Language Support for Lightweight transactions

Tim Harris & Keir Fraser
Presented by
Narayanan Sundaram
04/28/2008

Talk Outline

- Motivation
  - Problems with Concurrent programming
  - Types of concurrency control (Imperative vs Declarative)
- Support for Concurrency
  - Conditional Critical Regions (CCR)
  - Software Transactional Memory (STM)
- Implementation of STM in Java
  - Details
  - Optimizations
  - Modifications to JVM
- Results
- Conclusion

Concurrent Programming

- Concurrent programming is considered difficult because
  - Abstractions can be hard/unintuitive
  - Hard to write reliable and scalable software
  - Hard to monitor side-effects of concurrency control using locks (like priority-inversion and deadlocks)
  - Difficult performance optimizations

Styles of concurrency control

- Imperative (Mutex based)

```
public synchronized int get() {
  int result;
  while (items == 0) wait ();
  items --;
  result = buffer[items];
  notifyAll ();
  return result;
}
```

- Declarative (Conditional Critical Regions)

```
public int get() {
  atomic (items != 0) {
    items --;
    return buffer[items];
  }
}
```

Support for concurrency

- Programming Language features
  - Java's *util.concurrent* library provides atomic variables, locks, queues, thread pools etc
  - CCR-like features can be supported using semaphores, atomics
  - Static analysis can remove some pessimistic sharing scenarios, but cannot eliminate all
- Non-blocking algorithms can guarantee that as long as a thread is not contending with other threads for resources, it will make forward progress

Conditional Critical Regions

- CCRs are popular in teaching concurrency, but do not have efficient implementations
- There is no indication of which memory locations can be accessed inside CCRs
- Implementations that used mutex locks to allow only one CCR to execute at any time were inefficient
- This paper maps CCRs onto Software Transactional Memory (STM) and studies the implementation in the Java language

CCR implementation in Java

- atomic (condition) { statements; } is a CCR block
- CCR can access any field of any object
- Exceptions can be thrown from inside CCRs
- Native methods cannot be called inside CCRs
- CCRs can be nested
- Mutex locks can be used inside CCRs (with non-blocking implementations)

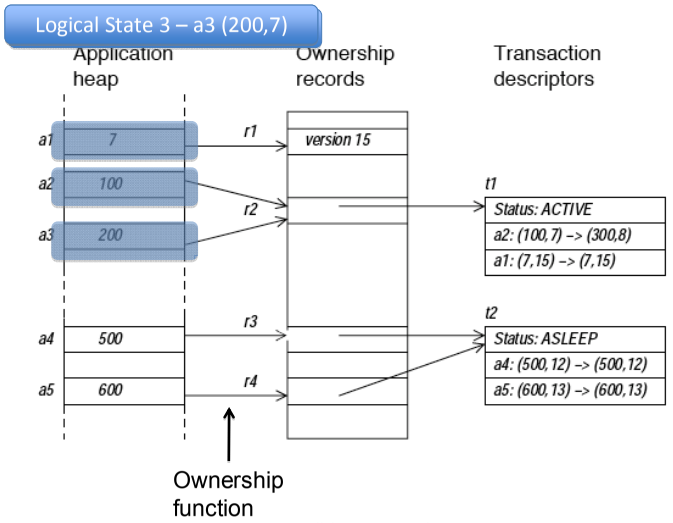
CCR implementation in Java (contd)

- Wait() and notify() methods should not be used inside CCRs
- Problems with class loading and initialization in Java
- "Correct" consistency is guaranteed if for any shared location
  - all accesses to it must be controlled by a given mutex, or
  - all accesses to it must be made within CCRs, or
  - it must be marked as volatile.

STM Implementation

- Five operations for transaction management
  1. STMStart()
  2. STMAbort()
  3. STMCommit()
     - Commits if the transaction is valid
  4. STMValidate()
     - Checks if the current transaction is valid
     - Used to check error conditions anytime inside a transaction
  5. STMWait()
     - Allows threads to block on entry to CCR
     - Simple implementation : STMWait() = STMAbort()
     - Efficient Implementation : Block the caller till there is an update to a location the transaction has accessed
- Two operations for memory access
  1. STMRead(addr a)
  2. STMWrite(addr a, stm_word w)

Heap Structure Modification



Implementation details

- STMStart
  - Allocate new descriptor and set status to ACTIVE
- STMAbort
  - Set status to ABORT
- STMRead
  - If no entry exists in orec, copy the value from heap
  - If an entry already exists in current transaction, use that value and version
  - Else, determine logical state of location and copy

Implementation details

- STMWrite
  - Perform a read on location
  - Set new_version <= old_version+1
  - Use this new_version for other accesses to that particular orec
- STMCommit
  - Acquire a lock to all locations accessed inside transaction
  - If that fails, one can either abort the owner, or wake it if in STMWait(), or abort current transaction
  - If lock acquisition succeeds, check if the versions match
  - If that succeeds, set status to COMMITTED, write back and release locks
  - Else, set status to ABORTED and release locks

Implementation

- STMCommit
  - While releasing the locks, set version of released orecs to new_version if committed
- STMValidate
  - Similar to STMCommit without the actual commit
- STMWait
  - Similar to STMCommit, acquiring locks to orecs
  - If successful, set status to ASLEEP
  - Other transactions that try to access the orec will wake up the waiting transaction

Optimizations

- More than 1 transaction can be asleep at an orec by including a thread list with each transaction descriptor
- Inefficiencies due to treating reads and writes as the same can be overcome having a READ_PHASE status where no locks are acquired, but other threads trying to write will have to abort
- Searches inside descriptors can be avoided by table lookups
- To allow non-blocking commits, stealing ownerships is allowed. However, the stealer has to take care of the victim's status (committed/aborted) to make appropriate changes in the end. Per-orec counters with number of pending transactions makes in-order commits easier to implement

Modification to JVM

- Both source-to-bytecode compiler and bytecode-to-native compiler need to be modified
- Each class has a second transactional variant of each method
- Bytecode-to-native compiler inserts STMValidate calls to detect looping
- Volatile fields are translated as small transactions

Experimental results

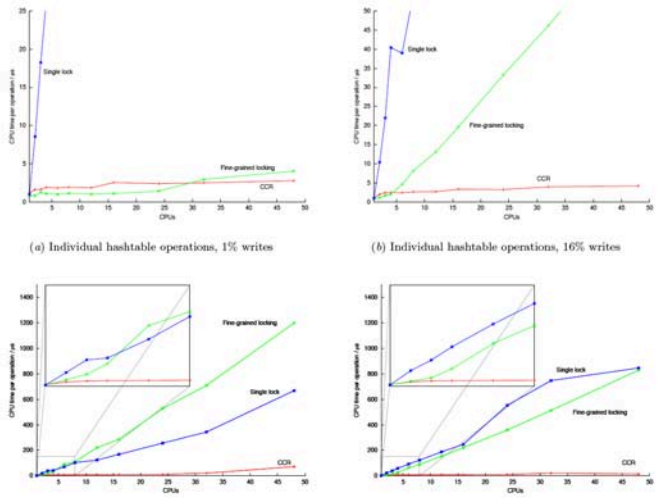
μs per operation

	1% updates			16% updates		
CPUs	CCR	S-l	FG-l	CCR	S-l	FG-l
1	1.8	1.1	0.9	1.9	1.1	0.9
2	1.8	3.3	0.9	2.0	7.9	1.0
3	2.1	25	1.3	2.4	23	1.1
4	1.8	30	1.1	2.4	30	1.4

	size=256			size=4096		
CPUs	CCR	S-l	FG-l	CCR	S-l	FG-l
1	4.8	2.1	2.6	5.1	2.3	2.7
2	6.2	17	5.0	6.3	17	4.4
3	7.2	27	6.4	7.2	28	6.3
4	7.4	37	8.3	7.5	40	6.9

Figure 6: Performance of the hashtable test (above) at 1% and 16% update rates and of the compound test (below) at table sizes of 256 entries and 4096 entries. In each case we record the mean number of microseconds to complete an operation using CCRs, using a single lock (S-l) and using fine-grained locking in `ConcurrentHashMap` (FG-l).

Results (contd.)



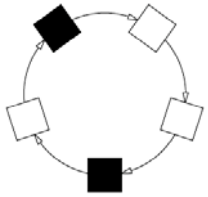
(a) Individual hashtable operations, 1% writes

(b) Individual hashtable operations, 16% writes

(c) Compound operations, 256-element table

(d) Compound operations, 4096-element table

Results – Wait test



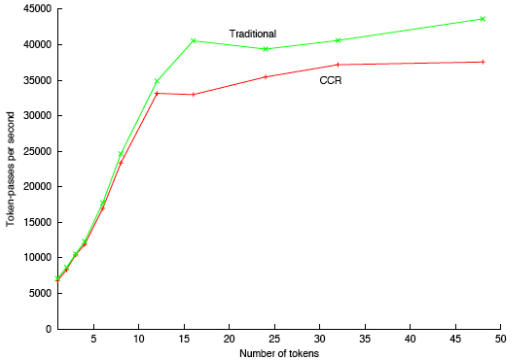


Figure 8: Performance of blocking operations running 48 threads.

Conclusion

- A complete STM implementation in Java is provided
- STM implementation is competitive with or better than other locking approaches
- STM scales better than locks
- Comparison done with other locking approaches - comparison with a single threaded non-locking algorithm is missing

Are Transactions the solution?

- Transactions offer several advantages
  - Easier to write than locks
  - Can offer good performance
  - Is declarative rather than imperative
  - Code composability is better
  - Can be implemented in hardware or software
  - Can solve priority-inversion and deadlocks that occur from lock usage