Revocable Locks for Non-Blocking Programming

Tim Harris
Microsoft Research
7 J J Thomson Avenue
Cambridge, UK, CB3 0FB
tharris@microsoft.com

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

ABSTRACT

In this paper we present a new form of revocable lock that streamlines the construction of higher level concurrency abstractions such as atomic multi-word heap updates. The key idea is to expose revocation by displacing the previous lock holder's execution to a safe address. This provides mutual exclusion without needing to block threads. This brings many simplifications, often removing the need for dynamic memory management and letting us strip operations from common-case execution paths. As well as streamlining algorithms' design, our results show that the technique leads to improved performance and scalability across a range of levels of contention.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming – Parallel programming; D.3.2 [**Programming Languages**]: Language Classifications – Concurrent, distributed, and parallel languages; D.4.1 [**Operating Systems**]: Process Management – Concurrency; Synchronization; Threads

General Terms

Algorithms, Experimentation, Performance

Keywords

Non-blocking algorithms, locks, transactional memory

1. INTRODUCTION

It is hard to build scalable concurrent programs using ordinary locks. To ensure correctness programmers must identify which operations are conflicting. To ensure liveness they must avoid introducing deadlock or priority inversion. To ensure good performance they must balance the granularity at which locking is performed against the number of locks that threads need to acquire and release.

To sidestep these problems, alternative abstractions have been developed to provide atomic multi-word updates [12, 16, 6, 11]. For instance, in recent work we showed how to provide atomic code blocks of the form `atomic { S }` in a high level language: the statements in `S` and all of the methods that they call are performed atomically with respect to other code [11, 10]. This abstraction allows single-threaded operations to be made safe for multi-threaded use.

However, although software implementations of atomic blocks can scale well, they have high baseline performance costs – around a factor-of-two overhead in uniprocessor cases is typical. Our research is looking at how much we can reduce these costs: if they are unavoidable then that provides motivation for hardware support [17, 25]. If costs can be reduced then that suggests hardware resources should be deployed elsewhere.

In this paper we make three contributions to the construction of non-blocking multi-word updates. Firstly, in Section 2 we introduce a new taxonomy of the problems involved, dividing them into *partial update problems* and *delayed operation problems*. The former are usually easy to solve, but the latter are insidious.

Secondly, in Section 3, we define a new form of revocable lock which allows us to avoid delayed operation problems.

The key novelty is in how conflicts are managed: if a thread `A` attempts to acquire a lock held by thread `B` then the lock is passed to `A` and, atomically with this, `B`'s execution is displaced to a recovery function which `B` specified when it acquired the lock. Typically, the recovery code would either propagate revocation as a higher level failure (for instance, returning *failed to commit* in a transactional memory implementation), or it would retry the operation (after some contention-management delay to avoid live-locking with `A`).

As with ordinary locks, it is usually profitable for `A` to spin briefly in the hope that `B` releases the lock before the heavyweight revocation operation proceeds. We describe three implementation schemes based on widely available operating system support.

Our final contribution, in Section 4, shows how revocable locks can be used to streamline the implementation of two non-blocking algorithms from the literature: Greenwald's *two-handed emulation* and our own *word-based software transactional memory* (STM). We obtain major simplifications to both algorithms and, as our results show in Section 5, these yield improved performance and scalability across a wide range of workloads.

Revocable locks are not intended for direct use by application programmers in the ways that atomic regions, mutexes, semaphores or barriers may be used. Rather, they are for use within the implementation of these higher level abstrac-

PPoPP'05, June 15–17, 2005, Chicago, Illinois, USA.

tions. For instance, we use a revocable lock just within the implementation of an STM commit operation: we do not need to hold such a lock through a complete transaction. This allows us to make clear simplifications in building revocable locks – in particular, we allow threads to hold at most one such lock at a time.

The performance improvements we achieve come from the fact that the *ability* to revoke locks allows us to safely simplify the common-case paths through the abstractions that we build using them, at the expense of high costs when revocation is actually performed. However, as we quantify in our results, we *expect revocation to be extremely rare* and so, as Amdahl's law suggests, it is a trade-off worth making.

We believe this is an acceptable assumption to make – the same assumption is made when using optimistic concurrency in the kind of non-blocking data structures that we can build using revocable locks. In cases where contention is high, one could switch to ordinary mutexes and use logging for rollback as Welc *et al* have suggested [27].

2. NON-BLOCKING ALGORITHMS

Non-blocking algorithms have been studied as a way of avoiding the problems caused by traditional locks [13]. For instance, recent algorithms for performing atomic multi-word updates are examples of non-blocking designs [12, 16, 6, 11]. A system is non-blocking if the suspension or failure of any 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 ordinary locks because, unless a lock-holder continues to run, the lock can never be released.

In common with most contemporary work on non-blocking systems, we assume the existence of a single word compare-and-swap operation (CAS):

Single-word CAS operation

```
word_t CAS(addr_t a, word_t o, word_t n)
```

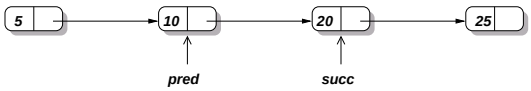
This operation atomically reads from memory address `a` and, if the value seen is equal to `o` it updates the location to hold `n`. It returns the value that it read from memory. CAS is implemented in hardware on SPARC, IA-32 and IA-64 processors and it can readily be constructed on Alpha, MIPS and PowerPC architectures.

2.1 Problems in non-blocking design

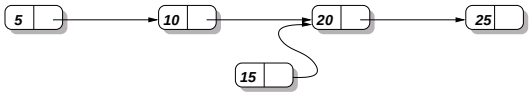
It is possible to identify two kinds of difficulty which occur when building non-blocking algorithms. The first are *partial update problems* which occur because new operations can start at any point and may therefore interact with partially-complete updates being performed by other threads. This means that system invariants must be maintained by each individual step of an update, rather than just by the operation as a whole. It also means that sufficient information must be available for threads encountering a partial update to either complete the operation (giving lock-free behaviour [13]) or to undo it (giving obstruction-free behaviour [15]). These problems can be readily solved by having threads publish their intentions in shared memory before starting to perform an operation ([12, 16, 6, 11] give numerous illustrations).

The second, and more severe, difficulty stems from what we term *delayed operation problems*. These occur because, once started, there is no guarantee about when the scheduler will actually select a given thread for execution. This means that, for any instruction reachable during an operation, the scheduler can potentially run the thread to that point, preempt it, and then resume at that instruction at any time in the future. Non-blocking algorithms must be designed so that the effect of any delayed operation is benign.

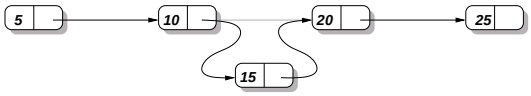
As an example, consider the insertion of a new node holding the value 15 into a sorted singly-linked list currently holding nodes with values 5, 10, 20 and 25. The insertion can proceed in three steps. Firstly, the predecessor and successor are identified in the usual way:



Next, the new node is prepared in thread-private storage:



Finally, a CAS is used to splice the new node into the list between the predecessor and the successor:



This example illustrates the three kinds of delayed operation problem:

- *Delayed reads* can cause segmentation faults if the address being accessed is no longer valid. For instance, a thread cannot free the memory containing the 5 node while it might be seen by threads traversing the list. Solutions include the use of garbage collection or techniques based on threads publishing sets of objects that they may access in the immediate future [20, 14].

- *Delayed writes* are more problematic: they have the potential to update the contents of a memory location. The usual way of making these writes benign is to arrange that they do not affect the logical state of the system even though they update its physical representation. For instance, the writes which initialise the 15 node are made to thread-private storage and will not be seen by other threads until the insertion is complete.

- *Delayed CASs* can cause segmentation faults as before and, in addition, can update the contents of a memory location if it holds the *expected value* specified by the CAS. In many cases designers must avoid so-called A-B-A problems [21] in which a thread is about to perform a CAS conditional on a location holding a value `A`, but then a series of operations by other threads changes the value to `B` and then back to `A` allowing the delayed CAS to succeed even though the update may no longer be correct. Typical solutions to A-B-A problems are to avoid re-using values while there may

be delayed CASs conditional on them – for instance, during this insertion it would be unsafe to re-use the storage holding the 20 node.

With these kinds of problem in mind, we can view the use of traditional locks and conventional non-blocking techniques as two points in a design space. At one extreme, locking provides a way for preventing partial updates from being visible and for preventing delayed operations from occurring. At the other extreme, non-blocking designs without locking require the programmer to make the algorithm robust against both kinds of problem.

3. REVOCABLE LOCKS

In this paper we introduce a new kind of non-blocking revocable mutual-exclusion lock which allows us to avoid delayed operation problems and thereby simplify non-blocking systems' design. As with conventional mutexes, these new locks allow at most one thread to hold them at any instant in time. However, if a thread `A` attempts to acquire a lock held by `B` then, instead of blocking `A`, the lock is revoked from `B` and passed on to `A`. Before `A`'s lock-acquire operation returns, `B`'s execution is displaced to a recovery function that `B` supplied when it acquired the lock.

As Figure 1 indicates, these semantics provide a middle-ground between using traditional mutexes and attempting to build non-blocking algorithms without any form of locks. They are sufficient to avoid delayed operation problems because at most one thread can execute code protected by each lock at any time: revocation has the effect of canceling delayed operations which would otherwise occur. Of course, revocation can occur at any stage through a lock-holder's operation and so this means that we cannot prevent partial updates from being visible.

By avoiding delayed operations we get a number of simplifications in the examples that we have studied. In these examples we typically associate a revocable lock with some form of *operation descriptor* structure in which a thread sets out the details of a multi-word operation that it is performing. The lock holder is responsible for performing this operation and has sole use of the structure. This lets us:

- replace dynamic heap allocation with static or on-stack allocation,
- avoid the need for tracing garbage collection, or for reference counting the operation descriptors (with its attendant costs in terms of memory write barriers),
- safely assume that the contents of descriptors are unchanged by other threads, meaning that we can replace many `CAS` operations by direct updates, and change many other double-word `CAS` operations into single-word variants.

We describe these cases in more detail in Section 4. We have yet to find an algorithm which requires more than a single revocable lock per thread – revocable locks are typically only used within critical parts of library functions, for instance in building the commit operation of an STM, or the contended lock-acquire case of a mutex implementation. We would not expect a thread to hold a revocable lock between libary calls – for instance while executing a software transaction, or while holding or blocking on an ordinary mutex.

	Partial updates visible	Delayed operations possible
Traditional locks	No	No
Revocable locks	Yes	No
No locks	Yes	Yes

Figure 1: Revocable locks provide a middle-ground between traditional locks and the direct construction of non-blocking algorithms.

3.1 Hold-release operations

In our current design we associate revocable locks with heap locations and provide operations to access a data item at that location along with operations to lock and unlock it. Four operations are provided in total:

Hold-release operations

```
hr_word_t HRRead(addr_t a)
void HRWrite(addr_t a, hr_word_t w)
void HRHold(addr_t a, pc_t r)
void HRRelease()
```

`HRRead` and `HRWrite` correspond to conventional reads and writes. The data values they deal with are of type `hr_word_t` which, in the implementations we present in Section 3.2, is an ordinary machine word with one bit reserved.

The third operation, `HRHold`, acquires a revocable lock on the location `a`. The lock is held until either (*i*) the thread releases it with a `HRRelease` operation, (*ii*) the thread invokes `HRHold` on a different location, or (*iii*) the lock is revoked by another thread performing a `HRWrite` or `HRHold` operation on the same location. If the lock is revoked then the program counter of the thread holding it is moved to the revocation target `r`.

3.2 Implementation of hold-release operations

In this section we describe three software-based implementations of the hold-release operations. In all cases, heap locations acted on by these operations have a single reserved bit: ordinarily this is 0 and said to be *unmarked*; if the location is held then it is 1 and said to be *marked*. The operations `MARK`, `UNMARK` and `IS_MARKED` are used to set, clear and interrogate such bits and are implemented using the obvious bit-wise operations.

If a location is not held then its contents are stored directly in it as an unmarked value. If a location is held then it contains a marked pointer to a statically allocated per-thread structure of the holder. The format of this structure is shown in Figure 2 (`hr_per_thread_t`). If `addr` is non-NULL then it indicates the address currently held by the thread and `displaced` holds the value logically held at that location. The two counters, `holds_started` and `holds_completed` are incremented respectively before and after the thread performs a `HRHold` operation.

In each of our three implementations the representation used in memory is the same; the differences lie in how revocation is implemented. We exploit this commonality by presenting the implementation in two stages, firstly using a `FETCH` operation which returns the current contents of the address after revoking any thread holding it, and secondly by showing showing how `FETCH` is implemented.

```
struct {
  addr_t addr;
  word_t displaced;
  int    holds_started;
  int    holds_completed
} hr_per_thread_t;

hr_word_t HRRead(addr_t a) {
  if (a == st -> addr) {
    return st -> displaced;
  } else {
    do {
      owner = *a;
      if (IS_UMMARKED(owner)) return owner;
      holds_started = owner -> holds_started;
      if (owner -> addr == a) {
        val = owner -> displaced;
        if (owner -> holds_completed ==
            holds_started) {
          return val;
        }
      }
    } while (TRUE);
  }
}

void HRWrite(addr_t a, hr_word_t w) {
  if (a == st -> addr) {
    st -> displaced = w;
  } else {
    do {
      expected = FETCH (a);
    } while (CAS(a, expected, w) != expected);
  }
}

void HRHold(addr_t a, pc_t d) {
  HRRelease();
  st -> holds_started ++;
  do {
    expected = FETCH(a);
    st -> addr = a;
    st -> displaced = expected;
  } while (CAS(st -> addr, expected, MARK(st)) != expected);
  st -> holds_completed ++;
}

void HRRelease() {
  if (st -> addr != NULL) {
   *(st -> addr) = st -> displaced;
  }
  st -> addr = NULL;
}
```

Figure 2: Implementation of the hold-release operations. `FETCH` performs revocation where necessary. The identifier `st` refers to the per-thread data structure of the current thread.

FETCH operation

```
hr_word_t FETCH(addr_t a)
```

`HRRead` and `HRWrite` act on `displaced` if the thread holds the indicated location. Otherwise, for `HRRead`, there are two cases to consider: if the value in the location is unmarked then it can be returned directly, if the value is marked then the value from the owner's `displaced` field is returned. The owner's `holds_started` and `holds_completed` fields are used to allow the reader to take a consistent snapshot of the `addr` and `displaced` fields.

`HRWrite` proceeds in two stages if the invoker does not hold the indicated location. The first stage is to `FETCH` the location, meaning to revoke the current holder (if any) and to return the location's current value. The second stage is to perform a CAS on the location from the value FETCHed to the new value: this ensures that the location has not become held again since retrieving its value. `HRHold` also uses a `FETCH` operation: the location is fetched, the caller's structure is updated and then a CAS is used to install a marked pointer to the structure. Figure 2 provides pseudo-code implementing these operations.

The first implementation of `FETCH` is a straightforward one which does not provide non-blocking behaviour: instead of performing revocation, the fetcher waits until it reads an unmarked value from the location.

The second implementation, for Solaris UNIX, does allow revocation. It uses the `/proc/` interface to suspend the thread currently holding a location and then to update its PC to its revocation target. In order to prevent deadlocks, for instance two threads suspending each other at the same time, a process-wide lock is employed to allow at most one thread to be performing a suspend operation at any time. This prevents the implementation from being non-blocking, however, we do not believe that this is a practical concern in a multi-threaded application running in a single process. We use the `schedctl` interface to discourage a thread holding the suspension lock from being descheduled. Similar facilities for thread suspension and control exist in the Win32 Platform SDK and other operating systems.

The third implementation, again for Solaris UNIX, allows revocation and is non-blocking. In it, each thread runs a separate copy of any functions executed while holding a revocable lock. Revocation is implemented by using `mprotect` to remove execute permission from the page holding the owner's copy of the functions.

In our performance results in Section 5 we use a hybrid scheme in which threads spin for a short while, waiting for the current holder of a location to release it voluntarily, before taking a slow path which uses the `/proc/` interface to revoke them. As with common implementations of mutual exclusion locks, this anticipates that the incumbent is likely to release the location in the near future. Even a modest spin limit (1000 iterations of a tight loop) is sufficient to make the slow path virtually never executed: the heavyweight cost of taking it is rarely incurred. We quantify this in Section 5.

4. USING HOLD-RELEASE OPERATIONS

In this section we consider two example non-blocking data structures from the literature and show how their design can be simplified by using revocable locks. The purpose of this is to demonstrate that revocable locks provide a general solution to delayed operation problems and that algorithms designed using it are simplified and more 'clearly correct' than their original counterparts. In Section 4.1 we consider Greenwald's *two handed emulation* scheme and then in Section 4.2 we consider our *word-based STM* [11].

4.1 Two-handed emulation

In his paper at PODC 2002, Greenwald introduces the mechanism of *two-handed emulation* as a way of simplifying the

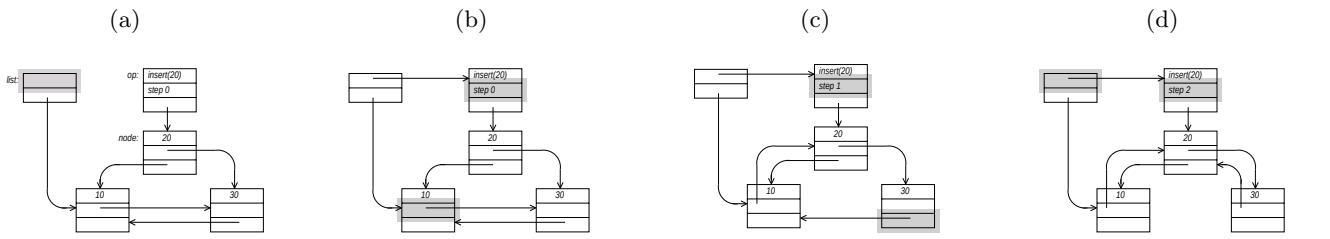


Figure 3: Insertion of a node into a doubly-linked list using two-handed emulation. Shaded boxes indicate the locations that are to be accessed by a CAS or DCAS in the current step.

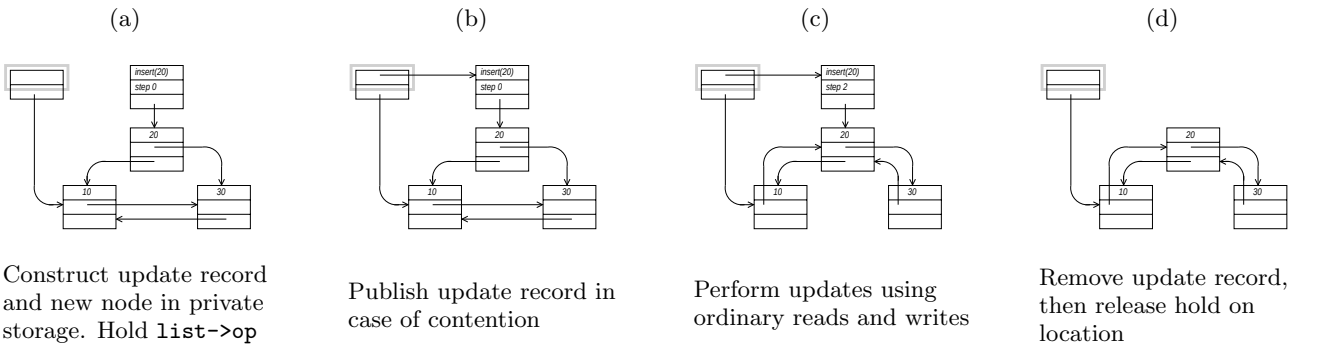


Figure 4: Insertion of a node into a doubly-linked list using hold-release operations. The grey rectangle indicates the location being held. Other memory accesses use ordinary read and write operations. DCAS is not required.

construction of non-blocking data structures [8]. Greenwald posits that his design provides evidence in favour of hardware support for a *double-word compare-and-swap* (DCAS) operation which takes six parameters:

Double-word CAS operation

```
bool_t DCAS(addr_t a1, word_t o1, word_t n1,
            addr_t a2, word_t o2, word_t n2)
```

DCAS acts as a double-word version of CAS, checking the contents of addresses `a1` and `a2` against `o1` and `o2` and, if both addresses hold their expected values, updating them with `n1` and `n2` respectively. In general DCAS returns a boolean result indicating success or failure. Although many published algorithms use it, DCAS has not been supported in hardware since the Motorola 68k processor family [1].

To perform non-blocking updates using two-handed emulation, each shared data structure is augmented with a *current operation* field. If any thread is performing an operation on the structure then this field points to a record describing the operation being done and how far that operation has progressed through a series of steps. An operation proceeds by using DCAS to increment the step counter while performing an update relating to the current step. The 'two hands' refer to the two atomic accesses which DCAS is able to make at each step. The scheme leads to a non-blocking implementation because if a thread `A` encounters thread `B` performing an operation which obstructs it, then `A` can help `B` complete its operation. The step counters avoid delayed operations during this helping.

For instance, Figure 3 shows how a thread performing an insertion into a doubly-linked list could proceed. Step (a) installs an operation record describing the insert. Step (b) links the new node in the 'forward' direction. Step (c) links the node in the 'reverse' direction. The final step, (d), removes the operation record.

However, the use of DCAS makes this design ineffective for two reasons. Firstly, no modern processor provides a hardware implementation of DCAS. Although software implementations exist, they expand each DCAS into a series of CAS operations (7 in the original design [12]) and require temporary dynamically allocated data structures. Secondly, in cases where contention is rare, the processor's ability to re-order memory accesses will be constrained by the need to serialise the execution of the DCAS operations.

In contrast, an implementation of insertion in doubly-linked lists can be developed using hold-release without needing DCAS. The new design proceeds using the same basic steps as two-handed emulation, but with the thread performing the insertion holding `list->op`. This ensures that that thread remains the only one acting on the data structure.

Figure 4 shows the resulting steps. Note that the operation record must still be published in the list structure in order to give non-blocking behaviour: it allows other threads, after revoking the hold, to continue the operation that was in progress. However, the design based on hold-release can use ordinary memory accesses to make the actual updates to the data structure and to the step counter without needing either CAS or DCAS. In some algorithms – although not in this list-based example – it may still be necessary to refactor the design to make each step idempotent since steps are no longer performed atomically with the step count updates.

4.2 Streamlined STM

In this section we show how the hold-release operations can be used to produce a streamlined word-based STM based on our design at OOPSLA 2003 [11]. After presenting the interface that the STM exposes we outline the common-case implementation of the STM for non-contended transactions in Section 4.2.1. Then, in Section 4.2.2 we discuss contended heap accesses and show how the hold-release operations can be used to avoid the delayed operation problems that occur.

Four operations are provided for transaction management:

Transaction management

```
void TransactionStart()
void TransactionAbort()
boolean TransactionCommit()
boolean TransactionValidate()
```

These have their usual meaning in transaction processing. Invoking `TransactionStart` begins a new transaction within the executing thread. `TransactionAbort` aborts the transaction in progress by the executing thread. `TransactionCommit` attempts to commit the transaction in progress by the executing thread, returning `true` if this succeeds and `false` if it fails. `TransactionValidate` indicates whether the current transaction would be able to commit.

Two operations are provided for performing memory accesses within a transaction:

Memory accesses

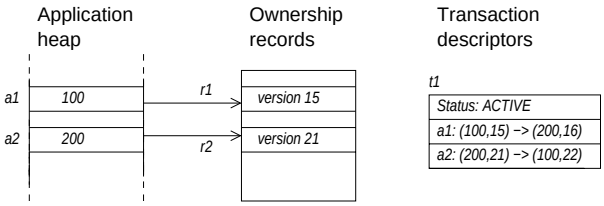
```
word_t TransactionRead(addr_t a)
void TransactionWrite(addr_t a, word_t w)
```

The memory locations accessed through `TransactionRead` and `TransactionWrite` are disjoint from those accessed directly through ordinary read and write operations.

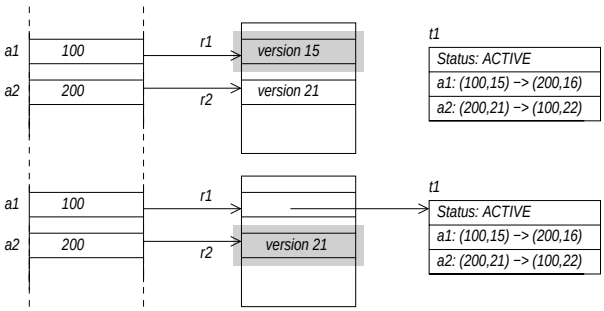
4.2.1 Non-contended transactions

The STM uses a form of optimistic concurrency control with a commit operation based on version numbering. Transactions initially execute in private, building up values in a *transaction descriptor* which sets out the memory accesses that it has performed. We indicate such accesses using the notation `a1:(o,vo)->(n,vn)` to indicate an access to heap address `a1` updating it from value `o` at version number `vo` to value `n` at version number `vn`. For a read-only access, `n==o` and `vn==vo`. For an update, `vn==vo+1`. Additionally, the descriptor has a status field indicating that it is either `ACTIVE`, `COMMITTED` or `ABORTED`.

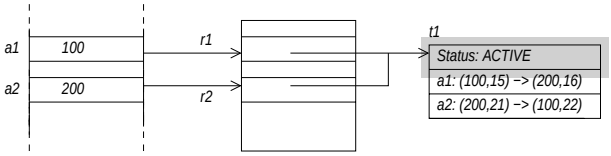
The `TransactionCommit` operation then attempts to validate these updates and, if successful, atomically exposes them to other threads. The STM uses a set of *ownership records* (orecs) to co-ordinate concurrent validation and



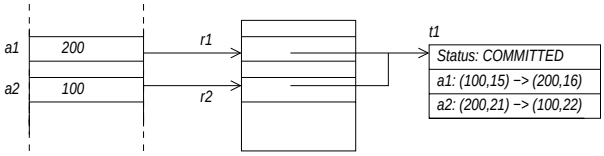
(a) The transaction executes in private until it attempts to commit.



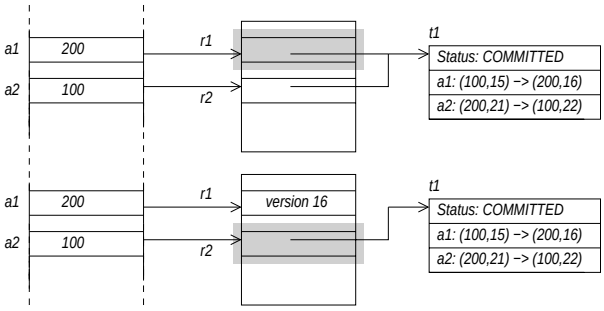
(b) CAS is used to acquire ownership records `r1` and `r2`, replacing the expected version number with a pointer to the transaction descriptor.



(c) CAS is used to set the status to `COMMITTED`.



(d) The updates are written back to the heap.



(e) Ownership is released on `r1` and `r2`, installing the new version numbers.

Figure 5: An uncontended commit swapping the contents of `a1` and `a2`. Grey boxes show where CAS operations are to be performed at each step. While an orec is owned, the *logical contents* of the locations involved are available via the transaction descriptor.

commit. An *ownership function* maps each heap address to an associated orec – this may be a many-to-one mapping and, in our implementation, we simply take a fixed number of the significant low-order bits of the heap address. Each orec either holds a reference to the transaction currently owning it, or it holds a version number indicating how many updates have been committed to locations associated with the orec.

An uncontended transactional commit proceeds in four stages. In the first stage it acquires ownership of the orecs associated with the locations that it has accessed, installing a pointer to the transaction descriptor in each of them using CAS. This serves two purposes: as well as acquiring ownership, it confirms that each orec held the version number expected by the transaction. In the second stage the transaction's status is set to `COMMITTED` using CAS. This is the point at which the update appears to occur atomically: threads reading from the locations while they are owned will take values from the owner's transaction descriptor on the basis of the status field. This single update therefore has the effect of atomically updating the *logical contents* of all of the locations involved.

In the third stage the transaction's updates are made to the heap. Finally, in the fourth stage, it releases ownership of the orecs it acquired, updating the version numbers.

Figure 5 illustrates this process for a transaction which is attempting to swap the contents of heap addresses `a1` and `a2`.

4.2.2 Delayed operation problems due to contention

Contended heap accesses pose two problems. Firstly, a call to `TransactionRead` or `TransactionWrite` may be performed on an address whose orec is currently owned, meaning that the version number is not directly available and that the value held in the location may be out of date (for instance just after the state shown in Figure 5(c)). This can be dealt with by obtaining the information from the owning transaction descriptor.

The more insidious problem is that one thread `A` performing `TransactionCommit` may encounter an orec which is already owned by another thread `B`. In order to get non-blocking behaviour it must be possible for `A` to continue with its commit without having to wait for `B`. There are a number of cases to consider based on the state of `B`'s transaction descriptor:

- If `B`'s transaction descriptor is currently `ACTIVE` then `A` can set it to `ABORTED` using CAS. This gives rise to potential A-B-A problems which prevent descriptors from being re-used directly: re-use would allow a delayed CAS by `A` to abort a subsequent transaction by `B`.

- If `B`'s transaction descriptor is already set to `ABORTED` then `A` can proceed to unlink `B` from the orecs that it acquired. This unlinking step again gives rise to potential A-B-A problems if the descriptor were re-used while `A` was unlinking a previous version of it.

- If `B`'s transaction descriptor is currently `COMMITTED` then `A` cannot revoke `B`'s ownership: `B` may be performing its updates to the heap (Figure 5(d)) and, even though `A` may perform those updates on behalf of `B`, there is nothing to prevent `B` being scheduled at some time in the future and performing the updates a second time.

In the first two cases, A-B-A problems can be dealt with by reference counting descriptors in order to prevent re-use. This requires that descriptors are dynamically allocated and adds reference counting operations (and associated memory barriers) to the STM's implementation.

In the third case, delayed writes can be made benign by avoiding unlinking a transaction descriptor from an ownership record until it is certain that no delayed writes may exist. This is done by adding an *ownership count* to each orec, holding the number of threads which may be performing writes to locations associated with that orec (i.e. step (d) in Figure 5). The version number is only restored to the orec when the count reaches zero. When one thread wishes to steal an orec from another, the thief merges the victim's transaction descriptor into its own and then performs an atomic update to swing ownership to the new record and to increment the ownership count.

This scheme based on stealing has three problems. Firstly, each orec must be large enough to accommodate the count field as well as a pointer to the owning transaction descriptor. This means that double-word-width CAS must be used to update them.

Secondly, if a thread is preempted while holding ownership records then, although others can make progress by stealing ownership, the orecs involved cannot be released until the original thread resumes execution. This slows `TransactionRead` operations to locations managed by the orec because they must search the transaction descriptor rather than being able to read directly from the heap.

Finally, when merging transaction descriptors before stealing ownership, the thief must ensure that sufficient space exists in their descriptor to accommodate the new entries.

4.2.3 Using hold-release

The hold-release operations provide a remarkably simplified mechanism for avoiding these delayed operation problems. During a commit operation, each thread holds the status field of the transaction descriptor that it is working on. This means that, while the thread is still executing the commit operation, it can be certain that it is exclusively responsible for performing the operations set out in the transaction descriptor: in Figure 5 stages (b)–(e) are all performed while holding the descriptor.

If a thread `A` encounters an orec owned by another thread `B` then `A` releases the status field on its own transaction descriptor and instead takes hold of the status field of `B`'s descriptor. At that point, it can be certain that it is the only thread acting on `B`'s descriptor because `B` will have been displaced to its revocation target. Once `A` has completed `B`'s operation it can release the status field of `B`'s descriptor, take hold of its own, and re-try its original commit operation.

In effect, the transaction descriptors are used to represent pieces of work which some thread wishes to perform. The revocable locks provide a way to ensure that at most one thread is performing the work specified in a given descriptor at a given time. This lets us make a series of simplifications to the implementation of the STM:

- Transaction descriptors can be statically allocated and a thread can immediately re-use its descriptor after committing a previous update in it.

The single-ownership enforced by hold-release takes the place of memory management schemes such as reference counting, PTB [14] or SMR [20] in preventing A-B-A problems caused by delayed CAS instructions. This simplification (*i*) removes memory management operations from the commit code, (*ii*) means that a thread can continually re-use the same descriptor, perhaps giving improved data-cache locality and (*iii*) removes a level of indirection between a per-thread structure and that thread's current descriptor.

- Since revocation prevents delayed writes, it is no longer necessary to allow multiple threads to own the same orec at the same time. This removes the need for ownership counts in the orecs and removes the need to merge updates from one descriptor into another.

  This simplifies the acquire and release steps (b) and (e) and, in our implementation, means that orecs can be updated with a single-word CAS rather than a double-word CAS.

5. RESULTS

In this section we evaluate the performance of a system built using revocable locks. We are concerned with two aspects of performance: the overall run-time of a variety of workloads and the likelihood of needing to perform a lock revocation.

Our baseline is a non-blocking implementation of the STM design from Section 4 built directly from CAS. This incorporates a number of low-level optimisations which are not present in the published algorithms. Instead of being a single table, the orecs are split into page-sized chunks with a main table giving the address of each chunk. This lets chunks be distributed throughout the memory on a ccNUMA machine in order to reduce contention in the interconnect. We also use a 'second chance' commit operation for read-only transactions: a read-only transaction can commit if all of the locations accessed still contain the values seen, even if the version numbers seen are no longer current. We compare this baseline against the equivalent STM built using revocable on each transaction descriptor.

We use a 106-processor ccNUMA SunFire e15k machine and perform experimental runs with 1..96 processors on an otherwise unloaded system. For workloads using small numbers of processors we confirmed that the results from this machine were consistent with those from a 4-way SMP system using the same processor family. In our tests we measure the CPU time required for each operation on a shared data structure and present median-of-five results with error bars indicating the minimum and maximum results seen.

We use two synthetic benchmarks built by implementing red-black trees and skip lists over the word-based transactional memory interface. A specified number of threads loop performing insert, delete and lookup operations on the tree. We can produce various forms of contention by varying (*i*) the number of active threads, (*ii*) the proportion of updates versus reads, (*iii*) the range of key values used.

The `HRHold` operation was configured to spin up to 1000 times before attempting revocation. As Figure 6 indicates, this was sufficient to avoid almost all revocations. Notice that, unlike two-phase locking with mutexes, our revocable locks are held *only when committing a transaction*, not throughout the transaction's execution. Furthermore, two threads only contend for a revocable lock when they attempt to access the same STM orec concurrently: this means that non-conflicting commit operations will usually not contend with one another.

Figure 7 compares the performance of the streamlined STM with our original design when performing red-black tree operations. The simplifications to the fast-path code for uncontended updates reduce the mean time taken to perform a tree update by over 30% on a single-threaded workload. This speed-up remains typical on workloads with low contention, for instance when performing operations with a key space $0 \ldots 2^{20}$. The STM based on hold-release scales better under higher contention than the original scheme – with a key space $0 \ldots 2^{10}$ at most 12% of commit operations fail when using hold-release, compared with over 18% when using CAS directly.

Figure 8 presents similar results from a skip-list implementation over the two STM designs.

6. RELATED WORK

Herlihy and Moss first introduced the concept of a *transactional memory* [17]. Their hardware design builds on existing multiprocessor cache-coherency mechanisms to buffer accesses within a private *transactional cache*, the contents of which are exposed to other CPUs and written back to main memory at the end of a successful transaction.

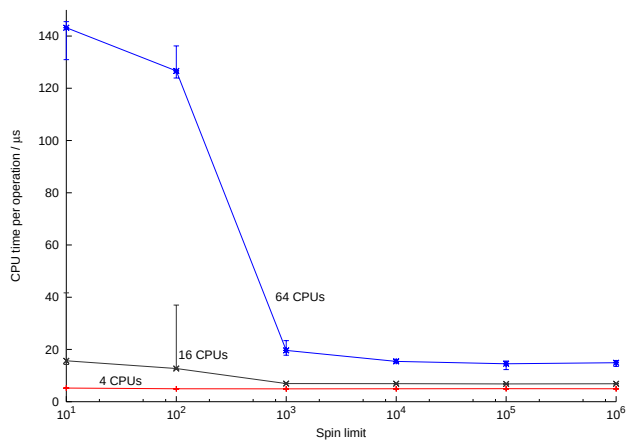
Rajwar and Goodman explore similar implementation techniques for automatically executing lock-based operations using hardware transactions [25]. As with their earlier work on speculative lock elision, they suggest that the processor can identify operations that are likely to be implementing locks [24]. This allows existing lock-based code to be executed.

These hardware schemes have the potential to allow very fast commit operations. They also allow direct sharing of locations between transactional and non-transactional access. However, they inevitably impose limits on the number of locations which can be buffered within the CPU.

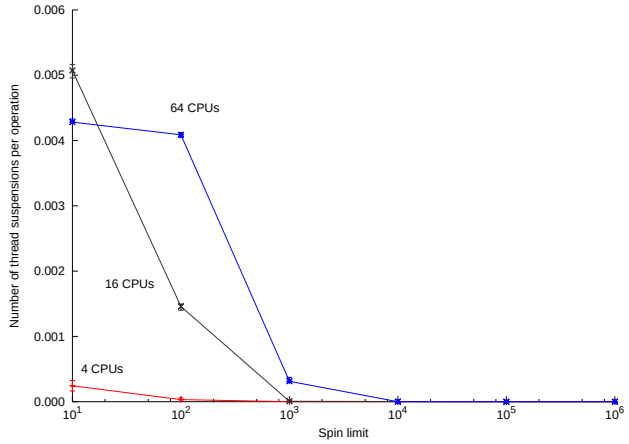
Shavit and Touitou proposed the first software-based non-blocking transactional memory [26]. As with other early designs, it used nesting LL/SC operations. These are not directly supported by hardware and possible implementations, using basic LL/SC or CAS, operate by reserving per-processor 'valid' bits or counters in each word [22] or by having several per-processor data structures for each word in the heap [18]. These space costs make the design impractical for general use.

Recently a number of practical STMs have been developed directly from CAS. As well as the *word-based* interface studied here, other researchers have investigated *object-based* STMs in which transactions 'open' the objects that they are acting on and are provided with a private copy which they then access directly. Usually each object is implemented with an additional level of indirection from an object header which points to the current contents of the object: a commit operation updates the object headers in a way that atomically installs the transaction's updates as the current versions of the objects. Herlihy *et al* designed an obstruction-free object-based STM [16]. Fraser produced a lock-free design [6]. Revocable locks could be applied to either design: as in Section 4.2, one lock would be associated with each transaction descriptor.

Other researchers have investigated using operating system support to help the design of non-blocking systems.

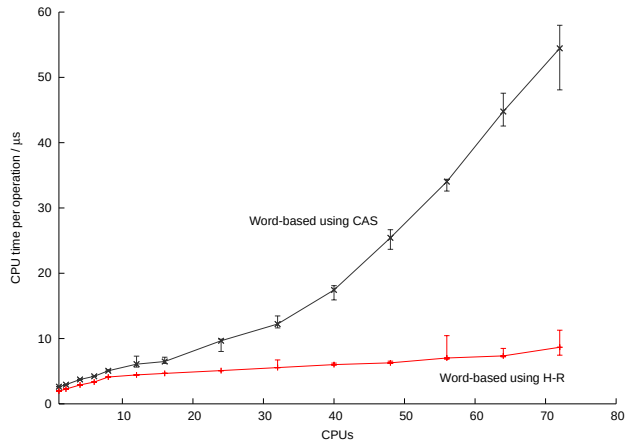


(*a*) Operation duration

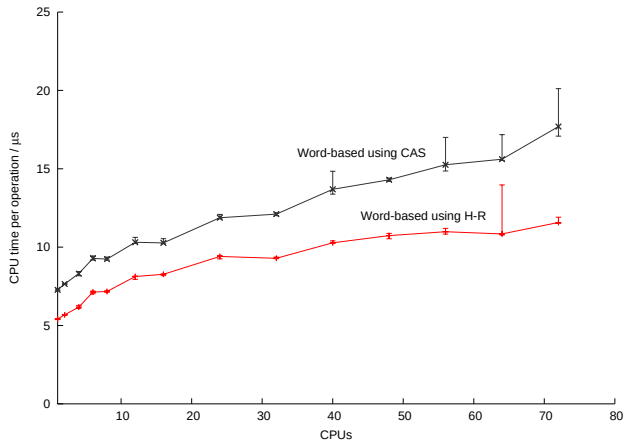


(*b*) Number of thread suspensions

Figure 6: The impact of delaying a thread before attempting to revoke a held location. Threads execute search, insert and delete operations on a skip list with keys uniformly picked from $0 \ldots 2^{10}$ and 75% read-only operations.



(*a*) Key space $0 \ldots 2^{10}$



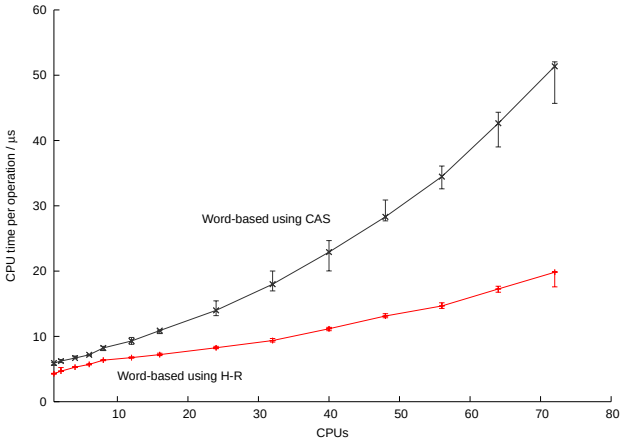
(*b*) Key space $0 \ldots 2^{20}$

Figure 7: Red-black tree performance using an STM built directly from CAS (top lines) and one built using hold-release (lower lines) performing operations on a red-black tree. 75% of operations were read-only.

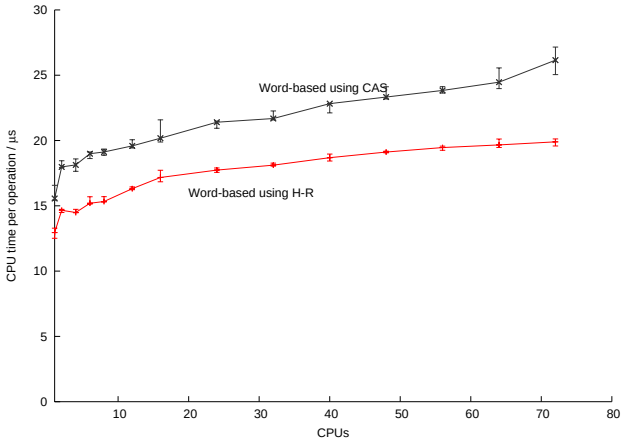
Bershad describes how CAS can be implemented on systems which lack it as a native operation [3]. On a uniprocessor system the OS can inspect the state of the previous process when switching from it and determine if it was executing within a special library function that implements CAS. The operation is rolled forward if the shared location has been updated. Otherwise it is rolled back. This uniprocessor scheme can, of course, be generalized to other operations, as Greenwald and Cheriton do in their software implementation of DCAS [9].

Alemany and Felten describe how the OS can help maintain a count of 'in progress' operations which the scheduler reduces after preempting a thread performing an operation on a shared data structure [2]. Threads use this count to avoid contending with other threads which are actively working on the same structure. They also describe a roll-back scheme in which each process builds a change log which the OS can use to restore the shared structure if the process is preempted before completing its update.

Implementation schemes based on revocation notifications or thread suspension have been used in a number of systems.



(*a*) Key space $0 \ldots 2^{10}$



(*b*) Key space $0 \ldots 2^{20}$

Figure 8: Skip-list performance using an STM built directly from CAS (top lines) and one built using hold-release (lower lines) performing operations on a skip-list. 75% of operations were read-only.

Dice and Garthwaite consider the problem of controlling access to state such as memory allocation meta-data that is CPU-local (rather than thread-local). They introduce multiprocessor restartable critical sections during which a thread receives an up-call if it is preempted or migrated to another CPU [5].

Burrows describes a scheme for implementing fast-path operations in a mutex implementation by using thread suspension and controlled roll-forward [4]. Kawachiya *et al* used the same underlying technique as our revocable locks in their implementation of mutexes for Java [19]. They allow each Java mutex to be *reserved* by a thread and use thread suspension and displacement when one thread wishes to acquire a lock reserved by another.

Pizlo *et al* implemented implemented atomic transactional methods for real-time Java [23], logging the values overwritten when executing inside a transactional method and restoring these if the transaction is aborted. Their implementation allowed at most one active transaction at any time – a subsequent transaction would abort an ongoing one by causing an exception to be raised in it. Revocable locks might provide a way of extending this infrastructure to support multiple ongoing transactions – although care would be needed if the analyzability necessary for real-time performance is to be retained.

Welc *et al* used a roll-back mechanism to allow locks to be preempted from Java threads [27]. Preemption is transparent to the programmer – the updates made within a synchronized block are rolled back and execution of the thread resumes at the start of the block. The implementation of roll-back is simpler than with a non-blocking transactional memory because the program is written in ordinary Java using locks rather than using optimistic concurrency control.

In distributed systems, *leasing* can be employed to avoid bad interactions between mutual exclusion and failures [7]: other processes can be certain that a lease has expired once sufficient time has elapsed. The benefits of our revocable locks have a similar feel, except at shorter timescales and using termination that is explicit rather than implicit.

7. CONCLUSIONS AND FUTURE WORK

In this paper we have shown how to build a form of revocable lock which simplifies the design of many non-blocking data structures. The key novelty in the design is to expose lock revocation by displacing the previous holder to a safe location, avoiding a class of problem relating to delayed operations.

When applied to Greenwald's scheme of *two-handed emulation* it allows CASs to be used in place of DCAS. When applied to our *word-based STM* it simplifies the management of temporary data structures, allows their size to be bounded by the number of locations accessed in a single active transaction and reduces the size of the ownership records used to co-ordinate transactions.

We originally considered whether the hold-release operations would be suitable for implementation in hardware: with the usual MESI cache coherence protocol, a held location would have to be retained in modified or exclusive mode and revocation would be triggered if it was invalidated. However, at least in the algorithms that we have studied, the short durations for which locations are held seem to make software implementations sufficient when coupled with a brief period of spinning before performing revocation.

8. REFERENCES

[1] *Motorola M68000 Family Programmer's Reference Manual*. Motorola Inc, 1992.

[2] Alemany, J., and Felten, E. W. Performance issues in non-blocking synchronization on shared-memory multiprocessors. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing* (Aug. 1992), ACM Press, pp. 125–134.

[3] Bershad, B. N. Practical considerations for non-blocking concurrent objects. Technical Report CMU-CS-91-116, Carnegie Mellon University, School of Computer Science, Oct. 1991.

[4] Burrows, M. How to implement unnecessary mutexes. In *Computer Systems: Theory, Technology and Applications* (Dec. 2003), Springer-Verlag.

[5] Dice, D., and Garthwaite, A. Mostly lock-free malloc. In *Proceedings of the third international symposium on Memory management* (2002), ACM Press, pp. 163–174.

[6] Fraser, K. *Practical lock freedom*. PhD thesis, University of Cambridge Computer Laboratory, 2003.

[7] Gray, C., and Cheriton, D. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the twelfth ACM symposium on Operating systems principles* (1989), ACM Press, pp. 202–210.

[8] Greenwald, M. Two-handed emulation: how to build non-blocking implementations of complex data-structures using DCAS. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing (PODC-02)* (July 2002), ACM Press, pp. 260–269.

[9] Greenwald, M., and Cheriton, D. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI '96)* (Oct. 1996), pp. 123–136.

[10] Harris, T. Exceptions and side-effects in atomic blocks. In *Proceedings of the 2004 Workshop on Concurrency and Synchronization in Java programs* (July 2004), pp. 46–53. Proceedings published as Memorial University of Newfoundland CS Technical Report 2004-01.

[11] Harris, T., and Fraser, K. Language support for lightweight transactions. In *Object-Oriented Programming, Systems, Langauges & Applications (OOPSLA '03)* (Oct. 2003), pp. 388–402.

[12] 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.

[13] Herlihy, M. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems 15*, 5 (Nov. 1993), 745–770.

[14] 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.

[15] Herlihy, M., Luchangco, V., and Moir, M. Obstruction-free synchronization: Double-ended queues as an example. In *23rd International Conference on Distributed Computing Systems (ICDCS'03)* (May 2003).

[16] Herlihy, M., Luchangco, V., Moir, M., and Scherer, III, W. N. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual ACM Symposium on Principles of distributed computing* (2003), ACM Press, pp. 92–101.

[17] 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.

[18] Jayanti, P., and Petrovic, S. Efficient and practical constructions of ll/sc variables. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing* (2003), ACM Press, pp. 285–294.

[19] Kawachiya, K., Koseki, A., and Onodera, T. Lock reservation: Java locks can mostly do without atomic operations. In *OOPSLA* (2002), pp. 130–141.

[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. ABA prevention using single-word instructions. Tech. Rep. RC-23089, IBM Research Division, Jan. 2004.

[22] Moir, M. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing* (Aug. 1997), pp. 219–228.

[23] Pizlo, F., Prochazka, M., Jagannathan, S., and Vitek, J. Transactional lock-free objects for real-time Java. In *Proceedings of the 2004 PODC Workshop on Concurrency and Synchronization in Java Programs* (July 2004).

[24] Rajwar, R., and Goodman, J. R. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual International Symposium on Microarchitecture* (Dec. 2001), IEEE Computer Society TC-MICRO and ACM SIGMICRO, pp. 294–305.

[25] Rajwar, R., and Goodman, J. R. Transactional lock-free execution of lock-based programs. *ACM SIGPLAN Notices 37*, 10 (Oct. 2002), 5–17.

[26] Shavit, N., and Touitou, D. Software transactional memory. *Distributed Computing, Special Issue 10*, 2 (1997), 99–116.

[27] Welc, A., Hosking, A. L., and Jagannathan, S. Preemption-based avoidance of priority inversion for java. In *Proceedings of the 2004 International Conference on Parallel Processing (ICPP)* (Aug. 2004), pp. 529–538.