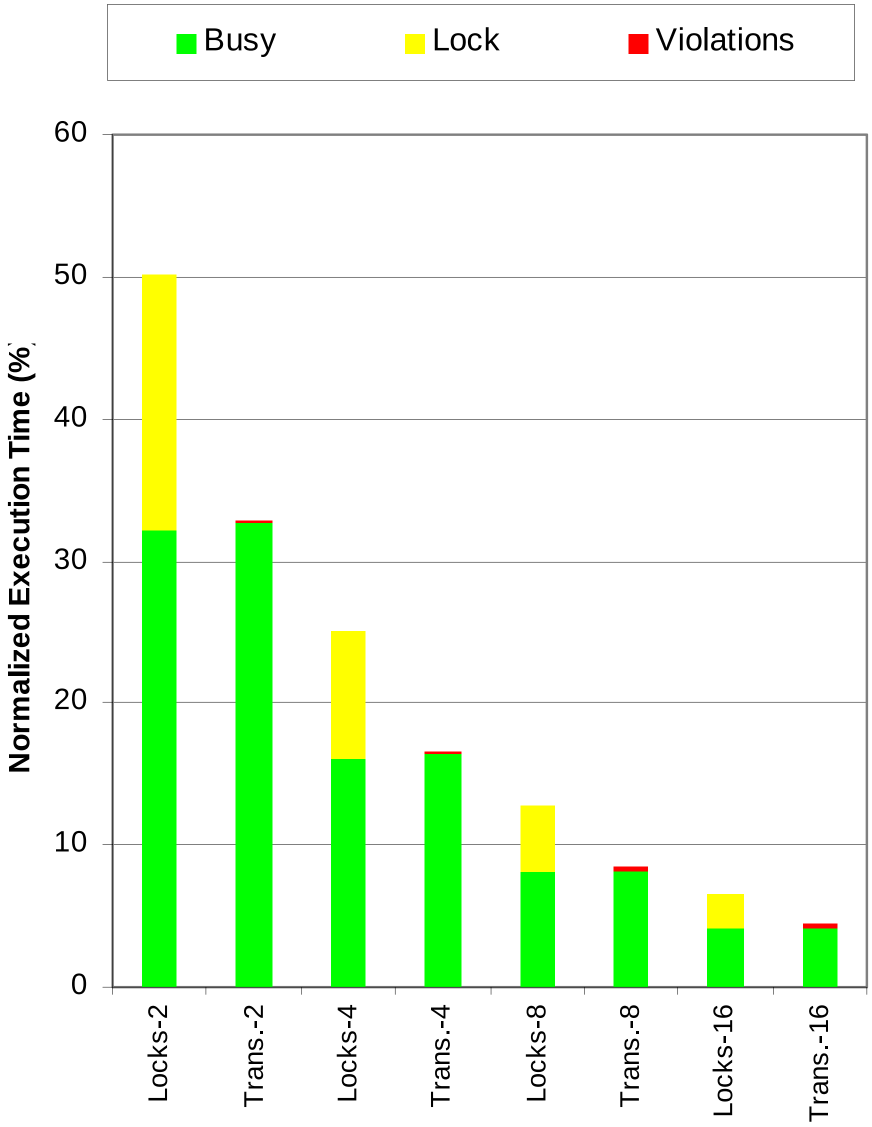
Java Grande Applications: MolDyn

- MolDyn
  - Time spent on locks close to time lost to violations
  - Both scale to 8 CPUs and slow at 16 CPUs



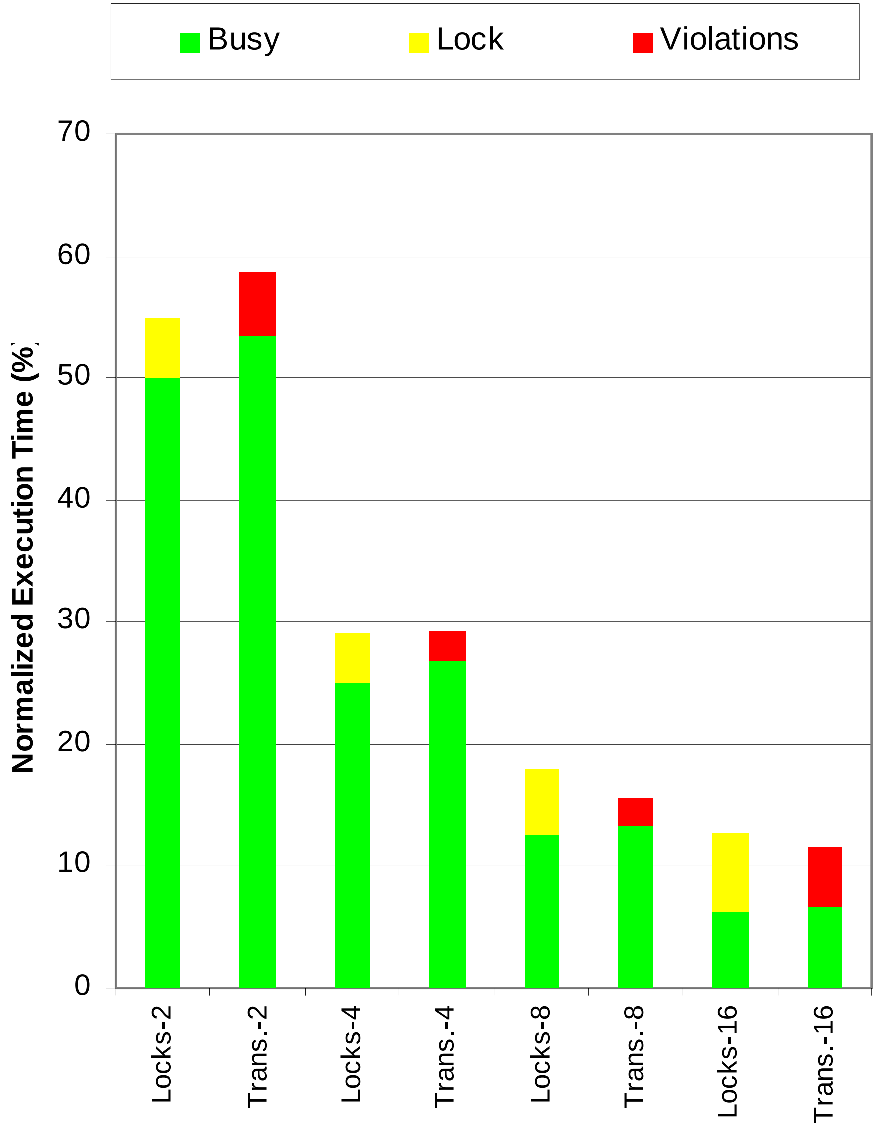
Java Grande Applications: MonteCarlo

- MonteCarlo
  - Similar to SPECjbb2000 (and Histogram in paper)
  - Performance difference attributable to lock overhead
  - Both scale to 16 CPUs



Java Grande Applications: RayTracer

- RayTracer
  - Another contention example
- 2 CPUs
  - Lock and Violation time approximately equal
  - Difference in Busy time attributable to commit overhead (see paper graph)
- 4 CPUs
  - Overall time about equal
  - Lock time as percentage of overall time has increased
- 8 CPUs
  - Transactions pull ahead as Lock percentage increases
- 16 CPUs
  - Transactions still ahead as Lock and Violation percentage grows



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