Jacek Karaśkiewicz, 2012

Language Support for Lightweight Transactions

Overview

Tim Harris, Keir Fraser
University of Cambridge Computer Laboratory

Plan of presentation

- Standard approach to concurrency
  - Problems with standard techniques
- „New" old idea – Conditional Critical Regions (CCR's)
  - Advantages over previous solutions
  - Implementation
  - Performance
  - Possible future development

Standard approach to concurrency

- Multiple threads
    - Mutual-exclusion locks
    - Condition variables controlling access to shared data

Problems with standard approach

Consider this simple code:

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

Problems with standard approach

- There is no check that the data accesses made are protected by the locks that are held,
- `get()` operation cannot proceed concurrently with `put()` operation (because of mutual-exclusion locks), even though they don’t have to conflict
- Strange looking constructs like `wait()` surrounded by a while loop

CCR (Conditional Critical Regions)

- Allow programmers to indicate *which* groups of operations should be executed in isolation, rather than *how* to enforce it through some concurrency control mechanism,
- Allow guarding regions by arbitrary boolean conditions, which causes a thread to be blocked until a guard is satisfied,
- Eliminate the downsides of previous solution.

CCR

- A basic syntax:

```
atomic (condition) {
        statements;
}
```

- Now our `get()` method looks like this:

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

CCR

- How to implement this mechanism? We use Software Transactional Memory (STM),
- STM groups together series of memory accesses and makes them appear atomic,
- Allows dynamically non-conflicting executions to operate concurrently,
- Non-blocking implementation is used, preventing deadlocks and priority inversions.

Non-blocking design

- CCR implementation should be non-blocking,
- Non-blocking design is a design in which a failure of any number of threads cannot prevent other threads from making progress,
- Non-blocking design used in this algorithm may be put into *obstruction-freedom* category.

Non-blocking design

- *Obstruction-free* algorithm guarantees, that any thread can progress as long as it doesn’t contend with other threads for access to any location,
- This construct is strong enough to prevent deadlocks and priority inversions from happening.

Language integration

- The basic syntax is (as we presented earlier):

```
atomic (condition) {
      statements;
}
```

- The *condition* may be simple `true`,
- A thread executing the CCR sees the updates it makes according to the usual single-threaded semantics,
- Other threads observe the CCR to take place atomically at some point between its start and completion,
- *Exactly-once* execution of statements.

CCR’s features

- CCR’s are allowed to access *any* field of *any* object,
- They cannot execute, though, native methods (which could contain arbitrary memory accesses),
- CCR’s can be nested,
- `wait()`, `notify()`, `notifyAll()` methods inside CCR are forbidden.

Software Transaction Memory (STM)

- The implementation of CCR is based on STM
- Hardware assumptions:
  - Atomic word-sized memory accesses
  - Atomic word-sized compare and swap (CAS) instruction
  - Available in all major architectures

STM interface

- Transaction management:
  - `void STMStart();`
  - `void STMAbort();`
  - `boolean STMCommit();`
  - `boolean STMValidate();`
  - `void STMWait();`

STM interface

- STMStart:
  - Begins new transaction within the executing thread,
- STMAbort:
  - Aborts the transaction in progress by the executing thread,
- STMCommit:
  - Attempts to commit the transaction in progress by the executing thread (returns true if succeeds, false otherwise),

STM interface

- STMValidate:
  - Indicates whether the current transaction would be able to commit,
- STMWait:
  - Allows thread to block on entry to a CCR

STM interface

- Memory accesses:
    - `stm_word STMRead(addr a)`
    - `void STMWrite(addr a, stm_word w)`

STM interface

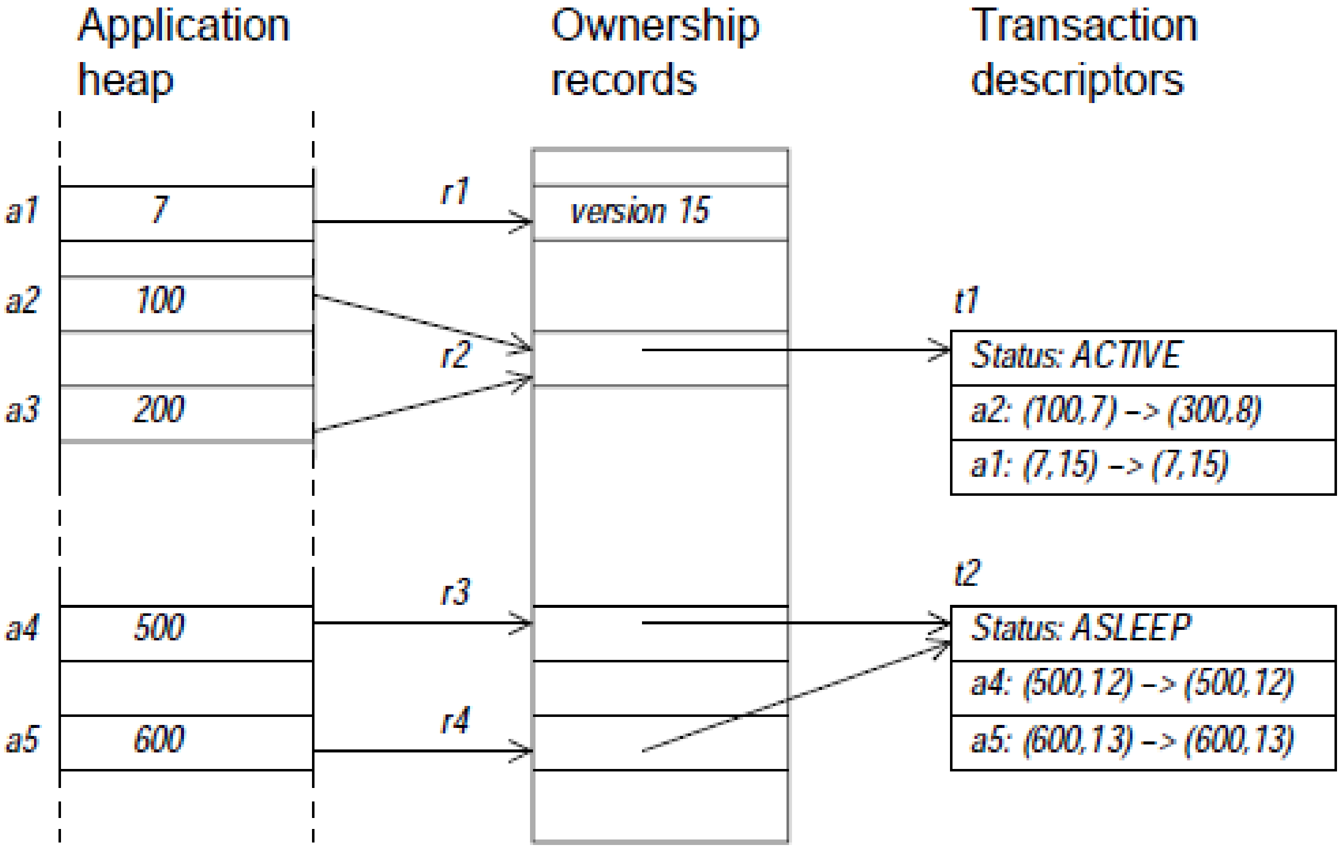
- Now the basic *atomic* block can be written in terms of STM interface:

```
boolean done = false;
while (!done) {
        STMStart();
        try {
                if (condition) {
                        statements;
                        done = STMCommit();
                } else
                        STMWait();
        } catch (Throwable t) {
                done = STMCommit();
                if (done)
                        throw t;
        }
}
```

Heap structure

- 3 kinds of data structure:
  - application heap (with actual data),
  - ownership records (orecs), used to coordinate transactions,
  - transaction descriptors.
- *Ownership function*, which maps each address in the application heap to an associated orec,
- Descriptors are never re-used.

Heap structure - example



Heap structure – ownership records

- An orec holds either a *version number* or a *current owner* (descriptor) for the addresses that associate with it,
- *Version numbers* indicate whether a transaction can be committed,
- A *version number* is incremented each time a location in application heap is updated,
- *Version numbers* are never re-used in the same ownership record.

Heap structure – transaction descriptors

- *Transaction descriptors* show current status of each active transaction and the accesses made to the application heap,
- Each access is described by a *transaction entry*, that contains:
    - address,
    - old and new values,
    - old and new version numbers.
- Each descriptor also has a status field, that may have one of the values: ACTIVE, COMMITTED, ABORTED, ASLEEP.

Heap structure – transaction descriptors

- A descriptor is *well-formed* if for each associated *ownership record* it either:
    - *Contains at most one entry associated with that record,*
    - *Contains many entries associated with that record, but the old and new version numbers are the same in all of them.*
- *Descriptors* in our implementation are maintained well-formed.

Logical state concept

- Each address in the application heap is connected to some logical state,
- Logical state can be described as a pair (x, y), where x is a value held at the address, and y is a version number associated with it,
- Logical state can be computed by an analysis of the heap structure.

STM Operations

- STMStart:
    - Allocates a fresh descriptor and sets its status to ACTIVE
- STMAbort:
    - Changes the value in the status field to ABORTED
- STMRead:
    - If current descriptor already contains an entry for requested location, then returns *new value*,
    - Otherwise determine the logical state of the location and initialize a new descriptor entry.

STM Operations

- STMWrite:
  - Writes new value and increases *version number* by 1,
- STMCommit:
  - Tries to acquire each of the *ownership records* it needs – then (if successful) updates the status field to COMMITTED, makes all necessary changes to application heap and releases *ownership records*.

STM Operations

- STMValidate:
  - Checks whether *version numbers* held in ownership records are equal to *version numbers* held in transaction descriptor entries,
- STMWait:
  - Aborts current transaction and blocks a caller until an update may have been committed to one of the locations accessed by the transaction.

Optimizations

- Multiple sleeping threads
  - More than one thread can sleep on the same location,
- Read sharing
- Avoiding searching
- Non-blocking commit

Implementation in JVM

- Modifications:
  - *atomic* blocks are treated as methods,
  - To each class a second method table is added – it holds references to transactional versions of its methods (compiled on demand) and is used by method invocations within transactional methods,
  - Compiler also is responsible for inserting STMValidate() calls to detect internal looping in transactions that cannot commit.

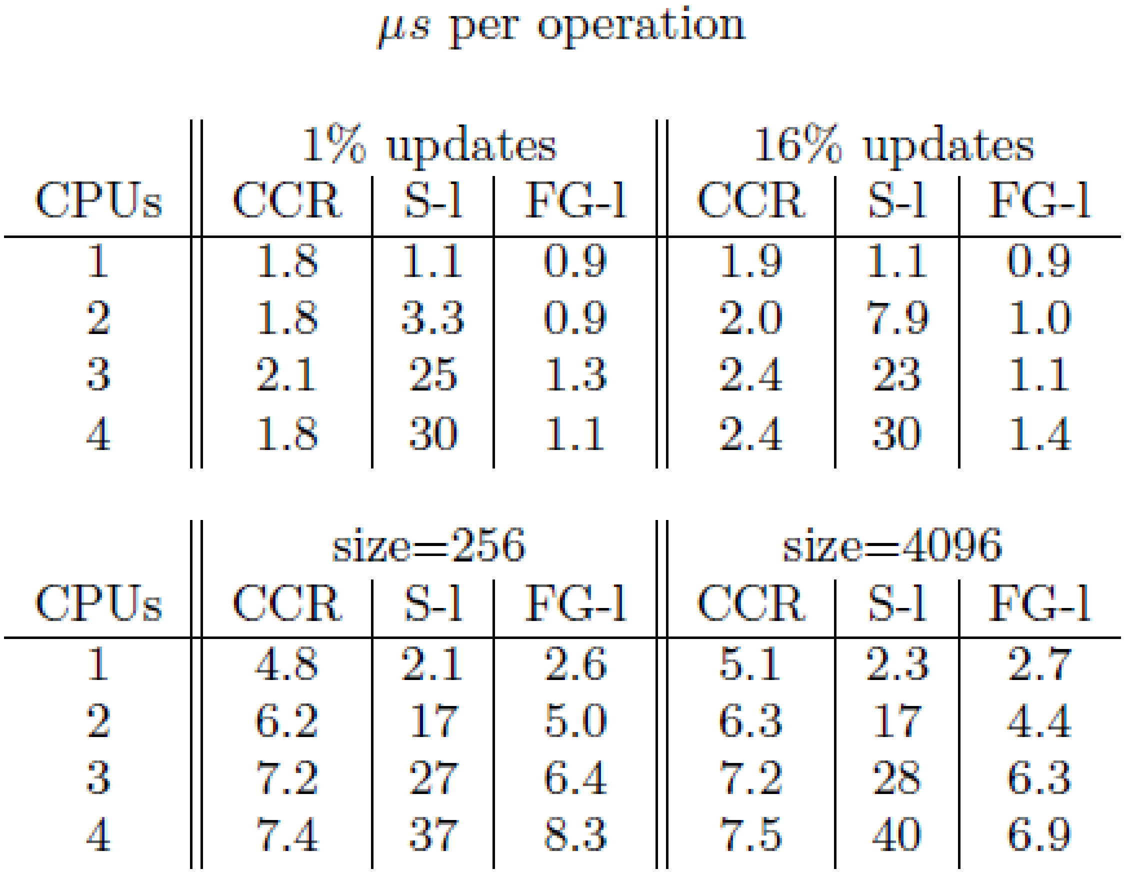
Implementation in JVM

- Memory management:
    - Descriptors allocated on garbage-collected heap,
    - Ownership records allocated statically

Performance

- Three testing set-ups:
  - **Hashtable** test – compares various implementations of concurrent hashtables:
    - Implementation from java.util library (with single mutex to protect the entire table),
    - Concurrent HashMap from util.concurrent package,
    - java.util.Hashtable with CCR without locks
  - **Compound** test – swapping values for two keys – combine update must be atomic,
  - **Wait** test – threads arranged in a ring with shared buffers

Performance – results for Hashtable test

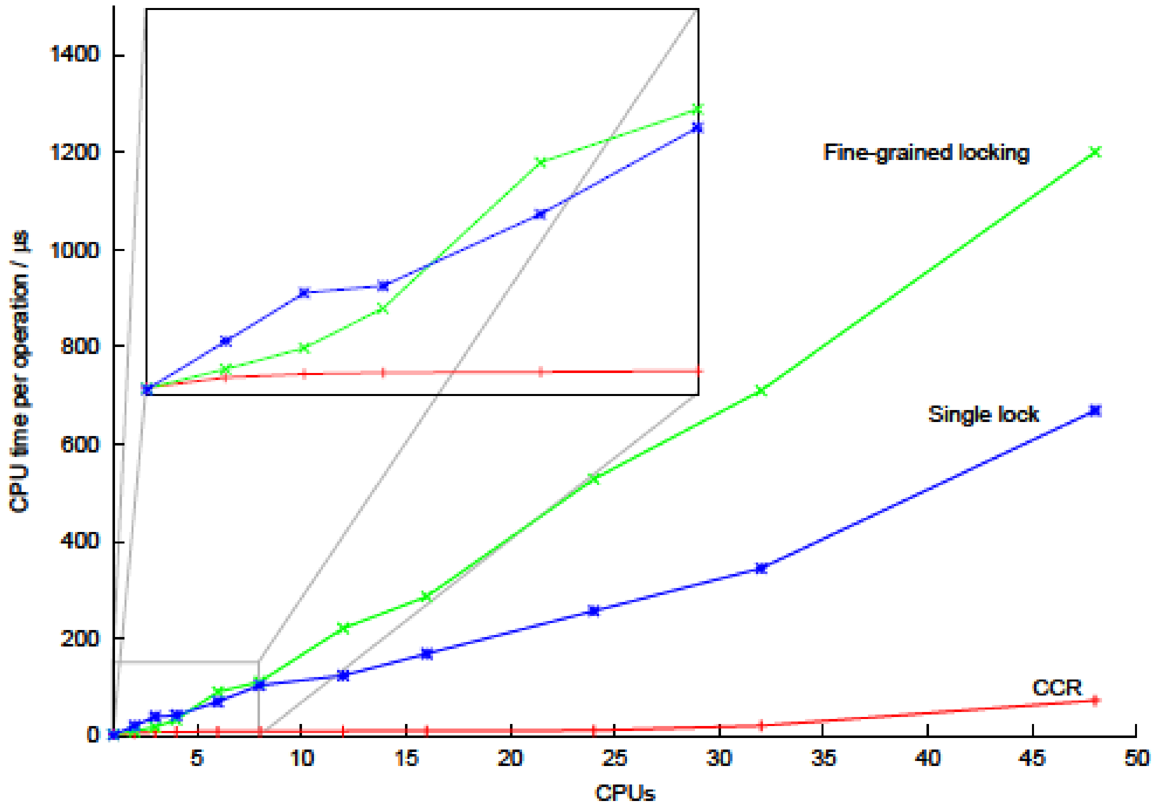


μs per operation

	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

Performance – results for Compound test



(*c*) Compound operations, 256-element table

Performance - summary

- Key feature – transactions which do not contend for the same ownership record can execute and commit in parallel,
- STM implementation works best when concurrent operations are likely to be *dynamically non-conflicting*

Future work

- Better benchmarking,
- Extended language-level interface,
- Hardware support
    - Hardware transactional memories

Thank you