

# Language Support for Lightweight Transactions

Tim Harris

University of Cambridge Computer Laboratory  
15 JJ Thomson Avenue  
Cambridge, UK, CB3 0FD  
tim.harris@cl.cam.ac.uk

Keir Fraser

University of Cambridge Computer Laboratory  
15 JJ Thomson Avenue  
Cambridge, UK, CB3 0FD  
keir.fraser@cl.cam.ac.uk

## ABSTRACT

Concurrent programming is notoriously difficult. Current abstractions are intricate to use and make it hard to design computer systems that are reliable and scalable. We argue in favour of a practical declarative style of concurrency control in which programmers directly indicate the safety properties that they require. In essence, the programmer demarks sections of code which execute within lightweight software-based transactions which commit atomically and exactly once.

These transactions can update shared data, instantiate objects, invoke library features and so on. They can also block, waiting for arbitrary boolean conditions to become true. These features support general-purpose shared data structures such as hashtables, queues and buffers. In general, no performance penalty is incurred for memory accesses outside transactions. Furthermore, transactions which do not access the same shared memory locations can commit concurrently.

We present a detailed design of this proposal along with an implementation and evaluation. We argue that the resulting system (*i*) is easier for mainstream programmers to use, (*ii*) prevents lock-based priority-inversion and deadlock problems and (*iii*) offers compelling performance.

## 1. INTRODUCTION

Mainstream object-oriented programming languages provide staid support for concurrency. Systems have gravitated toward providing multiple threads and using mutual-exclusion locks and condition variables to control access to shared data. These abstractions have many problems. For instance, consider implementing a shared buffer within an array. The core of a Java-style design could be:

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

The `synchronized` keyword means that a caller must obtain a mutual-exclusion lock (mutex) associated with the target object. The `wait` and `notifyAll` invocations are being used to block a thread which finds the buffer empty and to wake other threads (which may have blocked elsewhere

having found the buffer full). There are numerous difficulties here. Firstly, idioms such as the repeated `while` loop around the call to `wait` are often forgotten or mis-understood by novice programmers. Secondly, there is no check that the data accesses made are protected by the locks that are held. Thirdly, if a thread is pre-empted while holding the lock then no other thread can safely use the buffer. Finally, mutual-exclusion prevents `get` operations on a buffer proceeding concurrently with `put` operations, even if they do not conflict.

As a solution to these problems, we have returned to one of the oldest proposals for concurrency control – Hoare’s conditional critical regions (CCRs) [14]. In their most general style, CCRs allow programmers to indicate *what* groups of operations should be executed under mutual-exclusion rather than *how* to go about doing so with a particular scheme of locking. The programmer can also guard the region by an arbitrary boolean condition, with calling threads blocking until the guard is satisfied. The core of a shared buffer based on CCRs could look like:

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

This style closely expresses the safety properties which underlie the design. For this reason CCRs have long been popular in teaching concurrency and defining concurrent algorithms. Unfortunately, no good implementation technique has been known [2, 3]. The key problem is that the general form of CCRs gives no indication of what specific data items are to be accessed or, if a thread blocks at the guard, exactly when it may be released. Existing implementations allow only one CCR to execute at any time and re-evaluate every blocked CCR’s guard condition whenever any CCR completes. Unsurprisingly, performance is poor. To improve performance, mutual exclusion locks were introduced so that unrelated operations can execute concurrently and condition variables were introduced to control blocking and unblocking.

However, recent work on practical non-blocking concurrent data structures has led us to develop a new implementation technique [8]. We map CCRs down onto a *software transactional memory* (STM) which groups together series of memory accesses and performs them as-if atomically. We evaluate this technique under a number of different scenarios ranging from small multi-processors to a large server

with 106 CPUs. In our results, algorithms using CCRs can vastly out-perform those using simple mutual-exclusion. At all times they remain competitive with a well-engineered mutex-based scheme; under many workloads, our CCRs perform and scale better.

The paper makes three contributions:

- The implementation of CCRs is the first (*i*) to allow *dynamically non-conflicting* executions to operate concurrently, providing disjoint-access parallelism, (*ii*) to re-evaluate CCR conditions only when one of the shared variables involved has been updated, and (*iii*) to use a non-blocking implementation, preventing deadlock and priority inversion.
- This paper is the first to consider providing practical software transactions within a modern object-oriented programming language. Interesting problems arise over how to integrate transactional and non-transactional access to objects and how transactions interact with existing mechanisms for concurrency control and the memory consistency model.
- The STM developed is the first to allow word-size data to be held “in the clear” – that is, without reserving space in each word that may be accessed and without requiring the programmer to segregate objects which may be subject to transactional access. It is the first STM to consider synchronization between threads as well as simple atomic update.

After expounding on our motivation in Section 2, we survey related work in Section 3. Section 4 describes how we integrate CCRs in the Java programming language. Section 5 describes our software transactional memory. In Section 6 we introduce our current implementation and evaluate its performance. Section 7 concludes.

## 2. MOTIVATION

Concurrency is the norm in current computers, whether on large ccNUMA servers, on more modest symmetric shared-memory multi-processors, or even on single processors supporting simultaneous multi-threading. This provides a compelling need for practical mechanisms for controlling concurrency. We suspect that, although many programmers do not explicitly target parallel environments, this is in part due to the current complexity of doing so – new abstractions are needed if effective use is to be made of computer hardware.

Programmers using mutexes must decide what granularity of locking is appropriate. Protecting large data structures with a single lock aids implementation but reduces the parallelism that can be exploited. Using many smaller locks allows better parallelism, but leads to intricate code which spends much of its time juggling the locks. The optimal selection depends on the system’s workload, meaning that an informed decision is difficult in the general case.

Furthermore, current abstractions are entangled. Condition variables cannot be understood separately from mutexes; a “wait” on a condition variable must include an atomic “release” on a lock. Similarly, the semantics of a “notify” operation differ between systems – the programmer must know how many threads may (or must) be woken and how their scheduling interacts with that of the notifier.

Deadlock must be considered in systems with non-trivial locking. Unfortunately, preventing deadlock requires knowledge of the complete system and the execution paths through it. As well as deadlock, a programmer using priority-based scheduling must also understand priority inversion. Some cases can be handled by more sophisticated schedulers (e.g. priority inheritance), but others again require global knowledge (e.g. a priority ceiling protocol).

The root of these problems is the imperative style in which existing facilities for concurrency control are exposed to programmers. The resulting hand-compilation into these operations obfuscates the safety and progress properties that the programmer is actually trying to provide and commits the code, at the time when it is written, to following a particular locking discipline.

## 3. RELATED WORK

The JVM [17], Microsoft CLR [24] and POSIX pthreads APIs all provide mutexes and condition variables. These can either be used directly by programmers or can be built upon to provide higher level abstractions. For instance, the `util.concurrent` library being developed for future versions of Java<sup>1</sup> gives high-quality implementations of atomic variables, special-purpose locks, queues and thread pools.

Several languages have included CCR-style constructions. DP provides a style of guarded region defined by a `when` statement taking a series of boolean conditions and blocks of code; the statement blocks until one of the conditions is true and then executes the corresponding code [5]. Scheduling follows a co-routine model so no further support for mutual-exclusion is required. The Edison language provides an alternative `when` statement that acts as an atomic `if` [6]. Only one `when` statement may execute at any time. Lynx includes an `await` statement that allows a thread to block until a boolean expression becomes true [26]. It uses co-operative scheduling.

Rem shows how to build general CCRs over semaphores [2]. The design exhibits the classical problems: overly pessimistic concurrency and frequent expression re-evaluation. Schmid shows how static analysis can be used to avoid some re-evaluations [25]. His analysis is limited to expressions which are conditional on the values of statically allocated shared counters.

Argus has a style of transactions with an `enter` statement executing a body of code within a new transactional context [18]. An explicit `leave` allows transactions to be aborted. Nested transactions are permitted, but their definition is syntactically different from top-level ones, hindering code re-use.

There is a substantial body of work on theoretical models of concurrency, of which Hoare’s Communicating Sequential Processes [15] is a major starting point. The Pi Calculus introduces notions of communication channels as first class entities [22] and the Ambient Calculus introduces explicit notions of mobility and location [7]. The programming model differs greatly from that of object-oriented languages and there have been few successful attempts to integrate the two. Polyphonic C<sup>#</sup> is a notable current exception, although the emphasis there is on Internet programming and asynchronicity rather than general concurrency [4].

*Non-blocking* data structures have been studied as a way

<sup>1</sup>JSR166, <http://www.jcp.org/en/jsr/detail?id=166>

of sidestepping the problems caused by mutual-exclusion. A design is *non-blocking* if the failure of any (finite) number of threads cannot prevent the remainder of the system from making progress. This provides robustness against poor scheduling decisions as well as against arbitrary thread termination. It naturally precludes the use of locks because, unless a lock-holder continues to run, the lock can never be released.

Non-blocking designs can be classified according to the kind of progress guarantee that they make. In this work we focus on *obstruction-freedom*, a recent suggestion which is felt to make it easier to design efficient algorithms [11]. An obstruction-free algorithm guarantees that any thread can make progress so long as it does not contend with other threads for access to any location. This is strong enough to prevent deadlock and priority inversion, but requires an out-of-band mechanism to avoid livelock; exponential backoff is one option.

Building practical non-blocking algorithms directly from available hardware primitives is a very difficult task. For this reason, there is great interest in programming abstractions which make it easier to create non-blocking systems. One promising example is *transactional memory* [13], which allows memory accesses to be grouped into transactions which either commit, becoming globally visible at the same instant in time, or abort without being observed.

Although the original proposal suggested hardware support, Shavit and Touitou show how a similar design can be provided by software [27]. However, their design has two practical limitations: (i) it can only be applied to static transactions, whose data sets and operations are known in advance; and (ii) it requires a strong atomic primitive which is not provided by any processor architecture. Although mechanisms exist to build these stronger primitives, they involve reserving space in each word that may be accessed, so full word-size values can only be manipulated by fragmenting them across multiple locations with an attendant space cost [23].

More recently, Herlihy *et al* [10] have designed a practical software transactional memory which is obstruction-free and requires only the readily-available double-width compare-and-swap (CAS) instruction. Unlike the design in Section 5, it introduces per-object indirection and an explicit ‘open’ step when accessing an object in a transaction. This may not allow as direct an implementation as we describe. However, it does allow a more streamlined non-blocking commit and, aside from the ‘open’ operation, transactional access proceeds in the same way as ordinary access.

## 4. LANGUAGE INTEGRATION

Many questions arise when considering how CCRs could be integrated into an existing language. 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 interoperate with existing features for concurrency control?

With such questions in mind, our design is motivated by two principles. Firstly, CCRs should be able to enclose code with as few restrictions as possible. This encourages code reuse by allowing single-threaded libraries to be made thread-safe by wrapping each invocation with a CCR. The second principle is that the system should permit an implementa-

tion which does not impose a high overhead in parts of an application where CCRs are not used – it would be unfortunate for the implementation to mandate an extra field for every object or to complicate field accesses outside CCRs.

In this section we describe the high-level aspects of our design, showing how we integrate CCRs with the Java Programming Language. We will then turn to our implementation of these over a software transactional memory in Section 5.

### 4.1 Identifying CCRs

Our basic syntax of

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

defines a CCR which waits (if necessary) until `condition` is true and then executes `statements`. All other threads observe the CCR to take place atomically at some point between its start and its completion, so long as they follow the memory consistency rules in Section 4.7. The thread executing the CCR sees the updates it makes proceed according to the usual single-threaded semantics.

The `condition` may be omitted if it is simply “true” and `atomically` can also be applied as a method-modifier in place of `synchronized`.

We provide exactly-once execution of the statements. We did consider offering at-most-once semantics and using non-execution as an indicator of contention. We concluded that this may aid expert programmers, but most uses of CCRs would then require external looping to retry until an operation succeeds. We also considered timeouts but again, for simplicity, do not currently provide them – a thread blocked on a CCR’s guard can be ‘interrupted’ in the same way as a Java thread blocked on a condition variable.

Exceptions can be thrown from within a CCR to outside it. This is consistent with our first design principle of allowing their use as wrappers around existing single-threaded code.

### 4.2 Data accessible to CCRs

The principle of allowing code re-use suggests that we should not need to indicate through the class hierarchy which objects may be accessed within CCRs – for example by requiring them to extend a designated superclass. Doing so would require library classes to be re-implemented before they could be used.

This fact, along with our principle of avoiding overhead outside CCRs, led to the development of our word-based software transactional memory which aids the sharing of memory locations between transactional and non-transactional accesses.

### 4.3 Native methods

It is not practicable, in general, to allow native methods to execute within CCRs – native code containing arbitrary memory accesses and system calls raises the same problems that have hitherto made it difficult to provide CCRs. Native methods include all those within the standard libraries for performing I/O operations.

Our current design generally raises a runtime exception if a native method is invoked within a CCR. However, we treat a number of built-in native methods as special cases

in which their behavior is either thread local (for instance cloning an object or computing an object’s identity hash value) or relates to synchronization and therefore requires special handling.

#### 4.4 Nested CCRs

If CCRs are nested dynamically then the entire assembly executes as-if atomically at a time satisfying all of the conditions. Of course, the programmer is responsible for ensuring that such a time will exist; for instance, code containing

```

atomically (x == 1) {
  atomically (x == 0) {
    ...
  }
}

```

will never complete.

#### 4.5 Existing synchronization mechanisms

Allowing code re-use requires us to consider how concurrency control within CCRs interacts with the existing mechanisms of mutual-exclusion locks and condition variables; what does it mean for a block of code that manipulates locks or communicates through `wait` and `notify` operations to execute as-if atomically?

##### 4.5.1 Mutual-exclusion locks

If a CCR attempts to acquire mutexes then the system ensures that all of these are available at the point at which it appears to atomically take effect. This means that mutexes can be used to safely share data between access within CCRs and external access. Consequently, if the CCR implementation is non-blocking, then it precludes the risk of deadlock in code such as

```

atomically {
  synchronized (a) {
    synchronized (b) {
      ...
    }
  }
}

atomically {
  synchronized (b) {
    synchronized (a) {
      ...
    }
  }
}

```

We return to some further consequences of using locks when discussing the consistency model in Section 4.7.

##### 4.5.2 Condition variables

It is not possible to ascribe useful semantics to a `wait` operation on a condition variable within a CCR. The `wait` operation always blocks until selected for resumption by a notification. This makes it impossible to identify a single point at which the entire CCR executes as-if atomically. For symmetry we also forbid notification within CCRs, although there is no implementation impediment to allowing it. In both cases this is consistent with the fact that in Java the operations on condition variables, `wait`, `notify` and `notifyAll` are defined as native methods.

#### 4.6 Class loading

The Java programming language defines times at which new classes should be loaded and their initialization code be executed [16]. What happens if class loading or initialization is attempted by the JVM while executing within a CCR? There are two options:

The first is to consider that such operations are performed in the context of the CCR that causes them to happen; the class would appear to be loaded or initialized at the same time as the `atomically` block appears to occur. This raises difficult conceptual problems from a programmer’s viewpoint; for instance classes whose initialization involves the creation of a thread or calls to native methods.

The second option, which we select, is to dissociate class loading and initialization from the point within a CCR’s execution which triggers it. We require class loading and initialization to occur at some point between when a CCR begins and the point at which it appears to take place atomically.

#### 4.7 Consistency model

The final decision to highlight is the relationship between CCRs and the proposed Java memory model [19]. Memory consistency is particularly interesting in Java, because of the reliance placed on type-safety and because of the ongoing work to provide a formal definition.

In outline, the memory model defines safety properties which cannot be violated no matter what the application program does – for instance, if objects are implemented using a table of virtual methods, then that table and the references to it must have been initialized before it is used in method invocation. Beyond these properties, the model sets out a practical notion of “correctly synchronized” programs to define what is necessary for a memory access made in one thread to be visible to another. We elide some details, but broadly the requirement is that a memory update must be visible if the writer releases a lock subsequently acquired by the reader, or if the writer updates a `volatile` field subsequently read by the reader.

We extend this to CCRs in the natural way, placing the same onus on the programmer in order to achieve the intended atomicity if data is shared between different kinds of concurrency control. An ordering is induced between any CCRs that access common memory locations, between any CCR that accesses a `volatile` field and any other access to that field, and finally between CCRs that hold a mutex and other acquire/release operations on that lock.

We intend to formalize this model in future work. However, we believe it leads to the same kind engineering rule as currently proposed: if a location is shared between threads then accesses to it must (in any combination) (i) be protected by mutexes, (ii) be protected by CCRs, or (iii) be marked as `volatile`.

### 5. SOFTWARE TRANSACTIONS

We now turn to the STM we have developed as the basis for our implementation of CCRs. We present it here in terms of a software system but a hardware-based transactional memory could be used if available. We return to the question of hardware support when discussing future work, but for the moment we assume only that word-sized memory accesses are atomic and that a word-sized atomic compare and swap (CAS) instruction is available. This instruction, or an equivalent, is available in all major architectures.

As with existing STMs, our design groups memory accesses into transactions and performs these as-if atomically. The STM has a number of notable features which stem from our requirements:

```

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

```

Figure 1: A CCR of the most general form atomically (condition) { statements; } expressed in terms of transaction start/commit operations, assuming that done is an otherwise unused identifier.

- No reserved space is needed in the locations being accessed. This means that, in Java, fields can hold full 32-bit integers without additional per-field storage.
- It requires only word-sized updates to be atomic when accessing heap locations. Supporting double-word data types poses no problem.
- The permanent structure used to co-ordinate transactions can be statically allocated outside the application heap. We can trade off the likelihood that non-conflicting transactions commit in parallel against the size of the structure. Temporary data structures can be allocated from the same heap used by the JVM, or instead be held separately.
- 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. This is important to ensure effective caching [9].

We will introduce the interface exported by the STM in Section 5.1 and then describe the implementation in Sections 5.2-5.4.

## 5.1 STM interface

Our STM provides operations for non-nesting transactions accessing memory locations on a word-addressed basis. We define four operations for transaction management:

### Transaction management

```

void STMStart()
void STMAbort()
boolean STMCommit()
void STMWait()

```

STMStart begins a new transaction within the executing thread. It is an error to invoke STMStart if the current thread is already running a transaction. Similarly, it is an

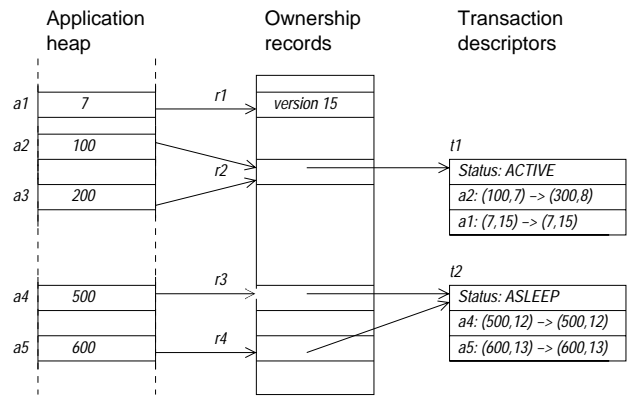


Figure 2: The STM heap structure. Two transactions are in progress here; t1 is attempting to commit accesses made to addresses a1 and a2 (checking a1 holds 7 and updating a2 from 100 to 300). The second transaction, t2, is asleep waiting for updates to either a4 (where it read 500) or to a5 (where it read 600).

error to invoke the other operations unless there is a current transaction. STMAbort aborts the transaction in progress by the executing thread. STMCommit attempts to commit the transaction in progress by the executing thread, returning true if this succeeds and false if it fails.

STMWait starts by validating the transaction so far – that is, checking that the values seen represent a current and consistent snapshot. If this validation succeeds then the thread is blocked atomically with taking the snapshot. Otherwise the thread continues. A thread blocked in STMWait is woken when another thread commits an update to a location that the waiter has accessed. The current transaction is always aborted before STMWait returns.

Figure 1 summarises how a non-nesting atomically block may be expressed in terms of these explicit transaction management operations. Note how the current transaction must attempt to commit both for normal termination of the statements and also for termination by throwing an exception. Nesting CCRs are implemented within the same transaction, counting the dynamic nesting depth and only invoking STMCommit when the top-level completes; they do not require support here.

The second set of operations exposed by the STM are for performing memory accesses:

### Memory accesses

```

stm_word STMRead(addr a)
void STMWrite(addr a, stm_word w)

```

Again, these may only be used while a transaction is active. In our implementation an address of type addr is an ordinary pointer into the heap and an stm\_word is simply an integer of the machine word size.

## 5.2 Heap structure

Our STM uses three kinds of data structure, as indicated in Figure 2. The first is the application heap in which the data itself is held. For instance, in the JVM, this would be

the objects allocated by the application, held in their usual format and including any header fields needed.

The second kind of structure are the *ownership records* (orecs) which are used to co-ordinate transactions. An *ownership function* maps each address in the application heap to an associated orec. There need not be a one-to-one correspondence between addresses and records – there could be one orec per object or, as in our prototype, a fixed-size table of records to which addresses map under a hash function. Each orec holds either a *version number* or a *current owner* for the addresses that associate with it. Each time a location in the application heap is updated, the version number must be incremented. Version numbers will be used to detect whether a transaction may be committed. We assume for the moment that they are never re-used in the same orec; we return to this when discussing our implementation.

The final kind of structure holds *transaction descriptors* which set out the current status of each active transaction and the accesses that it has made to the application heap. Each access is described by a *transaction entry* specifying the address in question, the old and new values to be held there and the old and new version numbers of those values. The status field indicates that the transaction is either **ACTIVE** (able to have **STMAbort**, **STMWait**, **STMCommit**, **STMRead** and **STMWrite** operations invoked for it), **COMMITTED**, **ABORTED** or **ASLEEP**. Descriptors are initially **ACTIVE** and move through the other states while attempting to commit, to abort or to wait.

A descriptor is *well formed* if for each ownership record it either (i) contains at most one entry associated with that orec, or (ii) contains multiple entries associated with that orec, but the old version number is the same in all of them and the new version number is the same in all of them. As with version numbers, we assume that descriptors are never re-used; we again return to this for our implementation.

Given this structure, we introduce the concept of the *logical state* of an address in the application heap. This is the pair of the value conceptually held at that address and the version number associated with that value being there. We define the logical state by a disjunction of three cases. In the first case the orec contains a version number:

LS1 The version number is taken from the orec and the value is held directly in the application heap. For instance, in Figure 2, the logical state of **a1** is (7, 15).

In the second and third cases the orec refers to a descriptor:

LS2 If the descriptor contains an entry for the address then that entry gives the logical state. For instance, the logical state of **a2** is (100, 7) because the descriptor shown has not yet committed and it holds an entry updating **a2** from (100,7) to (300,8).

LS3 If the descriptor does not contain an entry for the address, then the descriptor is searched for entries about other addresses which map to the same orec as the requested address. The value is taken from the application heap and the version is taken from the entry; the new version number if the transaction is **COMMITTED** and the old version number otherwise. The ‘well formed’ property ensures that this is uniquely determined. For instance, the logical state of **a3** is (200, 7) taking old version 7 from the entry for **a2**.

At run time, the logical state of an address can be determined from a consistent snapshot of the locations on which its value depends: the address itself, its orec, the status of an owning descriptor and information from entries in that descriptor.

Fortunately, a general-purpose snapshot algorithm is not necessary here and we can directly compute the logical state by reading locations as described in the three cases LS1..LS3. The non-re-use of descriptors and version numbers lets us employ a simple re-read-then-check design, re-computing the logical state if the orec’s value changes part-way through:

```
do {
  orec = orec_of (addr);
  <directly compute logical state based on orec>
} while (orec_of (addr) != orec);
```

As we shall see in the implementation of the STM operations, if the orec’s value is unchanged then the derived logical state is consistent with a snapshot of the locations involved. For LS1 the value is read from the application heap – it cannot have changed if the orec did not. For LS2 and LS3, the locations accessed in descriptor entries relating to an orec are constant once the pointer is installed as that record’s owner. The only other location involved – the descriptor’s status – can change exactly once from **ACTIVE** to one of the other states. The snapshot is consistent with the time when the status is read.

## 5.3 STM operations

We will now describe the implementation of the STM operations using this heap structure. In outline, orecs ordinarily hold version numbers, as **r1** does in Figure 2. An orec only refers to a descriptor when that transaction is attempting to commit or to sleep – until **STMCommit** or **STMWait** is invoked the transaction execution is private, building up a series of entries in the descriptor which set out the locations that it has accessed.

### 5.3.1 STMStartTransaction

Starting a transaction allocates a fresh descriptor and initializes its status field to **ACTIVE**.

### 5.3.2 STMAbortTransaction

Aborting a transaction writes **ABORTED** into its status field.

### 5.3.3 STMReadValue

To read a value we must consider two cases. Firstly, if the current descriptor already contains an entry (**te**) for the requested location then return **te.new\_value**. Otherwise determine the logical state of the location and initialize a new entry with the value seen as **old\_value** and as **new\_value**. Record the version seen as **old\_version**. If the descriptor already contains an entry for this orec then copy that entry’s new version number to this entry in order to keep the descriptor well formed, otherwise use **old\_version**. Although we describe these operations in terms of searching and copying, this can be streamlined in implementation as we shall describe in Section 5.4.

### 5.3.4 STMWriteValue

The implementation of **STMWriteValue** first ensures that the descriptor contains an entry (**te**) relating to the location being accessed. This can be done by performing a read operation from the location. Set **te.new\_value** to the value being

written and set `te.new_version` to `te.old_version+1`, copying this new version number to any other entries relating to the same orec so that the descriptor remains well formed.

### 5.3.5 STMCommit

Commit proceeds by temporarily *acquiring* each of the ownership records it needs, then – if it can acquire them all – it updates the descriptor’s status field from `ACTIVE` to `COMMITTED`, makes any updates to the application heap and then proceeds to *release* each of the ownership records. The key to all of these operations is that logical states are only updated when the status field is changed.

The details lie in how the *acquire* and *release* steps are implemented and, in particular, what happens when one transaction wishes to acquire an orec that is already held. Both take as parameters the descriptor in question (`td`) and an index into that descriptor’s table of transaction entries.

```
acquire (transaction_descriptor *td, int i) {
    transaction_entry te = td.entries[i];
    orec seen;
    seen = CAS (orec_of(te.addr),
               te.old_version, td); /*C1*/
    if (seen == te.old_version || seen == td) {
        return TRUE; /*1*/
    } else if (holds_version_number (seen)) {
        return FALSE; /*2*/
    } else {
        return BUSY; /*3*/
    }
}
```

This attempts to install `td` in the ownership record associated with the selected transaction entry. It may only be called for a descriptor that is `ACTIVE` or `ABORTED`. The only possible update to shared memory is at `/*C1*/`. If this succeeds then it preserves the logical contents of that location (case `LS2`); it also preserves the logical contents of any other locations which alias to the same ownership record (case `LS3` and well-formedness).

In return case `/*1*/` either `/*C1*/` succeeded, or transaction `td` already held the orec. In return case `/*2*/` the orec contained a version number other than the one expected by this transaction: the transaction is doomed to fail. In return case `/*3*/` the orec is discovered to be owned already and cannot be acquired.

Given this operation, `STMCommit` proceeds by invoking `acquire` for each entry in turn. If any invocation returns `FALSE` then the logical contents of that location were not consistent with the version expected in the entry; the commit has failed and the status is updated to `ABORTED` and `release` invoked for any entries successfully acquired. If any invocation returns `BUSY` then the transaction has encountered another active on the same orec. A simple reaction is to (i) abort the existing owner, (ii) wake it if it was blocked in `STMWait` and (iii) abort the current transaction and leave it to retry (hopefully when the existing owner has completed its own operation and relinquished ownership). We discuss alternative non-blocking strategies in Section 5.4.

If all locations can be acquired, the descriptor’s status field is updated to `COMMITTED`. This has the effect of atomically updating the logical state of all of the locations – at that point the descriptor is referred to by all of the ownership records relating to locations it has acted on. The transaction

then writes the new values to the application heap – note again that concurrent operations, when determining the values held at those locations would use the versions held in the descriptor; they will be unaware that these writes themselves occur at different times to different locations.

Finally, once it has made all of the writes, the transaction invokes `release` for each entry in turn. This attempts to remove a reference to a descriptor from an ownership record:

```
release (transaction_descriptor *td, int i) {
    transaction_entry te = td.entries[i];
    if (td.status == COMMITTED) {
        CAS (orec_of(te.addr),
            td, te.new_version); /*C2*/
    } else {
        CAS (orec_of(te.addr),
            td, te.old_version); /*C3*/
    }
}
```

Again, note that it preserves the logical contents of all locations associated with the orec so long as the descriptor’s state is `COMMITTED` or `ABORTED` and the locations have been updated as necessary.

### 5.3.6 STMWait

The last function to consider is the `STMWait` operation which causes the thread executing the current transaction to validate the accesses it has performed thus far and then to either abort directly (if they are not valid) or to block until an update is made to any of the locations concerned, again before aborting.

The implementation initially proceeds as with `STMCommit` in attempting to acquire all of the orecs relating to the transaction. If successful, this confirms that the memory accesses made so far are valid; the thread sleeps, settings its status field to `ASLEEP` and leaving references to its descriptor installed at the acquired orecs. These will act as “software tripwires” and signal the presence of the sleeper to any other transaction which attempts to commit an update to those locations; its acquire operation will return `BUSY` and it can wake the sleeper.

The simplest design is to provide `sleep` and `wake` as operations built over per-transaction mutexes and condition variables:

```
sleep (transaction_descriptor td) {
    mutex_lock (td.mutex);
    CAS (&td.status, COMMITTING, ASLEEP);
    if (td.status == ASLEEP) {
        condvar_wait (td.cv, td.mutex);
    }
    mutex_unlock (td.mutex);
}

wake (transaction_descriptor td) {
    mutex_lock (td.mutex);
    if (td.status == ASLEEP) {
        CAS (&td.status, ASLEEP, ABORTED);
        condvar_signal (td.cv);
    }
    mutex_unlock (td.mutex);
}
```

Figure 2 illustrates this situation, showing a transaction  $t_2$  whose thread is asleep waiting for updates to locations associated with orecs  $r_3$  and  $r_4$ . Although we do not describe it here, `wake` could be made non-blocking by using a counting semaphore in place of mutexes and condition variables.

## 5.4 Optimizations

The basic design in Section 5.3 provides safe concurrent transactions. However, as presented, it has several infelicities which would limit its practical performance – (i) at most one thread can sleep on any particular ownership record, (ii) both read and write operations involve searching the current descriptor for entries associated with a particular orec, (iii) processing an entry describing a read-only access to a location still involves updating its orec twice, harming data-cache performance, and (iv) the simple retry operation prompted by a `BUSY` return value prevents the design from being non-blocking.

We now introduce remedies to these problems. We present these separately to avoid cluttering the main design.

### 5.4.1 Multiple sleeping threads

In the basic design, both `STMCommit` and `STMWait` respond to a `BUSY` signal by waking the thread currently holding the orec if it is asleep. This means that at most one transaction can be asleep associated with any one orec. To allow multiple threads to sleep on the same location we extend each descriptor to include a list of other threads which wish to sleep on locations acquired by the descriptor. The list is protected by the same mutex used to control sleeping and waking.

### 5.4.2 Read sharing

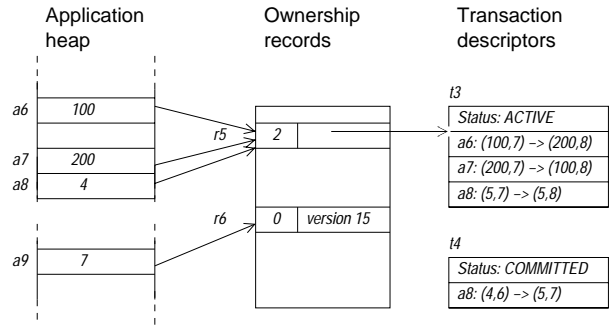
Modern cache coherence protocols allow multiple CPUs to concurrently hold the same cache block, so long as they do not attempt to write to it [9]. This makes it important to avoid contended writes to shared locations. Our basic design risks such writes to the locations that hold ownership records; an `STMCommit` operation must acquire and then release each of these records even when the underlying access in the transaction was a read.

We can modify `STMCommit` to introduce an additional phase to deal with read-only locations. It is tempting to do this by simply (i) acquiring the locations subject to updates and then (ii) checking the logical state of the locations which are just being read against the `old_value` and `old_version` in the descriptor before (iii) attempting to update the transaction’s state to `COMMITTED`. However, this does not provide atomicity: another transaction may update the locations being read between the second and third steps.

To prevent this problem we introduce a new transaction state, `READ_PHASE`, which is entered before checking read-only locations and therefore held throughout those checks up to the point at which the transaction commits or aborts. If another transaction encounters one in `READ_PHASE` then it causes the one encountered to abort. In practice the read phase is so short that it is not observed by other transactions under any non-synthetic workloads we have found.

### 5.4.3 Avoiding searching

We observe that many transactions exhibit temporal locality in the locations that they access. We can exploit this by maintaining, for each thread, a table mapping from owner-



**Figure 3: Ownership records extended to allow obstruction-free `STMCommit` operations. In this case transaction  $t_3$ ’s descriptor has stolen ownership of  $r_5$  from the lower descriptor. It now includes updates from both descriptors, even though  $t_4$ ’s write to  $a_8$  has not yet been performed to the application heap.**

ship records to entries in the thread’s current descriptor. In particular, this streamlines `STMWrite` operations which follow earlier transactional reads – the cache directly identifies the entry to update. A designated value indicates if there are multiple entries relating to the same ownership record, in which case a search of the descriptor is unavoidable.

### 5.4.4 Non-blocking commit

The final and most intricate development to describe is how to make the `STMCommit` operation non-blocking – currently, if a thread encounters an ownership record that is already held, it must wait for the current holder to release it. In some settings this may be an academic concern rather than a practical one; if the holder is making good progress then “helping” them complete their operation will do nothing but harm caching. However, in other settings, the stronger guarantees of non-blocking algorithms are desirable for the reasons given in Section 2.

Our approach is to permit one thread to *steal* ownership by using an atomic compare-and-swap on the orec. There are two problems in allowing this:

- Firstly, we must ensure that the theft does not change the logical contents of any location.
- Secondly, if a previous owner has `COMMITTED` but not yet written to the underlying locations in the application heap, there is no control over when those writes occur. We must ensure that the writes made by the new owner succeed those made by the previous owner.

Both of these problems have solutions. We will illustrate our solutions with respect to Figure 3. To ensure that the logical contents of locations are not updated, a transaction such as  $t_3$  that steals ownership of an orec from another ( $t_4$ ) must merge entries relating to the orec from  $t_4$ ’s descriptor into its own. We cannot merge from an `ACTIVE` transaction in case it commits while we are doing so; we would then risk changing the logical state of addresses when stealing. This is avoided by aborting the victim if it has not already committed. The stealer takes the `old_value` and `old_version` if the victim is



aborted and `new_value` and `new_version` if it has committed. In the figure, `t3` has merged the entry for address `a8`.

The second problem means that we cannot release ownership of a location until we can guarantee that (i) there are no transactions still making writes to the location, and (ii) the final value written relates to the most recent transaction. We deal with this by introducing a counter into each orec saying how many transactions are in the process of making updates to the locations it manages. In the figure `r5` has two transactions making writes to its locations (`t3` and `t4`) and `r6` has none because it is not owned. When stealing, the counter is incremented atomically with updating the owner. When releasing ownership, the counter is decremented, either leaving the owner unchanged (if the counter will remain above zero), or restoring the version number (if the counter becomes zero). If a thread discovers that ownership has been stolen from it (because it sees a different descriptor in the orec) then it re-does the updates made by the *new* owner, ensuring that the final value written before releasing ownership is that of the most recent transaction.

## 6. IMPLEMENTATION AND EVALUATION

Our implementation is based on version 1.2.2 of the Sun Java Virtual Machine for Research. This JVM implementation has already undergone extensive optimization; we are comparing our prototype against a best-of-breed system [1].

### 6.1 Modifications to the JVM and compilers

The implementation is split between compile-time support in the source-to-bytecode compiler and run-time support in the bytecode-to-native compiler. The intermediate `.class` file format is unchanged. At the bytecode level we implement `atomically` blocks only on a per-method basis, signalling them to the run-time system by suffices to the method's name. If an `atomically` block is defined within a method then the source-to-bytecode compiler extracts it into a separate method. In this way the implementation of transactions can directly access local variables without needing to keep undo information in case of abort and therefore without needing to use the transactional operations.

The run-time compiler produces specialised code for transactional methods, generating calls to `STMRead` and `STMWrite` for bytecodes which access heap locations. Each class has a second method table holding references to transactional versions of its methods (compiled on demand). This table is used by method invocations within transactional methods.

The run-time compiler generates specialised code for accessing `volatile` fields outside CCRs. For the moment they are translated as “small” transactions performing a single read or write as appropriate and looping until they commit successfully. This provides the ordering required by the memory model in Section 4.7. Access to other shared fields is unordered and is implemented directly by memory reads or writes to the application heap.

### 6.2 Memory management

The design presented here has assumed that descriptors are managed through a garbage collector. While we could simply allocate them on the garbage-collected heap, our implementation provides build-time options to allow re-use where possible. The options include simple reference counting [21] and the more recent designs due to Michael [20] and Herlihy *et al* [12].

In our experiments we allocate descriptors on the garbage-collected heap and use the *pass the buck* algorithm for re-use between collections [12], holding re-usable descriptors in thread-local pools. Descriptor allocation and deallocation form less than 2% of the time spent within the STM implementation.

Ownership records are statically allocated. Our experiments used a table of 65 536 records, indexed by bits 2...18 of a location's address. We tested our implementation for sensitivity to the number of orecs. So long as aliasing of different locations to the same orec is rare, the precise number has only a marginal effect on performance – less than 1% going from 4 096 to 65 536 in our experiments.

### 6.3 Version numbers

We represent version numbers as odd integer values, allowing us to distinguish them in the ownership records from aligned pointers to descriptors. We do not consider the possibility of overflowing the remaining 31 bits: a simple scheme would be to periodically suspend all threads, abort any active transactions and reset the version numbers to 1. Such a brief ‘stop-the-world’ situation already exists in the garbage collector. We use a double-word-width CAS to maintain counters in the non-blocking commit operation.

### 6.4 Performance

Of course, key to good performance on any shared-memory multiprocessor is avoiding contention for cache blocks. For application data this is the responsibility of the programmer, whether using locks or whether using CCRs.

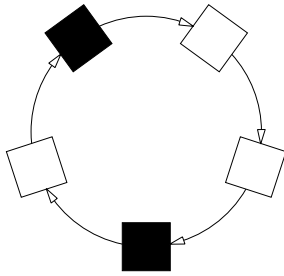
Our implementation makes a substantial separation between common-case code and the more involved aspects of our design. For instance, there is an ‘optimistic’ commit that assumes the contention is rare and executes an out-of-band non-blocking commit only if another descriptor is encountered; the code was invoked on fewer than 1% of commit operations in our experiments. Remember that even long-running transactions remain private until they start to commit.

Where a transaction commits without contention, reading from  $r$  locations and updating  $w$  locations involves  $w$  CAS operations to acquire ownership,  $r$  reads to check read-only locations, 2 updates to the status field,  $w$  writes to the application heap and then  $w$  CAS operations to release ownership.

### 6.5 Experimental set-up

We will present results from three different experimental settings:

- **Hashtable** compares various implementations of concurrent hash tables. The first of these uses the hashtable implementation from the JDK 1.2.2 `java.util` library which uses a single mutex to protect the entire table. The second implementation uses `ConcurrentHashMap` taken from the `util.concurrent` package (release 1.3.2). This is a carefully engineered design which allows most read-only operations to proceed without locking and often allows non-conflicting updates to proceed concurrently. The third implementation uses the same underlying simplistic design as `java.util.Hashtable` but wraps each access in a CCR without using any locks. In each case the table contains 4096 mappings and we perform a mix of  $p\%$  reads and  $(100 - p)\%$



**Figure 4: Experimental configuration for the Wait test.**  $t$  threads are conceptually arranged in a ring with a shared buffer between each. Initially  $n$  of these buffers contain tokens and the others are empty. In this case  $t = 5$  and  $n = 2$ .

updates. Operations are uniformly distributed across the keys.

- **Compound** compares operations involving several accesses to a hashtable. Each compound operation selects (uniformly) two keys and then swaps the values that they map to – the combined update must be atomic. We implement this either (i) with a single mutex, (ii) using per-key locks and `ConcurrentHashMap` or (iii) using an `atomically` block. We can vary the size of the table in order to vary the likelihood of contention.
- **Wait** evaluates blocking operations. The experimental configuration has  $t$  threads conceptually arranged in a ring with a shared buffer between each adjacent pair. Of these buffers, initially  $n$  contain tokens and the remainder are empty. Each thread loops removing an item from the buffer on its right and placing it in the buffer on its left. Figure 4 illustrates this configuration. We compare an implementation based on CCRs against one built using the mutexes and condition variables provided by the JVM.

In each case we ran tests for three wall-time seconds and took the median of 5 runs. In Section 6.6 we will use an entry-level symmetric shared-memory system and then in Section 6.7 we will use a larger ccNUMA sever.

Aside from trivial single-threaded cases, our results for a simple implementation using CCRs out-perform an equivalent implementation using locks. Our implementation could be extended to handle single-threaded execution as a special case. In every case our simple CCR-based implementation remains competitive with well-engineered locking; in some cases it performs even better.

## 6.6 Small systems

Our first set of measurements are from a 4-processor Sun-Fire v480. Figure 5 compares the performance of CCRs against lock-based implementations of the **hashtable** and **compound** tests.

In the individual operations of the **hashtable** test the solution based on fine-grained locking in `ConcurrentHashMap` performs best of all, sometimes by a factor of just over 2. However, the implementation of that class is much more involved than that of our simple comparison using CCRs.

$\mu s$  per operation

CPUs	1% updates			16% updates		
	CCR	S-1	FG-1	CCR	S-1	FG-1
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

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

Our benchmark accentuates the performance difference by attempting operations as frequently as possible; any real application would have less than a 100% duty cycle.

In the **compound** test the CCR-based solution continues to outperform the basic lock-based scheme in all but single-threaded use. Furthermore, the difference in performance between it and `ConcurrentHashMap` reduces and is eventually reversed under higher contention. Of course, using our CCR implementation brings the attendant benefits of its lock-free design.

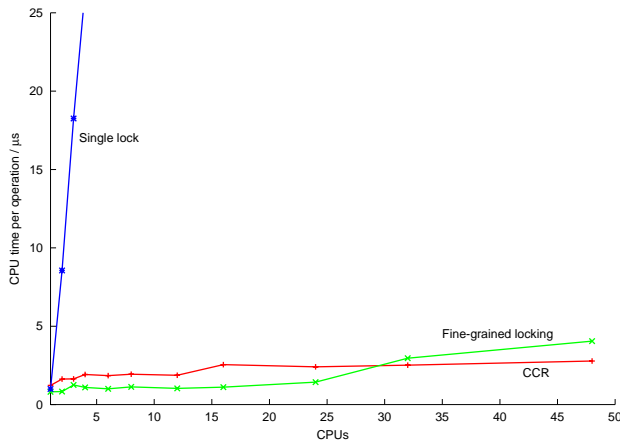
Finally, in the **wait** test measuring the throughput of blocking operations, the implementation using CCRs operated at 80% of the lock-based rate when using 2-4 threads, irrespective of the number of tokens circulating.

## 6.7 Large systems

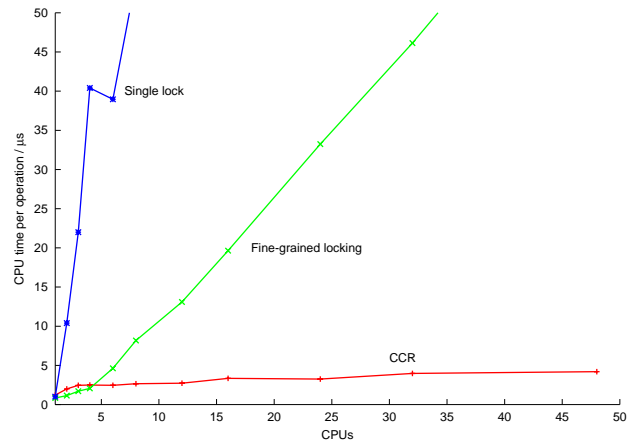
Our second experiments use a 106-processor ccNUMA Sun-Fire e15k machine. Figure 6 shows the results for 1.48 processors running the **hashtable** and **compound** tests. For the **hashtable** tests the performance of the lock-based system is deleterious on multi-threaded workloads. When write-contention is low, the `ConcurrentHashMap` design performs best; this is to be expected because it avoids the transaction management overheads of the STM. However, as write-contention rises, so does contention for locks and the STM-based design performs best.

Of course, it may be possible to specialise the lock-based designs in order to improve performance for this workload. However, the point we emphasise is that under *every* workload here, the design using CCRs around a naive hashtable has performance that is comparable with these other techniques. Furthermore, aside from its performance and programming ease, its non-blocking guarantees are automatically providing robustness against deadlock and priority inversion.

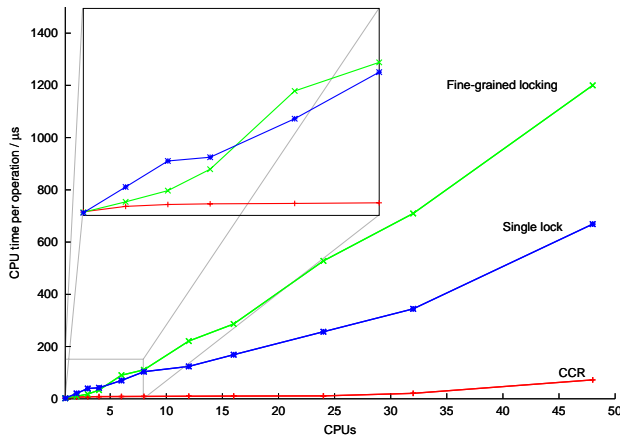
For the **compound** tests the CCR-based designs are fastest in every setting with more than 1 CPU (in the single-threaded case it is 15% slower than `ConcurrentHashMap` and 36%



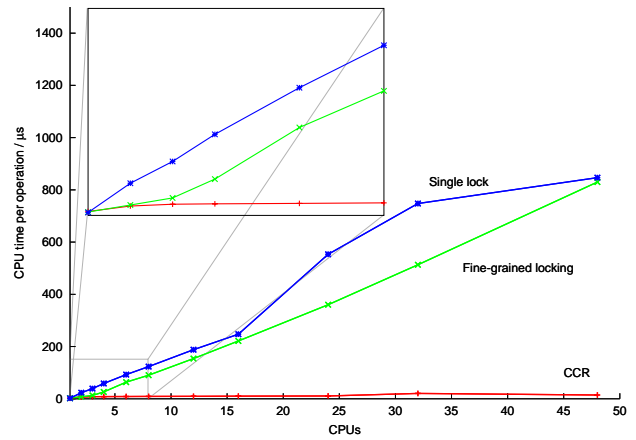
(a) Individual hashtable operations, 1% writes



(b) Individual hashtable operations, 16% writes



(c) Compound operations, 256-element table



(d) Compound operations, 4096-element table

**Figure 6: Performance of concurrent updates running 1.48 threads.** In (a) and (b) we perform individual atomic updates to a single hashtable. In (c) and (d) we perform compound operations making two updates to the table in one atomic step. The insets show 1.8 threads, confirming that our scalability is combined with good absolute performance.

slower than using a single lock). The improved performance comes from the fact that using CCRs allows parallelism between non-conflicting operations which would contend for the same lock in `ConcurrentHashMap` and from the fact that occasional pre-emption of one thread in any of the lock-based designs will stall other threads. Note that the CCRs performing swap operations commit only 2 updates to the application heap; the number of atomic operations to acquire and release those locations is likely to be the same as the number in a conventional lock implementation.

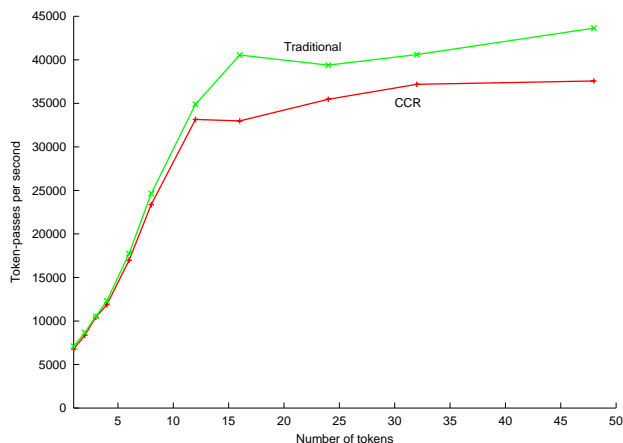
Figure 7 shows the performance of the `wait` test on the `e15k`, plotting how many `put` or `get` operations are achieved per second on the machine as a whole as the number of tokens available to the 48 threads increases. In principle

the results should scale linearly, although in practise any scheme reaches a plateau, representing the point at which the threads' time is consumed managing the shared buffers rather than exchanging tokens through them. Again, the STM-based design performs only marginally below the traditional one.

## 6.8 Ease of programming

We now turn to the final aspect of evaluation; do CCRs provide a programming abstraction that is easier to use than the existing facilities of Java? At the moment we can offer only anecdotal observations.

Firstly, as we remarked in Section 2, CCR-style abstractions are popular when introducing the topic of concurrency



**Figure 7: Performance of blocking operations running 48 threads.**

to students. We suspect that reasoning about the behaviour of CCRs – whether at a formal or an informal level – is made easier by the ability to consider the enclosed statements as a single step in an operational model. In contrast, lock-based schemes other than simple monitors require reasoning about interleavings between parts of operations.

Secondly, the provision of CCRs has close analogies with the concept of database transactions. Both provide simple semantics and isolated execution behaving as-if atomically. Both leave the exact implementation of this behaviour to a run-time system rather than requiring programmers to identify where to acquire and release locks. The popularity of transactional concepts, and the acceptance of this model by mainstream programmers, gives us confidence about our style of CCR.

Finally, we can compare the design of simple shared data structures using Java’s existing monitors against the corresponding design using CCRs. For data structures which do not involve `wait`, the difference is solely that `synchronized` is replaced by `atomically` on each method; the result is that the implementation may extract more concurrency from dynamically non-conflicting operations and that the risk of accessing shared data from outside the locked object is avoided. For data structures which involve blocking using `wait` and `notify`, the simplification is even greater – the CCR-based design expresses the pre-conditions directly and avoid lost-wake-up and premature-wake-up problems.

We hope, in the future, to be able to perform more methodical user-studies.

## 7. CONCLUSION AND FUTURE WORK

In this paper we have argued that concurrent programming is made easier by moving away from locks and condition variables and instead using facilities that more closely capture the safety properties that a programmer is trying to enforce.

We have a number of directions for future work. We plan to explore the performance of a wider range of data structures and, if suitable multi-threaded benchmarks are available, to study performance in larger systems. Our Java CCR implementation benefits from the fact that the bytecode in-

structions let us easily distinguish between accesses to local variables and accesses to potentially-shared fields. The underlying STM is more general; indeed source code to the core implementation is available under a BSD-style license at <http://www.cl.cam.ac.uk/Research/SRG/netos/lock-free>. It will be interesting to consider applying it to a lower level language such as C++.

One design question is whether to directly expose a transactional API in addition to CCRs. This raises interesting questions about reflective access, for instance allowing a thread to determine whether a transaction is active, to examine the set of updates that have been proposed thus far, or perhaps to explicitly create and attempt to commit transactions or to enter and leave transactional contexts.

Another question is whether there are merits in having *all* methods be implicitly atomic. In some cases this may simplify the mental model of an object’s behaviour: it receives a series of method invocations, each of which updates its state in isolation. The programmer would be concerned only with the interleaving of complete method calls. Of course, the initial problem with this is deciding when updates should actually become visible; perhaps models developed for pure functional languages may be appropriate.

A further avenue for research is to investigate how our in-memory atomic updates can be integrated with external storage. The correspondence between CCRs and database transactions may provide a particularly fruitful direction.

Another question, about which numerous viewpoints already exist in the literature, is what kinds of hardware support would benefit designs such as our STM. Proposals have already been made for hardware transactional memories, with suggestions for implementation techniques based on extended cache coherence protocols [13]. It would be interesting to consider what hardware/software interface would be appropriate to build `STMwait` – perhaps one in which “tripwire” locations can be registered with the CPU and an interrupt delivered should one be updated. Another option is support for multi-word atomic updates some where between the current single-word operations and a fully general transactional model. We have often found algorithms that would be simplified, both here and in previous work, through an operation to perform one CAS conditional on another location holding a specified value [8].

In summary, we have shown how general conditional critical regions can be supported and how a basic implementation of a data structure using this construct is competitive with a well-engineered lock-based scheme. We believe that this approach makes it substantially easier to write reliable concurrent systems; it is no coincidence that the same construct is frequently used in text books and in the specification of concurrent systems.

## 8. ACKNOWLEDGEMENTS

This work has been supported by a donation from the Scalable Synchronization Research Group at Sun Labs Massachusetts.

## 9. REFERENCES

- [1] AGESEN, O., DETLEFS, D., GARTHWAITE, A., KNIPPEL, R., RAMAKRISHNA, Y. S., AND WHITE, D. An efficient meta-lock for implementing ubiquitous synchronization. In *Proceedings of Object-Oriented*

- Programming, Systems, Languages & Applications* (Nov. 1999), vol. 34(10) of *ACM SIGPLAN Notices*, pp. 207–222.
- [2] ANDREWS, G. R. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings Publishing Company, Inc., 1991.
  - [3] BACON, J., AND HARRIS, T. L. *Operating Systems: Concurrent and Distributed Software Design*, 3rd ed. Addison Wesley, 2003.
  - [4] BENTON, N., CARDELLI, L., AND FOURNET, C. Modern concurrency abstractions for C#. In *Proceedings of European Conference on Object-Oriented Programming* (June 2002), vol. 2374 of *Lecture Notes in Computer Science*, pp. 415–425.
  - [5] BRINCH HANSEN, P. Distributed processes: A concurrent programming concept. *Communications of the ACM* 21, 11 (Nov. 1978), 934–941.
  - [6] BRINCH HANSEN, P. Edison – a multiprocessor language. *Software – Practice and Experience* 11, 4 (Apr. 1981), 325–361.
  - [7] CARDELLI, L., AND GORDON, A. Mobile Ambients. In *Proceedings of Foundations of Software Science and Computation Structures (FoSSaCS), European Joint Conferences on Theory and Practice of Software* (Lisbon, Portugal, 1998), M. Nivat, Ed., vol. 1378 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 140–155.
  - [8] HARRIS, T. L., FRASER, K., AND PRATT, I. A. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Symposium on Distributed Computing* (Oct. 2002), pp. 265–279.
  - [9] HENNESSY, J. L., AND PATTERSON, D. A. *Computer Architecture – A Quantitative Approach*, 3rd ed. Morgan Kaufmann Publishers, San Francisco, CA, USA, 2003.
  - [10] HERLIHY, M., LUCHANGCO, V., AND MOIR, M. Obstruction-free software transactional memory for supporting dynamic data structures. To appear.
  - [11] HERLIHY, M., LUCHANGCO, V., AND MOIR, M. Obstruction-free synchronization: Double-ended queues as an example. To appear.
  - [12] HERLIHY, M., LUCHANGCO, V., AND MOIR, M. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *Proceedings of the 16th International Symposium on Distributed Computing* (Oct. 2002), pp. 339–353.
  - [13] HERLIHY, M., AND MOSS, J. E. B. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture* (May 1993), IEEE Computer Society Press, pp. 289–301.
  - [14] HOARE, C. A. R. Towards a theory of parallel programming. In *Operating Systems Techniques* (1972), vol. 9 of *A.P.I.C. Studies in Data Processing*, Academic Press, pp. 61–71.
  - [15] HOARE, C. A. R. *Communicating Sequential Processes*. Prentice-Hall, 1985.
  - [16] LIANG, S., AND BRACHA, G. Dynamic class loading in the Java Virtual Machine. In *Proceedings of Object-Oriented Programming, Systems, Languages & Applications* (Oct. 1998), vol. 33(10) of *ACM SIGPLAN Notices*, pp. 36–44.
  - [17] LINDHOLM, T., AND YELLIN, F. *The Java Virtual Machine Specification*, 2nd ed. Addison-Wesley, Reading, MA, USA, 1999.
  - [18] LISKOV, B., AND SCHEIFLER, R. Guardians and actions: linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems* 5, 3 (July 1983), 381–404.
  - [19] MANSON, J., AND PUGH, W. Semantics of multithreaded Java. Tech. Rep. UCMP-CS-4215, Department of Computer Science, University of Maryland, College Park, Jan. 2002.
  - [20] MICHAEL, M. M. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing* (July 2002), ACM Press, pp. 21–30.
  - [21] MICHAEL, M. M., AND SCOTT, M. L. Correction of a memory management method for lock-free data structures. Tech. Rep. TR599, University of Rochester, Computer Science Department, Dec. 1995.
  - [22] MILNER, R. The pi calculus and its applications. In *Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming* (June 1998), pp. 3–4.
  - [23] MOIR, M. Transparent support for wait-free transactions. In *Proceedings of the 11th International Workshop on Distributed Algorithms* (Sept. 1997), vol. 1320 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 305–319.
  - [24] PLATT, D. S. *Introducing Microsoft .NET*, 2nd ed. Microsoft Press, 2002.
  - [25] SCHMID, H. A. On the efficient implementation of conditional critical regions and the construction of monitors. *Acta Informatica* 6, 3 (Aug. 1976), 227–249.
  - [26] SCOTT, M. L. Language support for loosely coupled distributed programs. *IEEE Transactions on Software Engineering SE-13*, 1 (Jan. 1987), 88–103.
  - [27] SHAVIT, N., AND TOUITOU, D. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing* (Aug. 1995), ACM Press, pp. 204–213.