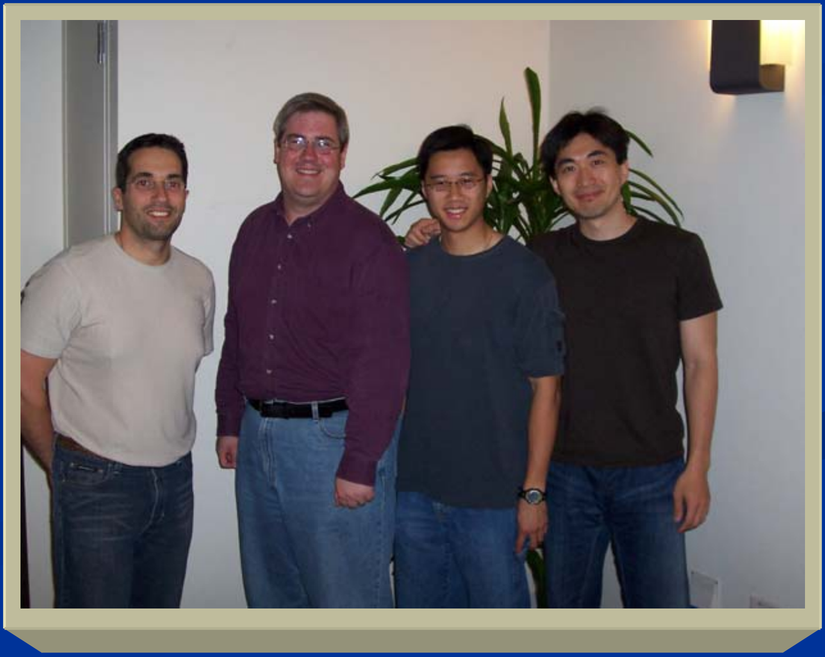
Parallelizing SPECjbb2000 with Transactional Memory

JaeWoong Chung,
Chi Cao Minh, Brian D. Carlstrom,
Christos Kozyrakis

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



The question we all share

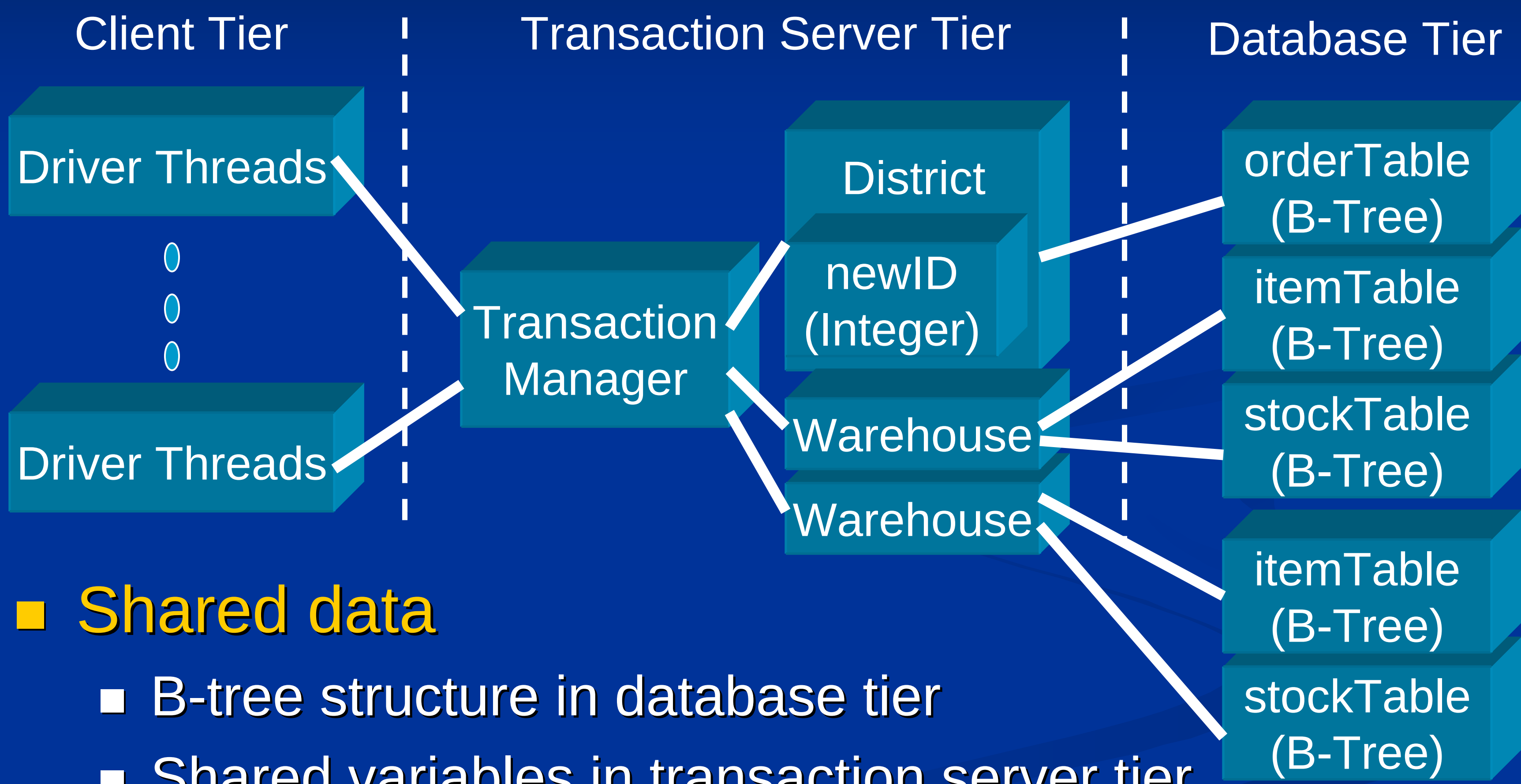
- TM provides
  - Speculative parallelism for sequential applications
  - Coarse-grain synchronization for parallel applications
- How can TM help parallelize complex applications?
  - Beyond basic data-structures
  - Can we get 90% of performance at 10% of the effort?
- We parallelized SPECjbb2000 with transactions
  - Irregular code from the enterprise domain

Contents

- SPECjbb2000 overview
- Methodology
- Transactional programming with
  - Flat transaction
  - Closed nesting
  - Open nesting
- Other interesting ideas
- Conclusion

SPECjbb2000 overview (1)

- 3 tier enterprise system



- Shared data
  - B-tree structure in database tier
  - Shared variables in transaction server tier
- Shared warehouse

SPECjbb2000 overview (2)

- TransactionManager::go()
  - 5 types of e-commerce transactions
  - We worked on this loop.

```
while (workToDo) {
   switch( e-commerce tx type ) {
        case new_order:
        case payment:
        case order_status:
        case delivery :
        case stock_level:
} }
```

Methodology

- Execution-driven simulator
  - Transactional Coherence and Consistency
  - 8 PowerPC core
  - 32K L1 and 256K L2 cache
  - 16 bytes bus
- Java environment
  - JikesRVM (JVM)
  - GNU classpath (Java runtime library)
  - *synchronized* blocks are removed.
    - For SPECjbb2000, too

Flat transaction

- Speculative parallelism
  - No analysis on potential races
  - 1 transaction for 1 e-commerce transaction
    - Equivalent to having 1 global lock

```
case new_order:
        atomic {  // generate new order }; break;
case payment:
        atomic {  // make payment }; break;
case order_status:
        atomic {  // check order status }; break;
case delivery :
        atomic {  // make delivery }; break;
case stock_level:
        atomic {  // check stock }; break;
```

- 3.09x speedup over coarse-grain locking
  - 62.7 % cycles lost due to violation

Analysis of violations

- Profiler provides us a violation report
- Violation sources
  - JikesRVM, GNU classpath
    - Minor impact
  - SPECjbb2000
    - New_order type takes almost 50% of all transactions.

Case new_order:

- ***// 1. initialize a new order e-commerce TX***
- ***// 2. assign a new order ID (newID++)***
- ***// 3. retrieve items/stocks from warehouse (itemTable, stockTable)***
- ***// 4. calculate the cost and update warehouse***
- ***// 5. record the order for delivery (orderTable)***
- ***// 6. display the processing result***

Shared Variable (newID++)

B-Tree (itemTable, stockTable)

B-Tree (orderTable)

Closed nesting (1)

- Child TX is merged to parent TX at commit.
  - Reduction of violation penalty
  - Parent RW-set <= Parent RW-set U Child RW-set
    - Closed nesting doesn't break the atomicity of original TX.

Core 0

```
// A is initially 0;
atomic {
  ....
  atomic {
      A++;  // 1
      ....
  }
}
```

Core 1

```
A = ;

= A; // 0

= A; // 1
```

Closed nesting (2)

- 2 closed nested transactions

Case new_order:

// 1. initialize a new order TX

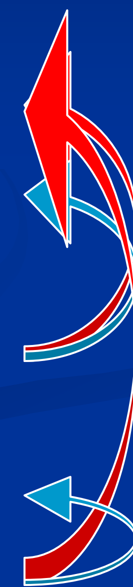
// 2. assign a new order ID (newID++)

// 3. retrieve items/stocks from warehouse (itemTable, stockTable)

// 4. calculate the cost and update warehouse

// 5. record the order for delivery (orderTable)

// 6. display the result



- 47.9 % reduction in violation cycles
- 5.36x speedup

Open nesting (1)

- Child TX communicates to all the other TXes
  - Child W-set is broadcasted through system.
    - Communication in the middle of a transaction
  - Child R-set is cleaned out.
    - Elimination of violations

Core 0

```
// A is initially 0;
atomic {
   ....
   open_atomic {
      A++;  // 1
      ....
   }

}
```

Core 1

```
A = ;

= A; // 1
A = ;
```

No conflict !

Open nesting (2)

- 1 open nested transaction

Case new_order:
- ***// 1. initialize a new order***
- ***// 2. assign a new order ID (newID++)***
- ***// 3. retrieve items/stocks from warehouse (itemTable, stockTable)***
- ***// 4. calculate the cost and update warehouse***
- ***// 5. record the order for delivery (orderTable)***
- ***// 6. display the result***

- 12 % reduction in the number of violation
- 4.96x speedup
- Compensation code for rollback
  - Here rollback results in only a gap in *newID*.

Other interesting ideas

- Mixture of open/close nesting
  - Advantages from both nested transactions

- Smaller flat transactions
  - *newID* is incremeted in a separate flat transaction.
  - In general, programmers should guarantee the correctness.
  - Composability is a challenge.

- Early release
  - For B-tree structure
  - See talk on "Early Release: Friend or Foe?"

Conclusion

- We parallelized SPECjbb2000 with transactions.
  - Flat transaction for speculative parallelism
    - A reasonable speedup is obtained.
  - Closed nesting
    - The violation penalty is reduced.
  - Open nesting
    - Violations are eliminated.

- Good speedup with small changes in source code
  - A couple of nested transactions

- We are heading for a transactional benchmark suite.
  - Realistic transactional applications

Questions?

Jae Woong Chung
jwchung@stanford.edu

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