Language Support for Lightweight Transactions
Authors: Tim Harris and Keir Fraser

Presented By: Eduardo J Diaz
October 29, 2008

Introduction

• Conditional Critical Regions (CCRs)
  • One of the oldest proposals for concurrency control
  • Programmers indicate what groups of operations should be executed in isolation
  • The region can be guarded by an arbitrary boolean condition
  • Calling threads block until the guard is satisfied

Outline

• Introduction
• Language Integration
• Software Transactions
• Evaluation
Introduction

• No good implementation technique has been known
  • CCRs give no indication of what specific data items are to be accessed
  • If a thread blocks at the guard, there is no way to know exactly when it may be released
  • Performance is poor
    • Only one CCR to execute at any time
    • The guards are constantly re-evaluated

Introduction

• Mutual exclusion locks
  • Introduced to improve CCRs
  • Unrelated operations can execute concurrently
  • Condition variables were introduced to control blocking and unblocking

Example code using Mutual Exclusion Locks

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

Introduction

• However, mutual exclusion locks lead to the following difficulties:
  • Idioms are often forgotten or mis-understood
  • Mutual exclusion prevents operations that do not conflict to proceed
  • There is no check that the data accesses made are protected by the locks that are held
  • If a thread is preempted while holding the lock for a shared resource, then no other thread can safely use the resource
Introduction

• This talk will be about a mapping of CCRs onto a Software Transactional Memory (STM) which
  • Allows dynamically non-conflicting executions to operate concurrently
  • Re-evaluates CCR conditions only when one of the shared variables involved may have been updated
  • Uses a non-blocking implementation

Language Integration

• Details of the design
  • What operations and method invocations should be permitted within a CCR?
  • What kinds of shared data may be accessed?
  • What guarantees are made about concurrent access to data items outside CCRs?
  • How do CCRs inter-operate with existing features for concurrency control?

Language Integration

• The design is motivated by two principles:
  • CCRs should be able to enclose code with as few restrictions as possible
  • The system should permit an implementation which does not impose a high overhead in parts of an application where CCRs are not used

Identifying CCRs

• Defines a CCR
  • The thread executing the CCR sees the updates it makes proceed according to the usual single-threaded semantics
  • All other threads observe the CCR to take place atomically at some point between its start and its completion
Data Accessible to CCRs

- CCRs can access any field of any object
  - Enables code reuse
  - Avoids having to re-implement library classes before they could be used
  - Aids the sharing of memory locations between transactional and non-transactional accesses.

Native Methods within CCRs

- The current design generally raises a runtime exception if a native method is invoked
- Some built-in native methods are treated as special cases in which their behavior is either
  - Thread local
    - cloning an object
    - computing an object’s identity hash value
  - Relates to synchronization and therefore requires special handling

Nested CCRs

- Their entire assembly appears to execute atomically at a time satisfying all of the conditions
- The programmer must ensure that all the conditions can be satisfied

Example:

```java
atomic (x == 1) {
atomic (x == 0) {
  ...
}
}
```

Inter-operating with Existing Synchronization Mechanisms

- Important for enabling code reuse
- The system ensures that all mutexes are available at the point at which it appears to atomically take effect
  - Precludes the risk of deadlock
  - Enables the use of mutexes for sharing data between access within CCRs and external access
Inter-operating with Existing Synchronization Mechanisms

• It is not possible to ascribe useful semantics to a wait operation on a condition variable
  • The wait operation always blocks
  • This makes it impossible to identify a single point at which the entire CCR appears atomic.
• For symmetry, notification is also prohibited

Consistency Model

• In order to guarantee sequential consistency, all accesses to shared location must either:
  • be controlled by a given mutex
  • must be marked as volatile
  • must be made within CCRs

Class Loading with CCRs

• Class loading and initialization occurs at some point between when a CCR begins and the point at which it appears to take place atomically
  • Avoids the difficulty in classes whose initialization involves the creation of a thread or calls to native methods

Software Transactions

• An STM was developed as the basis for their implementation of CCRs
• Has the following features:
  • No reserved space is needed in the locations being accessed
  • It requires only word-sized updates to be atomic when accessing heap locations
Software Transactions

- The permanent structure used to co-ordinate transactions can be statically allocated outside the application heap
- Outside transactions, access to non-volatile heap locations uses standard memory reads and memory writes
- Read operations, whether in transactions or otherwise, do not cause any updates to shared memory.

STM Interface

- Operations for transaction management
  - void STMStart()
  - void STMAbort()
  - boolean STMCommit()
  - boolean STMValidate()
  - void STMWait()

STM Interface

- Operations for performing memory accesses
  - stm_word STMRead(addr a)
  - void STMWrite(addr a, stm_word w)
- May only be used while a transaction is active
Heap Structure

• The STM uses three kinds of data structures

Heap Structure: Application Heap

• Holds the objects allocated by the application

Heap Structure: Ownership Records (orecs)

• Used to co-ordinate transactions
• Each orec holds either
  • A version number
  • The current owner for the addresses that associate with it

Heap Structure: Ownership Records (orecs)

• Each time a location in the application heap is updated, the version number must be incremented
• Version numbers are used to detect whether a transaction may be committed
Heap Structure: Ownership Records (orecs)

- The status field indicates that the transaction is either:
  - ACTIVE
  - COMMITTED
  - ABORTED
  - ASLEEP

Heap Structure: Transaction Descriptors

- Each access is described by a transaction entry specifying
  - The address in question
  - The old and new values to be held there
  - The old and new version numbers of those values
**Heap Structure: Transaction Descriptors**

**STM Operations**

- **STMStart**
  - Allocates a fresh descriptor and initializes its status field to ACTIVE
- **STMAbort**
  - Writes ABORTED into a descriptor’s status field

**STM Operations: STMRead Example**

- For transaction t1, STMRead(a1) will
  1. Get 7 from the App Heap
  2. Get 15 from the oreccs
  3. Create a new transaction entry (te)
  4. Set
     - te.old_value = 7
     - te.new_value = 7
     - te.old_version = 15
     - te.new_version = 15

- **STMRead**
  - If the current descriptor already contains an entry (te) for the requested location
    1. Return te.new_value
  - Otherwise
    1. Determine the value and version of the location.
    2. Initialize a new entry with the value seen as old value and as new value.
    3. Record the version seen as old version
    4. If the descriptor already contains an entry for this orecc
       - copy that entry’s new_version number to this entry
       Otherwise use old_version
STM Operations

STMRead Example

- Now, for transaction t1, STMRead(a1) would simply return 7

STM Operations

STMWrite
1. Perform and STMRead on the location to be written
2. Set te.new_value to the value being written
3. Set te.new_version to te.old+version+1
4. Copy the new_version number to any other entries relating to the same ore
c

STM Operations

STMWrite Example

- For transaction t1, STMWrite(a2,300) will
1. Call STMRead(a2)
2. Set
   - te.new_value=300
   - te.new_version=7+1=8

STM Operations

STMCommit
1. Temporarily acquire each of the orecs it needs
2. If it can acquire them all
   i. Update the descriptor’s status field from ACTIVE to COMMITTED
   ii. Make any updates to the application heap
3. Release each of the orecs
STM Operations

- **STMCommit** may abort while acquiring an if:
  - The logical contents of that location was not consistent with the version expected in the entry
  - The transaction has encountered another transaction active on the same orec.

STM Operations

- **STMValidate**
  - It checks that the ownership records associated with each location accessed in the caller’s current transaction contain the version number held in the transaction descriptor.
  - Validation succeeds if every record holds the expected value.
  - Otherwise, validation fails

STM Operations: STMCommit Example

- For transaction t1, STMCommit will:
  1. Acquire r1 and r2
  2. Update status to COMMITTED
  3. Set:
     - Value of a2=300
     - Version of a2=8
  4. Release r1 and r2

STM Operations

- **STMWait**
  - Try to acquire all of the orecs relating to the transaction
  - If successful
    - Set its status field to ASLEEP
    - Leave references to its descriptor installed at the acquired orecs
Performance Summary

- Compared to other mechanisms, implementations based on CCRs
  - Have higher initial costs
  - Scale well
- This implementation is well suited to applications in which concurrent operations are likely to be dynamically non-conflicting

Ease of Programming CCRs

- Based on anecdotal observations
  - Reasoning about their behavior is made easier by the ability to consider the enclosed statements as a single step in an operational model
  - They have close analogies with the concept of database transactions
    - Simple semantics
    - Isolated execution

Ease of Programming CCRs

- Compared to the design of simple shared data structures
  - The synchronized keyword is replaced with atomic on each method
  - For data structures which involve blocking using wait and notify, the CCR-based design design expresses the preconditions directly

Questions?