Concurrent Programming Without Locks

KEIR FRASER and TIM HARRIS
University of Cambridge Computer Laboratory

Mutual exclusion locks remain the *de facto* mechanism for concurrency control on shared-memory data structures. However, their apparent simplicity is deceptive: it is hard to design scalable locking strategies because they can harbour problems such as deadlock, priority inversion and convoying. Furthermore, scalable lock-based systems are not readily composable when building compound operations that span multiple structures. In looking for solutions to these problems, interest has developed in *non-blocking* systems which have promised scalability and robustness by eschewing mutual exclusion while still ensuring safety.

In this paper we present three abstractions which make it easier to develop non-blocking implementations of arbitrary data structures. The first abstraction is a *multi-word compare-and-swap* (MCAS) operation which atomically updates a set of memory locations. This can be used to advance a data structure from one consistent state to a another. The second abstraction is a *word-based software transactional memory* (WSTM) which can allow sequential code to be reused more directly than with MCAS and which provides better scalability when locations are being read rather than being updated. The third abstraction is an *object-based software transactional memory* (OSTM). This abstraction allows more predictable performance than WSTM and a more streamlined implementation at the cost of re-engineering the data structure to use OSTM objects.

We present practical implementations of all three of these abstractions, built from operations available across all the major CPU families used in contemporary parallel hardware. We illustrate the use of these abstractions by outlining how highly concurrent skip-lists and red-black trees can be built over them and we compare the performance of the resulting implementations against one another and against high-performance lock-based systems. These results demonstrate that it is possible to build useful non-blocking data structures with performance comparable to or better than sophisticated lock-based designs. Furthermore, and contrary to widespread belief, this work shows that existing hardware primitives are sufficient to build these practical lock-free implementations of complex data structures.

Categories and Subject Descriptors: D.4.1 [**Operating Systems**]: Process Management—*Concurrency*; *Mutual Exclusion*; *Synchronization*

General Terms: Algorithms, Languages, Performance

Additional Key Words and Phrases: Concurrency, lock-free systems, transactional memory

1. INTRODUCTION

Mutual-exclusion locks are one of the most widely used and fundamental abstractions for synchronisation. This popularity is largely due to their apparently simple programming model and the availability of implementations which are efficient and scalable. Unfortunately, without specialist programming care, these virtues rarely hold for systems containing more than a handful of locks:

— For correctness, programmers must ensure that threads hold the necessary locks to avoid conflicting operations being executed concurrently. To avoid mistakes, this favours the development of simple locking strategies which pessimistically serialise non-conflicting operations.

— For liveness, programmers must be careful to avoid introducing deadlock, possibly causing software to hold locks for longer than would otherwise be necessary. Without scheduler support, programmers must also be aware of priority inversion.

— For high performance, programmers must balance the granularity at which locking operates against the time that the application will spend acquiring and releasing locks.

This paper is concerned with the design and implementation of software which is safe for use on multi-threaded multi-processor shared-memory machines but which does not involve the use of locking. Instead, we present three different abstractions for making atomic updates across a set of words. These enable the direct development of concurrent data structures from sequential implementations which, we believe, reduces the risk of programmers making mistakes when building multi-threaded systems.

To introduce these techniques we shall sketch their use to insert items into a singly-linked list holding integers in ascending order. In each case the list is structured with sentinel head and tail nodes whose keys are respectively less than and greater than all other values. Each node's key remains constant after insertion. In each of our examples, the insert operation proceeds by identifying nodes prev and curr between which the new node is to be placed. For comparison Figure 1 shows the corresponding insert operation when implemented using a single mutual exclusion lock to protect the entire list.

Our three alternative abstractions all follow a common style in which the core sequential code is wrapped in a loop which retries the insertion until it succeeds in committing the updates to memory. The first abstraction provides *multi-word compare-and-swap* (MCAS) which generalises the single-word CAS operation found on many processors; it atomically updates one or more memory locations from a set of expected values to a set of new values [Harris et al. 2002]. Figure 2 shows how the list-insert operation could be expressed using MCAS. The changes from sequential code are that operations reading from any words accessed by MCAS must use an MCASRead function and that proposed updates are grouped together and supplied to an invocation of MCAS: here there is only a single update to be made, as in linked-list implementations built directly from CAS [Harris 2001; Michael 2002a]. A delete operation would pass two updates to MCAS: one to excise the node from the list and a second to clear its next field to NULL to prevent concurrent insertion after a deleted node.

```
1  typedef struct { int key; struct node *next; } node;
   typedef struct { mutex lock; node *head; } list;
3  void list_insert_locked (list *l, int k) {
     mutex_lock(l→lock);
5    node *n := new node(k);
     node *prev := l→head;
7    node *curr := prev→next;
     while ( curr→key < k ) {
9      prev := curr;
       curr := curr→next;
11   }
     n→next := curr;
13   prev→next := n;
     mutex_unlock(l→lock);
15 }
```

Fig. 1. Insertion into a sorted list protected by a mutex.

```
1  typedef struct { int key; struct node *next; } node;
   typedef struct { node *head; } list;
3  void list_insert_mcas (list *l, int k) {
     node *n := new node(k);
5    do {
       node *prev := MCASRead( &(l→head) );
7      node *curr := MCASRead( &(prev→next) );
       while ( curr→key < k ) {
9        prev := curr;
         curr := MCASRead( &(curr→next) );
11     }
       n→next := curr;
13   } while ( ¬MCAS (1, [&prev→next], [curr], [n]) );
   }
```

Fig. 2. Insertion into a sorted list managed using MCAS.

The second abstraction provides a *word-based software transactional memory* (WSTM) which allows a series of read and write operations performed by a thread to be grouped as a software transaction and applied to the heap atomically [Harris and Fraser 2003]. Here, in Figure 3, the changes from sequential code are that read and write operations are performed through WSTMRead and WSTMWrite functions and that this whole set of updates is wrapped in a pair of WSTMStartTransaction and WSTMCommitTransaction operations.

The third abstraction provides an *object-based software transactional memory* (OSTM) which allows a thread to 'open' a set of objects for transactional accesses and, once more, to commit updates to them atomically [Fraser 2003]. Figure 4 illustrates this style of programming: each object is accessed through an *OSTM handle* which must be subject to an OSTMOpenForReading or OSTMOpenForWriting call in order to obtain access to the underlying data.

While these techniques do not provide a silver-bullet to designing scalable concurrent data structures they represent a shift of responsibility away from the program-

```
1  typedef struct { int key; struct node *next; } node;
   typedef struct { node *head; } list;
3  void list_insert_wstm (list *l, int k) {
     node *n := new node(k);
5    do {
       wstm_transaction *tx := WSTMStartTransaction();
7      node *prev := WSTMRead(tx, &(l→head));
       node *curr := WSTMRead(tx, &(prev→next));
9      while ( curr→key < k ) {
         prev := curr;
11       curr := WSTMRead(tx, &(curr→next));
       }
13     n→next := curr;
       WSTMWrite(t, &(prev→next), n);
15   } while ( ¬WSTMCommitTransaction(tx) );
   }
```

Fig. 3. Insertion into a sorted list managed using WSTM.

```
1  typedef struct { int key; ostm_handle<node*> *next; } node;
   typedef struct { ostm_handle<node*> *head; } list;
3  void list_insert (list *l, int k) {
     node *n := new node(k);
5    ostm_handle<node*> := new ostm_handle(n);
     do {
7      ostm_transaction *tx := OSTMStartTransaction();
       ostm_handle<node*> *prev_obj := l→head;
9      node *prev := OSTMOpenForReading(tx, prev_obj);
       ostm_handle<node*> curr_obj := prev→next;
11     node *curr := OSTMOpenForReading(tx, curr_obj);
       while ( curr→key < k ) {
13       prev_obj := curr_obj;       prev := curr;
         cuur_obj := prev → next; curr := OSTMOpenForReading(tx, curr_obj);
15     }
       n→next := curr_obj;
17     prev := OSTMOpenForWriting(tx, prev_obj);
       prev→next := n;
19   } while ( ¬OSTMCommitTransaction(tx) );
   }
```

Fig. 4. Insertion into a sorted list managed using WSTM.

mer: the abstraction's implementation is responsible for correctly ensuring that conflicting operations do not proceed concurrently and for preventing deadlock and priority-inversion between concurrent operations. The programmer remains responsible for ensuring scalability by making it unlikely that concurrent operations will need to modify the same words. However, this is a performance problem rather than a correctness or liveness one and, in our experience, even straightforward data structures, developed directly from sequential code, offer performance that competes with and often surpasses state-of-the-art lock-based designs.

1.1 Goals

We set ourselves a number of goals in order to ensure that our designs are practical and perform well when compared with lock-based schemes:

Concreteness. We must consider the full implementation path down to the instructions available on commodity CPUs. This means we build from atomic single-word read, write and compare-and-swap (CAS) operations.

Linearizability. In order for functions such as MCAS and OSTMCommitTransaction to behave as expected in a concurrent environment we require that their implementations be linearizable, meaning that they appear to occur atomically at some point between when they are invoked and when they return [Herlihy and Wing 1990].

Non-blocking behaviour. In order to provide robustness against liveness problems such as deadlock our abstractions should be non-blocking. This means that even if any set of threads is stalled then the remaining threads can still make progress.

Disjoint-access parallelism. Our abstractions should avoid introducing contention in the sets of memory locations they access: operations which access disjoint sets of words in memory must be able to execute in parallel.

Read parallelism. Our abstractions should preserve the ability for sets of operations performing read-only accesses to intersecting sets of memory locations to execute in parallel. This is ordinarily provided by popular MESI cache protocols; we must avoid introducing conflicting updates when shared locations are read by multiple threads.

Practicable space costs. Storage costs should scale well with the number of threads and the volume of data managed using the abstraction. It is generally unacceptable to reserve more than two bits in each word (often such bits are always zero if locations hold aligned pointers) and it is desirable to avoid doing even that if words are to hold general word-sized values.

Composability. If multiple data structures separately provide operations built with one of our abstractions then these should be composeable to form a single compound operation which occurs atomically (and which can itself be composed with others).

Our MCAS implementation is concrete, linearizable, non-blocking and disjoint-access parallel. It is not read parallel. It reserves two bits in every location that may be accessed using MCAS. It has no fixed storage costs. MCAS operations are not composable.

Our WSTM implementation is concrete, linearizable, non-blocking and, if hashed values of the addresses access by concurrent threads are distinct, it is disjoint-access parallel and read parallel. It does not reserve any space in locations being accessed. It has a single fixed-size table whose size can be tuned to control the likelihood of operations proceeding in parallel. WSTM operations are composable.

Our OSTM implementation is concrete, linearizable, non-blocking and, if threads access different objects, it is disjoint-access parallel and read parallel. It requires one machine word per object, but does not reserve any space in locations being accessed and has no fixed storage costs. OSTM operations are composable.

We also have a number of non-goals: (*i*) although these APIs can be used concurrently in the same application, we do not intend that they be used to manage parts of the same data structure, (*ii*) we defer the problem of storage management of application data to automatic garbage collection, or to schemes such as Herlihy et al's [2002], or Michael's [2002b], or to limbo-lists [2003], and, (*iii*) where a system exhibits high contention we assume that separate contention management will be employed, for instance using a plug-in contention manager of the kind Scherer and Scott describe [2004].

1.2 Source code availability

Source code for our MCAS, WSTM and OSTM systems, data structure implementations and test harnesses is available for Alpha, Intel IA-32, Intel IA-64, PowerPC and SPARC processor families at `http://www.cl.cam.ac.uk/netos/lock-free`.

1.3 Structure of this paper

In Section 2 we present the interface to the three alternative abstractions and compare and contrast their features and the techniques for using them effectively.

We discuss previous work with respect to our goals in Section 3; in summary, previous designs have required unrealistic hardware primitives, had unrealistic storage costs or offered only lacklustre performance or scalability.

In Section 4 we describe our overall design method and the common facets of each of our designs. In Sections 5–7 we turn to the details of these three abstractions in turn and present our design, its relationship to previous work and, where applicable, to contemporary work which has had similar goals of practicability [Herlihy et al. 2003].

In Section 8 we evaluate the performance of data structures built over each of the abstractions, both in comparison with one another and in comparison with high-quality lock-based schemes. We use skip-lists and red-black trees as running examples, highlighting any particular issues that arise when adapting a sequential implementation for concurrent use. Finally, Section 9 concludes.

2. PROGRAMMING ABSTRACTIONS

In this section we present the programming interfaces for using MCAS, WSTM and OSTM. These each provide mechanisms for accessing multiple unrelated words in a single atomic step; however, they differ in the way in which those accesses are specified and the adaptation required to make a sequential operation safe for multi-threaded use.

2.1 Multi-word compare-and-swap (MCAS)

Multi-word compare-&-swap (MCAS) extends the well-known hardware CAS primitive to operate on an arbitrary number of memory locations simultaneously. As with the linked-list example in Figure 2, it is typically used by preparing a list of updates to make in a thread-private phase before invoking MCAS to apply them to the heap.

MCAS extends the single-word CAS primitive to operate on multiple locations simultaneously. More precisely, MCAS is defined to operate on N distinct memory locations (a_i), expected values (e_i), and new values (n_i): each a_i is updated to

value n_i if and only if each a_i contains the expected value e_i before the operation. MCAS returns TRUE if these updates are made and FALSE otherwise.

Heap accesses to words which may be subject to a concurrent MCAS must be performed using MCASRead operations. These provide extra flexibility to the MCAS implementation so that it need not retain data it is updating 'in the clear' as it does so. The locations being updated must hold hold aligned pointer values, allowing the implementation of MCAS to use low-order bits (which would otherwise be zero) for its own purposes. The full API is consequently:

```
1  // Update locations a[0]..a[N-1] from e[0]..e[N-1] to n[0]..n[N-1]
   bool MCAS (int N, word **a[ ], word *e[ ], word *n[ ]);
3  // Read the contents of location a
   word *MCASRead (word **a);
```

This API is effective when a small number of locations can be identified which need to be accessed to update a data structure from one consistent state to another particularly if, as in our linked-list example, the locations and values involved are directly available from local variables.

Using MCAS also allows skilled programmers to reduce contention between concurrent operations by paring down the set of locations passed to MCAS, or by decomposing a series of related operations into a series of MCAS steps. For instance, when inserting a node into a sorted linked-list, we relied on the structure of the list and the immutability of key fields to allow us to update just one location rather than needing to check that the complete chain of pointers traversed has not been modified by a concurrent thread. However, this flexibility presents a pit-fall for novice programmers.

The MCAS API also precludes our goal of composability.

2.2 Word-based software transactional memory (WSTM)

Although MCAS eases the burden of ensuring correct synchronisation of updates, many data structures also require consistency among groups of read operations and it is cumbersome for the application to track these and present them as arrays of 'no-op' updates to MCAS. For instance, consider searching within a move-to-front list, in which a successful search promotes the discovered node to the head of the list. As indicated in Figure 5, a naïve search algorithm which does not consider synchronisation with concurrent updates may incorrectly fail, even though each individual read from shared memory operates on a consistent snapshot of the list.

Software transactional memories provide a way of dealing with these problems by grouping shared-memory access into transactions which appear to succeed or fail atomically. Furthermore, composability is gained by allowing nested transactions: a series of WSTM transactions can be composed by bracketing them within a further transaction. In general, our implementation of the WSTM API allows a transaction to commit so long as no other thread has committed an update to one of the locations that has been accessed.

Within a transaction, data accesses are performed by WSTMRead and WSTMWrite operations. In previous work we have discussed how a managed run-time environment can transparently introduce calls to these for all accesses within transactions [Harris and Fraser 2003]. As with MCAS programmers are responsible for

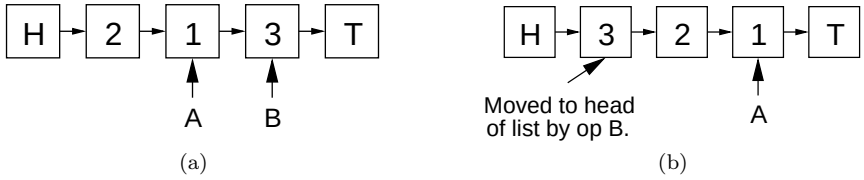


Fig. 5. The need for read consistency: a move-to-front linked list subject to two searches for node 3. In snapshot (a), search A is preempted while passing over node 1. Meanwhile, in snapshot (b), search B succeeds and moves node 3 to the head of the list. When A continues execution, it will incorrectly fail.

using these operations when accessing words which may be subject to a concurrent WSTMCommitTransaction.

Unlike MCAS, our WSTM implementation does not reserve space in each word, allowing it to act on full word-size data rather than just pointer-valued fields in which 'spare' bits can be reserved. The full API is:

```
1  // Transaction management
   wstm_transaction *WSTMStartTransaction();
3  bool WSTMCommitTransaction(wstm_transaction *tx);
   bool WSTMValidateTransaction(wstm_transaction *tx);
5  void WSTMAbortTransaction(wstm_transaction *tx);
   // Data access
7  word WSTMRead(wstm_transaction *tx, word *a);
   void WSTMWrite(wstm_transaction *tx, word *a, word d);
```

As we will show later, the easier-to-use interface often results in reduced performance compared with MCAS.

2.3 Object-based software transactional memory (OSTM)

The third API, OSTM provides an alternative transaction-based interface. As with WSTM, data managed with OSTM can hold full word-size values and transactions can nest allowing composability.

However, rather than accessing words individually, a programmer using OSTM accesses objects through a level of indirection provided by OSTM handles. OSTM objects are alloced and deallocated by OSTMNew and OSTMFree which behave analogously to the standard malloc and free functions, but act on pointers to OSTM handles rather than directly on pointers to objects.

Before the data it contains can be accessed, an OSTM handle must be *opened* in order to obtain a private copy of the underlying object – this is done by OSTMOpenForReading and OSTMOpenForWriting which take handles of type ostm_handle<t*> and return object pointers of type t* on which ordinary memory access operations can be invoked. Both of these open operations are idempotent: if the object has already been opened for the same access mode within transaction the specified transaction then the same pointer will be returned again.

The OSTM interface leads to a different cost profile from WSTM: OSTM introduces a cost of opening objects for access and potentially producing shadow copies to work on, but subsequent data access is made directly (rather than through func-

tions like WSTMRead and WSTMWrite) and it admits a simplified non-blocking commit operation. The programmer must be careful not to update objects that it has opened in read-only mode because these may be shared with other threads – violations of this requirement can, of course, be checked at commit-time in a debugging mode.

The OSTM API is therefore:

```
1  // Transaction management
   ostm_transaction *OSTMStartTransaction();
3  bool OSTMCommitTransaction(ostm_transaction *tx);
   bool OSTMValidateTransaction(ostm_transaction *tx);
5  void OSTMAbortTransaction(ostm_transaction *tx);
   // Data access
7  t *OSTMOpenForReading(ostm_transaction *tx, ostm_handle<t*> *o);
   t *OSTMOpenForWriting(ostm_transaction *tx, ostm_handle<t*> *o);
9  // Storage management
   ostm_handle<void*> *OSTMNew(size_t size);
11 void OSTMFree(ostm_handle<void*> *ptr);
```

3. RELATED WORK

The literature contains several designs for abstractions similar to MCAS, WSTM and OSTM. However, these have generally not shared our goals of practicality – for instance much work builds on instructions such as DCAS or strong-LL/SC which are not available as primitives. Our experience is that although this foundational work has highlighted the problems which exist and has introduced terminology and conventions for presenting and reasoning about algorithms, it has not been possible to effectively extend these algorithms by layering them above software implementations of DCAS or strong-LL/SC.

This section is split into three parts. Firstly, in Section 3.1 we introduce the terminology of *non-blocking* systems and the progress guarantees that they make. These properties underpin the liveness guarantees that are provided to users of our algorithms. Secondly, in Section 3.2 we discuss the design of 'universal' transformations that build non-blocking systems from sequential code or from lock-based code. Finally, in Section 3.3, we present previous implementations of multi-word operations such as MCAS, WSTM and OSTM and we assess them against our goals.

3.1 Non-blocking systems

Non-blocking algorithms have been studied as a way of avoiding the liveness problems that are possible when using traditional locks [Herlihy 1993]. A design 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 locks because, unless a lock-holder continues to run, the lock can never be released.

Non-blocking algorithms can be classified according to the kind of progress guarantee that they make:

— *Obstruction-freedom* is the weakest form of guarantee: a thread performing an operation on the data structure is only guaranteed to make progress so long as it does not contend with other threads for access to any location [Herlihy et al. 2003]. This requires an out-of-band mechanism to avoid livelock; exponential backoff is one option.

— *Lock-freedom* adds the requirement that the system as a whole makes progress, even if there is contention. In some cases, lock-free algorithms can be developed from obstruction-free ones by adding a *helping* mechanism: if thread `A` encounters thread `B` obstructing it then `A` helps `B` to complete `B`'s operation. This is sufficient to prevent livelock, although it does not offer any guarantee of per-thread fairness.

— *Wait-freedom* adds the requirement that *every thread* makes progress, even if it experiences contention. This gives a hard bound on the number of instructions that need to be executed to perform any operation on the data structure. However, it is seldom possible to develop wait-free algorithms from lock-free ones and, where wait-free algorithms exist, they rarely offer competitive practical performance.

Some previous work has used the terms *lock-free* and *non-blocking* interchangeably: we follow Herlihy et al's recent usage in using lock-freedom to denote a particular kind of non-blocking behaviour [Herlihy et al. 2003]. In this paper we concentrate on lock-free algorithms, although we highlight where simplifications can be made to our implementations by designing them to satisfy the weaker requirement of obstruction freedom.

3.2 Universal constructions

Universal constructions are a class of design technique that can straightforwardly transform a sequential program to make it safe for concurrent execution. Herlihy describes a universal construction for automatically creating a non-blocking algorithm from a sequential specification [Herlihy 1993]. This requires a snapshot of the entire data object to be copied to a private location where shadow updates can safely be applied: these updates become visible when the single 'root' pointer of the structure is atomically checked and modified to point at the shadow location. This means that concurrent updates will always conflict, even when they modify disjoint sections of the data structure.

Turek *et al* devised a hybrid scheme that may be applied to develop lock-free systems from deadlock-free lock-based ones [Turek et al. 1992]. Each lock in the unmodified algorithm is replaced by an ownership reference which is either *nil* or points to a continuation describing the sequence of *virtual instructions* that remain to be executed by the lock 'owner'. This allows conflicting operations to execute these instructions on behalf of the owner and then take ownership themselves, rather than blocking on the original lock. Interpreting a continuation is cumbersome: after each 'instruction' is executed, a virtual program counter and a non-wrapping version counter are atomically modified using a double-width CAS operation which acts on an adjacent pair of memory locations.

Barnes proposes a similar technique in which mutual-exclusion locks are replaced by *operation descriptors* [Barnes 1993]. Lock-based algorithms are converted to operate on a private copy of the data structure; then, after determining the sequence of updates to apply, each required operation record is acquired in turn, the updates

are performed, and finally the operation records are released. Copying is avoided if contention is low by observing that the private copy of the data structure may be cached and reused across a sequence of operations. This two-phase algorithm requires strong LL/SC operations.

3.3 Programming abstractions

Although universal constructions have the benefit of requiring no manual modification to existing sequential or lock-based programs, each exhibits some substantial performance or implementation problems which places it beyond practical use. Another class of techniques provides high-level programming abstractions which, although not automatic 'fixes' to the problem of constructing non-blocking algorithms, make the task of implementing non-blocking data structures much easier compared with using atomic hardware primitives directly. The two best-known abstractions are multi-word compare-&-swap (MCAS) and forms of software transactional memory (STM).

Israeli and Rappaport described the first design which builds a lock-free MCAS from strong LL/SC primitives [Israeli and Rappoport 1994]. For N threads, their method for building the required LL/SC from CAS reserves N bits within each updated memory location; the MCAS algorithm then proceeds by load-locking each location in turn, and then attempting to conditionally-store each new value in turn. The cost of implementing the required strong LL/SC makes their design impractical.

Anderson and Moir designed a wait-free version of MCAS that also requires strong LL/SC [Anderson and Moir 1995], although they devised a method for constructing strong-LL/SC using $\log N$ reserved bits per updated memory location. This bound is achieved at the cost of considerable bookkeeping to ensure that version numbers are not reused. A further drawback is that the accompanying *MCASRead* operation is based on primitives that acquire exclusive cache-line access for the location, preventing read parallelism.

Moir developed a stream-lined version of this algorithm which provides 'conditionally wait-free' semantics [Moir 1997]. Specifically, the design is lock-free but an out-of-band helping mechanism may be specified which is then responsible for helping conflicting operations to complete. This design suffers many of the same weaknesses as its ancestor; in particular, it requires strong-LL/SC and a non-read-parallel *MCASRead*.

Anderson *et al* provide further versions of MCAS suitable for systems using strict priority scheduling [Anderson et al. 1997]. Both algorithms store a considerable amount of information in memory locations subject to MCAS updates: a valid bit, a process identifier ($\log N$ bits), and a 'count' field (which grows with the base-2 logarithm of the maximum number of addresses specified in an MCAS operation). Furthermore, their multiprocessor algorithm requires certain critical sections to be executed with preemption disabled, which is not feasible in many systems.

Greenwald presents a simple MCAS design in his PhD dissertation [Greenwald 1999], which constructs a record describing the entire operation and installs it into a single shared location which indicates the sole in-progress MCAS operation. If installation is prevented by an already-running MCAS, then the existing operation is helped to completion and its record is then removed. Once installed, an oper-

ation proceeds by executing a DCAS operation for each location specified by the operation: one update is applied to the *address* concerned, while the other updates a progress counter in the operation record. Note that Greenwald's design is not disjoint-access parallel, and that it requires DCAS. His subsequent mechanism of 'two-handed emulation' generalised this technique but did not address the lack of disjoint-access parallelism [Greenwald 2002].

Herlihy and Moss first introduced the concept of a *transactional memory*, which allows shared-memory operations to be grouped into atomic transactions [Herlihy and Moss 1993]. They originally proposed a hardware design which leverages existing multiprocessor cache-coherency mechanisms. Rajwar and Goodman have subsequently suggested similar techniques for speculatively executing lock-based code [Rajwar and Goodman 2002; 2001]. The major practical drawback of these designs is that, even if they were to be implemented in practice, hardware would impose limits on the volume of data that could be subject to a single transactional access: software mechanisms, such as those that we have investigated, would be necessary when these limits are breached.

Shavit and Touitou proposed a software-based lock-free transactional memory built from strong-LL/SC [Shavit and Touitou 1995]. A notable feature is that they abort contending transactions rather than recursively helping them, as is usual in lock-free algorithms; non-blocking behaviour is still guaranteed because aborted transactions help the transaction that aborted them before retrying. Their design supports only 'static' transactions, in which the set of accessed memory locations is known in advance — the interface is therefore analogous to MCAS rather than subsequent STM designs, including our own.

Moir presents lock-free and wait-free STM designs [Moir 1997] which provide a dynamic programming interface, in contrast with Shavit and Touitou's static interface. The lock-free design divides the transactional memory into fixed-size blocks which form the unit of concurrency. A header array contains a word-size entry for each block in the memory, consisting of a block identifier and a version number. Unfortunately arbitrary-sized memory words are required as there is no discussion of how to handle overflow of the version number. The design also suffers the same drawbacks as the conditionally wait-free MCAS on which it builds: bookkeeping space is statically allocated for a fixed-size heap, and the read operation is potentially expensive. Moir's wait-free STM extends his lock-free design with a higher-level helping mechanism based around a 'help' array which indicates when a process i has interfered with the progress of some other process j: in this situation i will help j within a finite number of execution steps.

Recently, Herlihy *et al* have implemented an obstruction-free STM [Herlihy et al. 2003]. It was developed concurrently and shares many of our goals. Firstly, the memory is dynamically sized: memory blocks can be created and destroyed on the fly. Secondly, an implementation is provided which builds on a readily-available form of the CAS primitive (this is at the cost of an extra pointer indirection when accessing the contents of a memory block, however). Finally, the design is disjoint-access parallel, and transactional reads do not cause writes to occur in the underlying STM implementation. These features serve to significantly decrease contention in many multiprocessor applications, and are all shared with my own lock-free STM

which I describe in the next chapter. This makes Herlihy *et al*'s design an ideal candidate for comparison in Section 8.

4. DESIGN METHOD

Our implementations of the three APIs in Sections 2.1–2.3 have to solve a set of common problems and, unsurprisingly, use a number of similar techniques.

The key problem is that of ensuring that a set of memory accesses appear to occur atomically when they are being built from machine instructions accessing separate words. We deal with this problem by decoupling the notion of a location's *physical contents* in memory from its *logical contents* when accessed through one of the APIs. The physical contents can, of course, only be updated one word at a time. However, as we shall show, we arrange that the logical contents of a set of locations can be updated atomically.

For each of the APIs there is only one operation which updates the logical contents of memory locations: MCAS, WSTMCommitTransaction and OSTMCommitTransaction. We call these operations collectively the *commit operations* and they are the main source of complexity in our designs.

For each of the APIs we present our design in a series of four steps:

(1) Define the format of the heap, the temporary data structures that are used and how an application goes about allocating and de-allocating memory for data structures that will be accessed through the API.
(2) Define the notion of logical contents in terms of these structures and show how it can be computed from a series of single-word instructions. This underpins the implementation of functions other than the commit operations. In this step we are particularly concerned with ensuring non-blocking behaviour and read-parallelism so that, for instance, two threads can perform WSTMRead operations to the same location at the same time without producing conflicts in the memory hierarchy.
(3) Show how the commit operation arranges to atomically update the logical state of a set of locations when it executes without interference from concurrent commit operations. In this stage we are particularly concerned with ensuring disjoint-access parallelism so that threads can commit updates to disjoint sets of locations at the same time.
(4) Show how contention is resolved when one commit operation's progress is impeded by a conflicting commit operation. In this step we are concerned with ensuring non-blocking behaviour so that the progress is not prevented if, for example, the thread performing the existing commit operation has been descheduled.

Before considering the details of the three different APIs we discuss the common aspects of each of these four steps in Sections 4.1–4.4.

4.1 Memory formats

All three of our implementations introduce *descriptors* which set out the 'before' and 'after' versions of the memory accesses that a particular commit operation proposes to make along with a status field, indicating how far the commit operation

has progressed. These descriptors satisfy three properties which make it easier to manage them in a concurrent system:

Firstly, descriptors are conceptually managed by garbage collection rather than being re-used directly. This means that if a thread holds a reference to a given descriptor then it can be sure that it is not re-used for another purpose. Of course, in practice, we do not mandate the use of a tracing garbage collector and can use schemes such as reference counting to encourage prompt reuse and affinity between descriptors and threads.

Secondly, once a descriptor is exposed to other threads, each of the entries relating to a memory access in a descriptor is read-only. This means that a thread can read from a series of such locations (e.g., in the case of an MCAS descriptor to read the location accessed and a value that is proposed to be written there) and be sure of receiving a consistent view of those locations.

Finally, once the outcome of a particular commit operation has been decided then the descriptor's status field remains constant: if a thread wishes to retry a commit operation, e.g. if the code in Figures 2–4 loops, then each retry uses a fresh descriptor. This means that threads reading from a descriptor and seeing that the outcome has been decided can be sure that the status field will not subsequently change.

4.2 Logical contents

Each of our API implementations uses descriptors to define the logical contents of memory locations by providing a mechanism for a descriptor to *own* a set of memory locations.

In general, when a commit operation relating to it is not in progress, then a location is unowned and it holds its logical contents directly. Otherwise, when a location is owned, the logical contents are taken from the descriptor and chosen from the 'before' and 'after' versions based on the descriptor's status field. This means that updating the status field has the effect of updating the logical contents of the whole set of locations that the descriptor owns.

We use different mechanisms for representing ownership in each design: with MCAS ownership is represented by installing a pointer to a descriptor in the location itself, limiting the range of values that an application can store in each word because it is necessary to distinguish pointers to descriptors from ordinary values. WSTM represents ownership using separate 'ownership records' associated with each location, allowing full word-size values to be used but adding complexity. OSTM manages ownership on an object-by-object basis by updating information in the object's header.

4.3 Uncontended commit operations

The commit operations themselves are each structured in three phases. A first phase *acquires* exclusive (but revocable) ownership of the locations being updated, a second *read-check* phase ensures that locations that have been read but not updated hold the values expected in them, this is followed by the *decision point* at which the outcome of the commit operation is decided and made visible to other threads through the descriptor's status field, and then the final *release* phase in which the thread relinquishes ownership of the locations being updated.

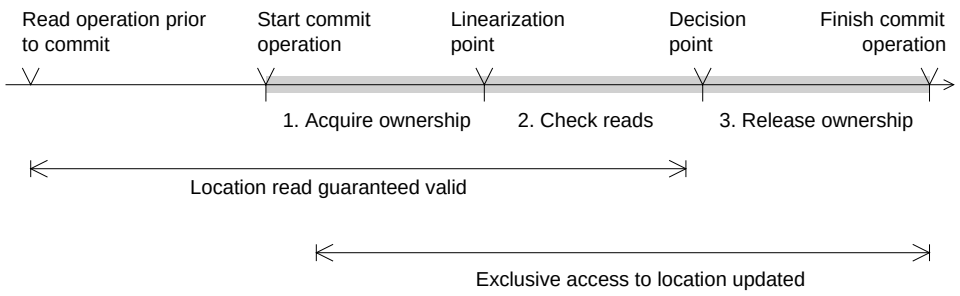


Fig. 6. Timeline for the three-phase commit operations showing the linearization point. In this example one location has been read but not updated, so its value is confirmed in the read-check phase. The gray box indicates where the commit operation is in progress.

A descriptor's status field is initially UNDECIDED at the start of a commit operation. If there is a read-check phase then the status is set to READ-CHECK for this duration. At the decision point it set to SUCCESSFUL if all of the required ownerships were acquired and the read-checks succeeded; otherwise it is set to FAILED.

In order to show that an entire commit operation is atomic we identify a *linearization point* within its execution at which it appears to operate atomically on the logical contents of the heap from the point of view of other threads. As Figure 6 shows, the linearization point occurs at the *start* of the read phase whereas the decision point, at which the outcome is actually signalled to other threads, occurs at the *end* of the read phase.

This choice of linearization point may appear perverse: how can an operation commit its updates before it has finished checking its assumptions? The rationale for this is that read-checks ensure that any locations accessed in a read-only mode have not been updated since they were read: there is nothing to prevent them from being updated after they are checked but before the decision point. At the same time, if the descriptor retains ownership of the locations being updated, then these remain under the control of this descriptor from acquisition until release. Both of these intervals straddle the linearization point.

Of course, between the linearization point and the decision point, it is not possible to determine the logical contents of locations being updated because the outcome of the commit operation has not been determined. The key insight is that this is not a problem if, when a commit operation is between its linearization point and decision point, any thread encountering its descriptor can help advance it to it decision point.

4.4 Contended commit operations

In order to achieve non-blocking behaviour we have to be careful about how to proceed when one thread t2 encounters a location that is currently owned by another thread t1. Although we aim to provide non-blocking mechanisms for contention resolution, and require some form of helping when a READ-CHECK phase is used, it is worth noting that there are, of course, a number of simpler options of how to proceed:

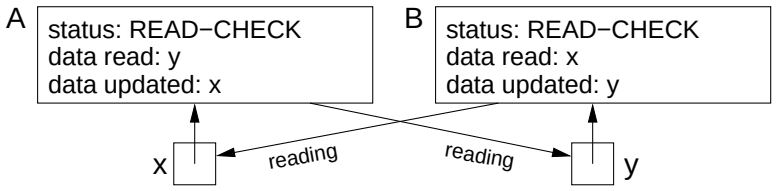


Fig. 7. An example of a dependent cycle of two transactions, A and B. Each needs the other to exit its read phase before it can complete its own.

—A basic strategy is to spin-wait until the current owner releases ownership. Note that since data is only acquired during commit operations this window of contention is likely to be small and operating system interfaces such as Solaris' schedctl can attempt to prevent pre-emption during this time.

—A further strategy is to abort the transaction the encounters the contention (t2 in this case). Although this is still not non-blocking, it may be appropriate where contention indicates that the subsequent commit is doomed to abort in any case because t1 will make a conflicting update.

These schemes perform well in practice where priority inversion or thread termination is not a problem; they can avoid deadlock by sorting the resources that they have to acquire.

If non-blocking behaviour is required then there are two general strategies:

—The first strategy is for t2 to cause t1 to abort if it has not yet reached its decision point. This leads to obstruction-free behaviour and the risk of livelock unless contention management is employed to prevent t1 retrying its operation and aborting t2.

—The second strategy is for t2 to help t1 complete its operation. This kind of *recursive helping* leads to lock-free behaviour because it ensures that thread at the head of a chain of helping will complete its commit operation.

There is one final complication when using a read-check phase and aiming for lock-free behaviour: what happens if there is a cycle of transactions, all in their read phase and each trying to read an object that is currently owned by the next? Naïvely helping transactions in their read phases risks developing an endless cycle of helping. The simple example in Figure 7 shows how this can occur.

The solution is to abort at least one of the transactions to break the cycle; however, care must be taken not to abort them all if we wish to ensure lock-freedom rather than obstruction-freedom. This can be done by imposing a total order $\prec$ on all transactions, based on the machine address of each transaction's descriptor. The loop is broken by allowing a transaction t1 to abort a transaction t2 if and only if: (*i*) both are in their read phase; (*ii*) t2 owns a location that t1 is attempting to read; (*ii*) t1 $\prec$ t2. This guarantees that every cycle will be broken, but the 'least' transaction in the cycle will continue to execute. Of course, other orderings can be used if fairness is a concern.

5. MULTI-WORD COMPARE-&-SWAP (MCAS)

We now introduce our practical design for implementing the MCAS API. MCAS extends the single-word CAS primitive to operate on multiple locations simultaneously. More precisely, MCAS is defined to operate on N distinct memory locations (a_i), expected values (e_i), and new values (n_i): each a_i is updated to value n_i if and only if each a_i contains the expected value e_i before the operation.

We initially define the implementation of MCAS using an intermediate *conditional compare-&-swap* operation. CCAS uses a second *conditional* memory location to control the execution of a normal CAS operation. If the contents of the conditional location are zero then the operation proceeds, otherwise CCAS has no effect. The conditional location may not itself be subject to updates by CCAS or MCAS. CCASRead operations must be used to read from locations that may be updated by CCAS.

```
atomically void CCAS (word **a, word *e, word *n, word *cond) {
    if ( (*a = e) ∧ (*cond = 0) ) *a := n;
}
atomically word *CCASRead (word **a) {
    return *a;
}
```

Unlike the DCAS and strong-LL/SC operations used by previous work, CCAS has a straightforward implementation using CAS; we present this in Section 5.4.

5.1 Memory formats

Each MCAS descriptor sets out the updates to be made (a set of (a_i, e_i, n_i) tuples) and the current status of the operation (UNDECIDED, FAILED, or SUCCESSFUL). In our pseudo-code we define an MCAS descriptor as:

```
1  typedef struct {
       word status;
3      int N;
       word **a[ ], *e[ ], *n[ ];
5  } MCASDesc;
```

The MCAS API can be used on data structures held in arbitrary heap locations subject, in our implementation, to the restriction that two bits of storage can be reserved in each location. In practice this means that it can act on pointer-valued data in which the pointers refer to naturally-aligned words in memory. A heap location either holds a value in the ordinary way, or it contains a pointer to an MCAS descriptor that is currently performing a commit operation on the location.

5.2 Logical contents

There are four cases to consider when defining the logical contents of a location. If the location holds an ordinary value then that is the logical contents of the location. If the location refers to an UNDECIDED descriptor then the descriptor's old value (e_i) is the location's logical contents. If the location refers to a FAILED descriptor then, once more, the old value forms the location's logical contents. If the location refers to a SUCCEEDED descriptor then the new value (n_i) is the logical contents.

```
1   word *MCASRead (word **a) {
      word *v;
3   retry_read:
      v := CCASRead(a);
5     if (IsMCASDesc(v))
        for ( int i := 0; i < v→N; i ++ )
7         if ( v→a[i] = a ) {
            if (v→status = SUCCESSFUL)
9             if (CCASRead(a) = v) return v→n[i];
            else
11            if (CCASRead(a) = v) return v→o[i];
            goto retry_read;
13        }
      return v;
15  }
```

Fig. 8. MCASRead operation used by applications to read from locations which may be subject to concurrent MCAS operations.

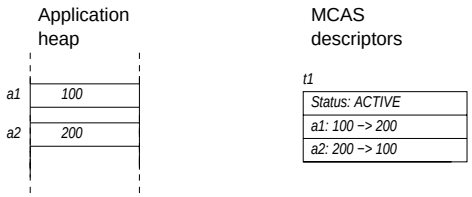
The fact that only the status field of MCAS descriptors is updated once they become reachable by threads other than their creator makes it straightforward to determine the logical contents of a location. Figure 8 presents this is pseudo-code. If the location does not refer to a descriptor then the logical contents are returned directly and this forms the linearization point of the read operation (line 4).

Otherwise, the descriptor is searched for an entry relating to the address being read (line 7) and the new value or old value returned as appropriate so long as the descriptor still owns the location. In this case the last check of the status field before returning forms the linearization point (line 8) and the re-check of ownership (line 9 or line 11) ensures that the status field was not checked 'too late' once the descriptor had lost ownership of the location (and was consequently not determining its logical contents).

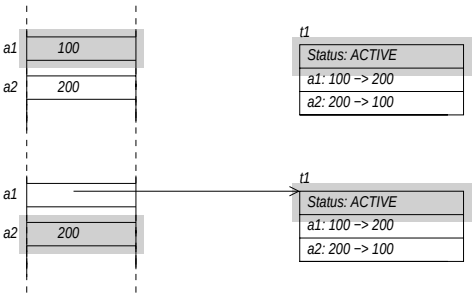
5.3 Commit operations

Figure 9 illustrates the progress of an uncontended MCAS commit operation attempting to swap the contents of addresses a1 and a2. This directly follows the design method in Section 4 aside from the simplification that it does not need a read-check phase because it does not aim to provide read-parallel behaviour: if the arrays passed to MCAS happen to specify the same value as e_i and n_i then this is treated in the same manner as an update between two values. It would be incorrect to simply check locations' values because there is no guarantee that if a series of checks succeed that there is any valid linearization point at which all of the locations simultaneously held the values seen.

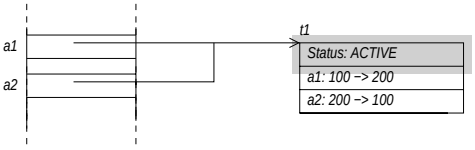
Figure 10 presents the algorithm in pseudo-code. The first phase (lines 10–19) attempts to acquire each location a_i by updating it from its expected value, e_i, to a reference to the operation's descriptor. Note that when the CCAS operation invoked on a_i must preserve the logical state of the location: either the CCAS fails (making no updates), or it succeeds, installing a reference to a the descriptor holding e_i as the old value for a_i. The 'conditional' part of CCAS ensures that the descriptor's status is still UNDECIDED, meaning that e_i is correctly defined as the logical contents of a_i.



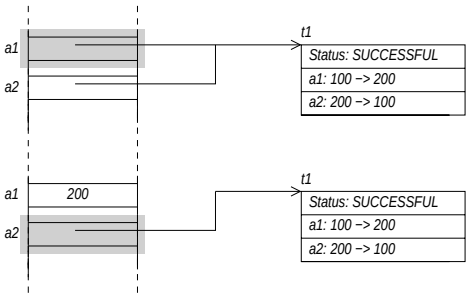
(a) The operation executes in private until it invokes MCAS. The MCAS descriptor holds the updates being proposed: in this case the contents of `a1` and `a2` are to be swapped.



(b) CCAS is used to acquire ownership of addresses `a1` and `a2`, replacing the values expected there with references to the MCAS descriptor. The updates are conditional on the descriptor remaining ACTIVE in order to guarantee that the loctions' logical contents do not change.



(c) CAS is used to set the status to `SUCCESSFUL`. This has the effect of atomically updating the locations' logical contents.



(d) Ownership is released on `a1` and `a2`, installing the new values.

Fig. 9. An uncontended commit swapping the contents of `a1` and `a2`. Grey boxes show where CAS and CCAS operations are to be performed at each step. While a location is owned, its *logical contents* remain available using the MCAS descriptor.

```
1   bool MCAS (int N, word **a[ ], word *e[ ], word *n[ ]) {
      MCASDesc *d := new MCASDesc();
3     (d→N, d→a, d→e, d→n, d→status) := (N, a, e, n, UNDECIDED);
      AddressSort(d); /* Memory locations must be sorted into address order. */
5     return MCASHelp(d);
    }
7   bool MCASHelp (MCASDesc *d) {
      word *v, desired := FAILED;
9     bool success;
      /* PHASE 1: Attempt to acquire each location in turn. */
11    for ( int i := 0; i < d→N; i++ )
        while ( TRUE ) {
13        CCAS(d→a[i], d→e[i], d, &d→status);
          v := *d→a[i];
15        if ( v = d ) break; /* move on to next location */
          if ( ¬IsMCASDesc(v) ) goto decision_point;
17        MCASHelp((MCASDesc *)v);
        }
19    desired := SUCCESSFUL;
      /* PHASE 2: No read-phase is used in MCAS */
21  decision_point:
      CAS(&d→status, UNDECIDED, desired);
23    /* PHASE 3: Release each location that we hold. */
      success := (d→status = SUCCESSFUL);
25    for ( int i := 0; i < d→N; i++ )
        CAS(d→a[i], d, success ? d→n[i] : d→e[i]);
27    return success:
    }
```

Fig. 10. MCAS operation.

Note that the algorithm must acquire update locations in address order. This ensures that recursive helping eventually results in system-wide progress because each level of recursion must be caused by a conflict at a strictly higher memory address than the previous level. To ensure that updates are ordered correctly it sorts the update locations before calling MCASHelp (lines 4–5). The sort can be omitted if the caller ensures that addresses are specified in some global total order. If addresses are not ordered then a recursive loop may be entered.

The first phase terminates when the loop has completed each location, meaning that the descriptor has been installed in each of them (line 13), or when an unexpected non-descriptor value is seen (line 16).

The first thread to reach the decision point for a descriptor must succeed in installing SUCCESSFUL or FAILED. If the MCAS has failed then the point at which an unexpected value was seen forms the linearization point of the operation: the unexpected value was the logical contents of the location and it contradicts the expected value e_i for that location. Otherwise, if the MCAS has succeeded, note that when the status field is updated (line 22) then *all* of the locations a_i must refer to the descriptor and consequently the single status update changes the logical state of all of the locations. This is because the update is made by the first thread to reach line 22 for the descriptor and so no threads can yet have reached lines 23-26 and have starting releasing the addresses.

```
1   typedef struct {
      word *a, e, n, *cond;
3   } CCASDesc;

    void CCAS (word *a, word e, word n, word *cond) {
5     CCASDesc *d := new CCASDesc();
      (d→a, d→e, d→n, d→cond) := (a, e, n, cond);
7     while ( ¬CAS(d→a, d→e, d) ) {
        word v := *d→a;
9       if ( ¬IsCCASDesc(v) ) return;
        CCASHelp((CCASDesc *)v);
11    }
      CCASHelp(d);
13  }

    word CCASRead (word *a) {
15    word v;
      for ( v := *a; IsCCASDesc(v); v := *a )
17      CCASHelp((CCASDesc *)v);
      return v;
19  }

    void CCASHelp (CCASDesc *d) {
21    bool success := (*d→cond = 0);
      CAS(d→a, d, success ? d→n : d→e);
23  }
```

Fig. 11. Conditional compare-&-swap (CCAS). CCASRead is used to read from locations which may be subject to concurrent CCAS operations.

The final phase then is to release the locations, replacing the references to the descriptor with the new or old values according to whether the MCAS has succeeded.

5.4 Building conditional compare-and-swap

The MCAS implementation is completed by considering how to provide the CCAS operation which is used for acquiring locations on behalf of a descriptor.

Figure 11 shows how CCAS can be implemented using CAS. It proceeds by installing a *CCAS descriptor* in the location to be updated (line 7). This ensures that the location's logical value is the expected value while the conditional location is tested, so that a successful CCAS operation linearises (atomically occurs) when the conditional location is read from. If the update location doesn't contain the expected value then CCAS fails (line 9); if it contains another CCAS descriptor then that operation is helped to complete before retrying (line 10).

If the update location is successfully acquired, the conditional location is tested (line 21). Depending on the contents of this location, the descriptor is either replaced with the new value, or with the original expected value (line 22). CAS is used so that this update is performed exactly once even when the CCAS operation is helped to complete by other processes.

6. WORD-BASED SOFTWARE TRANSACTIONAL MEMORY

We now turn to the word-based STM that we have developed. WSTM builds on the MCAS implementation from Section 5 by removing the requirement that space be reserved in each location in the heap and by presenting an alternative interface in

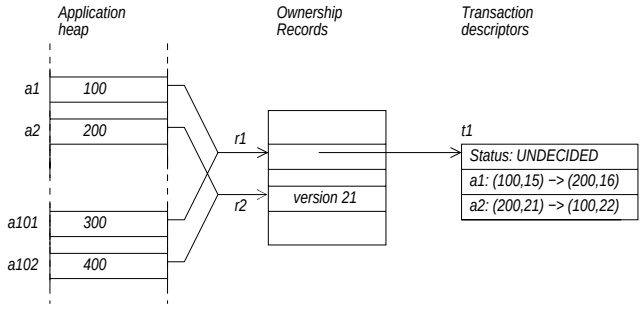


Fig. 12. The heap structure used in WSTM. The transaction set out in descriptor t1 is midway through committing an update to locations a1 and a2 which will swap over their contents. Locations a101 and a102 are examples of other locations which happen to map to the same ownership records, but which are not part of the update.

which the WSTM implementation is responsible for tracking the locations accessed. Unfortunately, the cost of this is that the WSTM system is more complex although a careful implementation performs well in common cases. This complexity motivates the straightforward OSTM system in Section 7.

6.1 Memory formats

Our WSTM implementation introduces two kinds of structure in addition to the actual data structures that WSTM is being used to access.

The first data structure is a table of *ownership records* (orecs) which are used in co-ordinating commit operations. An *ownership function* maps each address in the 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 implementation, a fixed-size table of records to which addresses map by taking a number of the significant low-order bits from the address as a hash function.

Each orec holds either a *version number* or points to 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. As with descriptors, we assume for the moment that version numbers are never re-used; we return to such matters when discussing implementation issues in Section 8.

The second kind of data structure are the *transaction descriptors* which perform an analgous role to MCAS descriptors in setting out the current status of each active transaction and the accesses that it has made to the heap. Each access is described by a *transaction entry* specifying the address in question a_i, the old and new values to be held there (o_i, n_i) and the old and new version numbers of those values (vo_i, vn_i). As usual, the status field indicates that the commit operation is either UNDECIDED, READ-CHECK, SUCCESSFUL or FAILED.

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.

Figure 12 contains an example of these two structures which we shall use for illustration. Within the transaction descriptors we indicate memory accesses using the notation $a_i{:}(o_i, vo_i) \rightarrow (n_i, vn_i)$ to indicate that address a_i is being updated from logical value o_i at version number vo_i to logical value n_i at version number vn_i. For a read-only access, $o_i == n_i$ and $vo_i == vn_i$. For an update, $vn_i == vo_i + 1$.

6.2 Logical contents

As with MCAS, we proceed by defining the logical state of a location in the heap. However, we now consider this to 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 12, the logical state of `a2` is (200, 21).

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 `a1` is (100, 15) because the descriptor shown has not yet committed and it holds an entry updating `a1` from (100,15) to (200,16).

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 SUCCESSFUL and the old version number otherwise. The 'well formed' property ensures that this is uniquely determined. For instance, the logical state of `a101` is (300, 15) taking old version 15 from the entry for `a1`.

At run time, the logical contents 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 (if any) and information from entries in that descriptor.

Fortunately, the descriptor-management properties in Section 4.1 means that 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:

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

For LS1 the value is read from the application heap – it cannot have changed if the orec's contents 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 to SUCCESSFUL or FAILED. The snapshot is consistent with the time when the status is read in the last execution of line 3.

If the descriptor is observed in a READ-CHECK state then, as discussed in Section 4.3, the operation specified in the descriptor is helped to reach its decision point at which point the logical contents can be correctly determined. We consider how helping can be implemented along with contended commit operations in Section 6.4.

Given the ability to determine the logical contents of the location, the WSTMValidateTransaction, WSTMRead and WSTMWrite operations can be implemented directly:

—Unless the transaction is already marked as FAILED, validation proceeds by checking the logical contents of all of the entries in the current transaction's descriptor: if any does not match the expected logical contents then the transaction could not commit and is consequently invalid.

—Reading proceeds by checking for an existing entry relating to the location and returning n_i if such an entry is found. Otherwise a new entry is added to the transaction record, taking care to ensure it remains well formed – the descriptor may contain existing entries relating to the same orec but which were added before the orec was updated by another transaction. If the descriptor would otherwise stop being well formed then it must be marked as FAILED. Otherwise, the logical state read forms o_i and vo_i, $n_i == o_i$ and $vn_i == vo_i$ (if the descriptor does not contain updates to locations associated with this orec) and $vn_i = vo_i + 1$ otherwise.

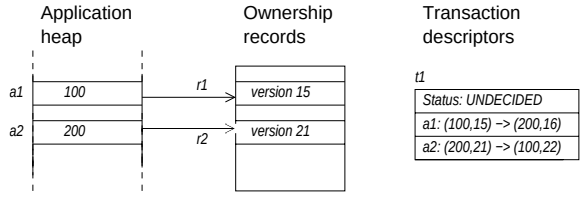
—Writing proceeds by performing a read on the location, setting the new value in the transaction entry to be the value being written and setting the new version number $vn_i = vo_i + 1$.

6.3 Uncontended commit operations

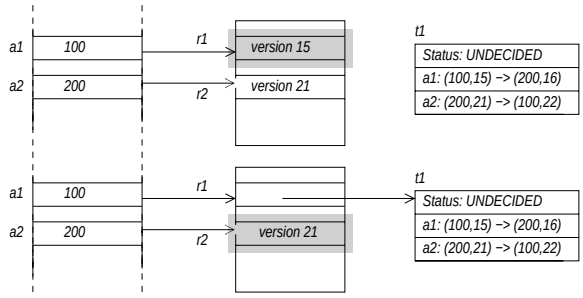
An uncontended WSTM commit operation follows the design method in Section 4. In outline, orecs ordinarily hold version numbers, as `r1` and `r2` do in Figure 13. An orec only refers to a descriptor when that transaction is attempting to commit – until `WSTMCommitTransaction` is invoked the transaction execution is private, building up a series of entries in the descriptor which set out the locations that it has accessed as in Figure 13(a).

During the first commit phase, the orecs for which $vn_i != vo_i$ are sorted and CAS is used to attempt to replace the expected version number in the orec with a reference to the transaction descriptor. This preserves the logical state of the addresses in the descriptor (changing them from LS1 to LS2) and the logical state of other locations which are associated with the same orecs (changing them from LS1 to LS3). Figure 13(b) shows these steps in our example of an uncontended commit operation. We defer, for the moment, what happens in a contended operation when one of these CAS invocations fails because it encounters a reference to another transaction descriptor.

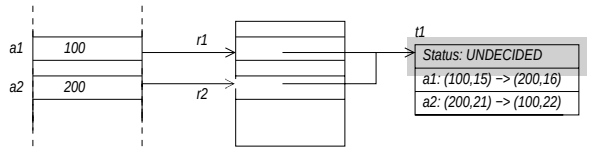
If all of the orecs are acquired successfully then CAS is used to mark the descriptor



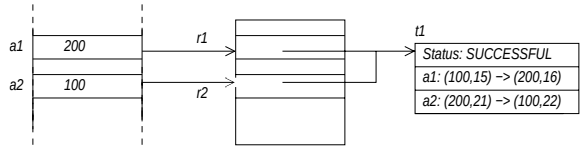
(a) The transaction executes in private until it attempts to commit.



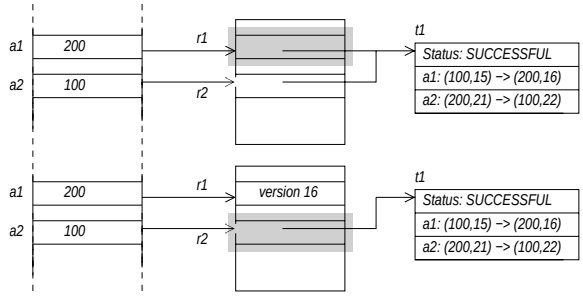
(b) CAS is used to acquire ownership records `r1` and `r2`, replacing the expected version number with a pointer to the transaction descriptor.



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



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



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

Fig. 13. 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 using the transaction descriptor.

as READ-CHECK; otherwise it is marked as FAILED. Read-checking proceeds as with validation: the logical contents of each orec associated with a read-only entry are determined and, if any does not match, the descriptor is marked as FAILED.

If the acquisition phase and the read-check phase have succeeded then CAS is used to mark the descriptor as SUCCESSFUL. Note that this decision point when the status field is updated has the effect of atomically updating the logical state of the locations accessed in the transaction (under LS2) and updating the version number, but not the logical contents, of other locations outside the transaction (under LS3). Figure 13(c) shows this step.

The next stage, for successful transactions, is to write any updates made by the transaction back to the heap. Since the transaction descriptor has acquired the orecs involved the logical contents of the locations are taken from the descriptor and so these crucial writes do not, in fact, update the logical state of the heap. Figure 13(d) shows this step.

The final phase of the WSTM commit operation is to release ownership of those orecs which were acquired successfully. This means proceeding through these orecs replacing the references to the transaction descriptor with the new version number (for successful transactions) and the old version number (for failed ones). Figure 13(e) shows this final step.

6.4 Contended commit operations

Designing a contention resolution strategy which is non-blocking is more difficult with WSTM then with MCAS. This is because, with MCAS, it was possible to deal with contention by having the second thread that attempted to acquire a resource resolve the contention by helping the first thread complete its work; this was the purpose of the MCASHelp function in Figure 8.

The problem with WSTM lies in helping commit operations which are in the process of writing updates to the heap – that is, leading up to Figure 13(d). These updates are made directly by ordinary write operations and so, even if one thread performs these updates on behalf of another, it is unsafe to release ownership of the orecs on its behalf because the first thread may subsequently perform the updates when it is next scheduled.

The approach we take, if non-blocking commit operations are required, is to ensure that an orec does not return to holding a version number until it is certain that no threads are in the process of writing updates to heap locations that it controls. If one transaction, say t2, encounters an orec that has been acquired by another transaction, say t1 then t2 performs three steps:

—Firstly, it ensures that t1 has reached its decision point. This is straightforward: if t1 is still UNDECIDED or READ-CHECK then it it is aborted.

—Secondly, to ensure that the logical contents of locations accessed by t1 are not accidentally updated by t2, t2 merges the contents of t1's descriptor into its own, ensuring that if t1 has SUCCEEDED then these values will be restored to the heap even if t2 subsequently aborts.

—Finally, we introduce a counter into each orec saying how many transactions are in the process of making updates to the locations it manages. When stealing, the counter is incremented atomically with updating the owner using a double-word-

width CAS instruction (available on IA-32 and SPARC systems). 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 if that of the most recent transaction.

This design gives obstruction-free behaviour but, in theory, has many undesirable properties: (*i*) if a thread is not scheduled for a long time then any orecs it was holding cannot be released, (*ii*) the merging of descriptors means that if it is common for orecs to be stolen then the size of an individual transaction descriptor is bounded only by the heap size and (*iii*) the need to use a double-word-width CAS limits the practicality of the algorithm because not every CPUs provide it.

We have a further scheme which obtains many of the benefits of non-blocking behaviour but relies on operating system support. In this scheme, if `t2` encounters an orec held by `t1` then `t2` proceeds to release any orecs it already holds, it then acquires a process-wide 'suspension lock', suspends `t1`, updates the program counter of the suspended thread to be just after the commit operation, ensures `t1`'s descriptor has reached its decision point, updates the heap if necessary, releases the suspension lock and then resumes `t1`. This relies on operating system support to prevent pre-emption while holding the lock and to implement the control interfaces for displacing the execution of `t1` – on Solaris UNIX the former can be hinted using `schedctl` and the latter is possible through control files under `/proc`.

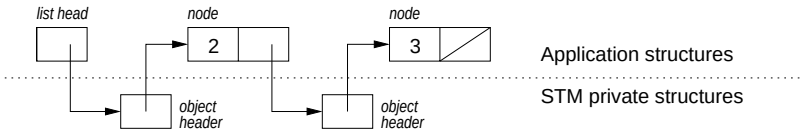
This suspension scheme is effectively a rather direct and heavyweight way for `t2` to help `t1` but, when combined with the simple approach of `t2` briefly spinning for `t1` to complete its operation this code path is rarely executed. Similar techniques have been used in implementing fast locks [Burrows 2003] and memory allocators [Dice and Garthwaite 2002]. As our results show in Section 8, the fact that the suspension path is cumbersome is of limited concern because it is rarely executed when contention management is effective. Indeed, by removing the need for counters in orecs, it simplifies the implementation of uncontended execution paths.

7. OBJECT-BASED SOFTWARE TRANSACTIONAL MEMORY

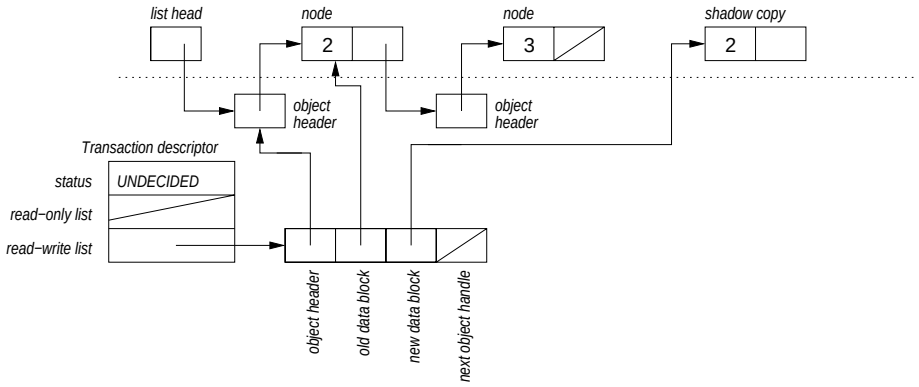
Following several previous transactional memory designs [Moir 1997; Herlihy et al. 2003], OSTM groups memory locations into contiguous blocks, or *objects*, which act as the unit of concurrency and update. Rather than containing pointers, data structures contain opaque OSTM *handles* which may be converted to directly-usable machine pointers by *opening* them as part of a transaction. Each object that is opened during a transaction is remembered as a consistency assumption to be checked before closing the object during the commit phase.

7.1 Memory formats

We begin this section by describing the memory layout when no transactions are in progress. We then describe how the OSTM implementation tracks the objects that a transaction opens for reading and writing and how, as with WSTM transaction descriptors are used during commit operations.



(a) Example OSTM-based linked list structure used by pseudocode in Figure 4. List nodes are chained together via object headers, which are private to the STM. References to object headers are known as object references and must be converted to list-node references using OSTMOpenForReading or OSTMOpenForWriting within the scope of a transaction.



(b) Example of a transaction attempting to delete node 3 from the list introduced in Figure 14. The transaction has accessed one object (node 2) which it has opened for writing. The read-only list is therefore empty, while the read-write list contains one object handle describing the modified node 2.

Fig. 14. Memory formats used in the OSTM implementation.

The current contents of an OSTM object are stored within a *data block*. As with transaction descriptors, we assume for the moment that data blocks are not re-used and so a pointer uniquely identifies a particular use of a particular block of memory. Outside of a transaction context, shared references to an OSTM object point to a word-sized *object header*. The OSTM handles introduced in Section 2.3 are implemented as pointers to object headers. Figure 14(a) shows an example OSTM-based structure which might be used by the linked-list pseudocode described in the introduction.

The state of incomplete transactions is encapsulated within aa per-transaction *descriptor* structure which indicates the current status of the transaction and store lists of objects which have been opened in read-only mode and in read-write mode. Each of these list entries holds the object reference and data-block pointer and, for objects opened in read-write mode, a pointer to the thread-local *shadow copy* of the data block. Figure 14(b) illustrates the use of transaction descriptors and object handles by showing a transaction in the process of deleting a node from an ordered linked list.

Ordinarily, object headers refer to the current version of the object's data via a pointer to the current data block. However, if a transaction is in the process of

committing an update to the object, then they can refer to the descriptor for the *owning transaction*.

7.2 Logical contents

As with MCAS and WSTM, we proceed by defining thelogical contents of an object. However, the definition is more straightforward than with WSTM because (*i*) we avoid the problem of locations which are not part of a transaction from becoming owned which occurred because of the many-to-one relationship between heap words and orecs, (*ii*) we do not need to consider version numbers as part of the logical state: the non-re-use of data blocks serves this purpose.

There are two cases:

LS1 : If the object header refers to a data block then that block forms the object's logical contents.

LS2 : If the object header refers to a transaction descriptor then we take the descriptor's new value for the block (if it is SUCCESSFUL) and its old value for the block if it is UNDECIDED or FAILED.

As usual we require threads encountering a READ-CHECK descriptor to help advance it to its decision point at which point the objects involved have well-defined logical contents.

7.3 Commit operations

A transaction's commit operation follows the three-phase structure introduced in Section 4.3 and subsequently used with MCAS and WSTM.

Acquire phase. The header of each object opened in read-write mode is acquired using in some global total order (e.g. arithmetic ordering of object references) by using CAS to replace the data-block pointer with a pointer to the transaction descriptor.

Read-check phase. The header of each object opened in read-only mode is checked against the value recorded in the descriptor.

Decision point. Success or failure is then indicated by updating the status field of the transaction descriptor to indicate the final outcome (atomic update from *UNDECIDED* to *SUCCESSFUL* or *FAILED*).

Release phase. Finally, on success, each updated object has its data-block pointer updated to reference the shadow copy. On failure each updated object has its data-block restored to the old value in the transaction descriptor.

Note that this algorithm slightly complicates finding the current data block of an object: when an object is opened we may have to search within a transaction descriptor to find the data-block pointer. For clarity, in pseudocode we use ISOSTMDesc(p) to determine whether the given pointer p is a reference to a transaction descriptor. As with MCAS, reading from an acquired object header does not need to involve recursive helping unless the owner is in a read phase: the current logical version of the object can be determined from the contents of the transaction descriptor.

```
1  typedef struct { t *data; } ostm_handle<t*>;
   typedef struct { stm_obj<t*> *obj; t *old, *new; } obj_entry<t*>;
3  typedef struct { word status;
                    obj_entry_list read_list, write_list; } ostm_transaction;

5  static t *obj_read (ostm_transaction *tx, ostm_handle<t*> *o) {
     t *data := o→data;
7    if ( ISOSTMDesc(data) ) {
       ostm_transaction *other := (ostm_transaction *)data;
9      obj_entry<t*> *hnd := search(o, other→write_list);
       if ( other→status = READ_CHECK )
11       if ( (tx→status ≠ READ_CHECK) ∨ (t > other) )
           commit_transaction(other); // Help other
13       else
           CAS(&other→status, READ_CHECK, FAILED); // Abort other
15     data := (other→status = SUCCESSFUL) ? hnd→new : hnd→old;
     }
17   return data;
   }
19 t *OSTMOpenForWriting (ostm_transaction *tx, ostm_handle<t*> *o) {
     obj_entry<t*> *hnd := search(o, tx→write_list);
21   if ( hnd ≠ NULL ) return hnd→new;
     if ( (hnd := search(o, tx→read_list)) ≠ NULL ) {
23     remove(o, tx→read_list); // Upgrading to write
     } else {
25     hnd := new obj_entry<t*>();
       (hnd→obj, hnd→old) := (o, obj_read(t, o));
27   }
     hnd→new := clone(hnd→old);
29   insert(hnd, tx→write_list);
     return hnd→new;
31 }
   bool commit_transaction (ostm_transaction *tx) {
33   word data, status, desired_status := FAILED;
     obj_entry *hnd, *ohnd;
35   for ( hnd in tx→write_list ) /* Acquire phase */
       while ( ¬CAS(&hnd→obj→data, hnd→old, t) ) {
37       if ( (data := hnd→obj→data) = t ) break;
         if ( ¬ISOSTMDesc(data) ) goto decision_point;
39       commit_transaction((ostm_transaction *)data);
       }
41   CAS(&tx→status, UNDECIDED, READ_CHECK);
     for ( hnd in tx→read_list ) /* Read phase */
43     if ( (data := obj_read(t, hnd→obj)) ≠ hnd→old ) goto decision_point;
     desired_status := SUCCESSFUL;
45 decision_point:
     while ( ((status := tx→status) ≠ FAILED) ∧ (status ≠ SUCCESSFUL) )
47     CAS(&tx→status, status, desired_status);
     for ( hnd in tx→write_list ) /* Release phase */
49     CAS(&hnd→obj→data, t, status = SUCCESSFUL ? hnd→new : hnd→old);
     return (status = SUCCESSFUL);
51 }
```

Fig. 15. OSTM's OSTMOpenForWriting and OSTMCommitTransaction interface calls. Algorithms for *read* and *read-write* lists are not given here. Instead, search, insert, remove and for-in iterator operations are assumed to exist, e.g. acting on linked lists of obj_entry structures.

Figure 15 presents pseudocode for the OSTMOpenForWriting and OSTMCommitTransaction operations. Both operations use obj_read to find the most recent data block for a given object reference; we therefore describe this sub-operation first. In most circumstances the latest data-block reference can be returned directly from the object header (lines 6 and 17). If the object is currently owned by a committing transaction then the correct reference is found by searching the owner's read-write list (line 9) and selecting the old or new reference based on the owner's current status (line 15). If the owner is in its read phase then it must be helped to completion or aborted, depending on the status of the transaction that invoked its obj_read and its ordering relative to the owner (lines 10–14).

OSTMOpenForWriting proceeds by checking whether the object is already open; if so, the existing shadow copy is returned (lines 20–21). If the object is present on the read-only list then the matching handle is removed (line 23). If the object is present on neither list then a new object handle allocated and initialised (lines 24–25). A shadow copy of the data block is made (line 28) and the object handle is inserted into the read-write list (line 29).

OSTMCommitTransaction itself is divided into three phases. The first phase attempts to acquire each object in the read-write list (lines 35–40). If a more recent data-block reference is found then the transaction is failed (line 38). If the object is owned by another transaction then the obstruction is helped to completion (line 39). The second phase checks that each object in the read-only list has not been updated since it was opened (lines 42–43). If all objects were successfully acquired or checked then the transaction will attempt to commit successfully (lines 46–47). Finally, each acquired object is released (lines 48–49); the data-block reference is returned to its previous value if the transaction failed, otherwise it is updated to its new value.

8. EVALUATION

There is a considerable gap between the pseudocode designs presented for MCAS, WSTM and OSTM and a useful implementation of those algorithms on which to base our evaluation. In this section we highlight a number of these areas elided in the psedocode and then assess the practical performance of our implementations by using them to build concurrent skip-lists and red-black trees.

8.1 Implementation concerns

We consider four particular implementation problems: supporting nested transactions for composition (Section 8.1.1), performing validation to detect uncommitable transaction (Section 8.1.2), distinguishing descriptors from application data (Section 8.1.3) and managing the memory within which descriptors are contained (Section 8.1.4).

8.1.1 *Nested transactions.* In order to allow composition of STM-based operations we introduce limited support for nested transactions. This takes the simple form of counting the number of StartTransaction invocations that are outstanding in the current thread and only performing an actual CommitTransaction when the count is returned to zero. This means that it is impossible to abort an inner transaction without aborting its enclosing transactions.

An alternative implementation would be to use separate descriptors for enclosed transactions and, upon commit, to merge these into the descriptors for the next transaction out. This would allow an enclosed transaction to be aborted and retried without requiring that all of the enclosing transactions be aborted.

8.1.2 *Transaction validation.* A non-obvious but in practice rather serious complication arises when dealing with transactions which become inconsistent at some point during their execution – for instance a transaction that has read from a location to which an update has subsequently been committed. In practice there are two ways in which inconsistent data can prevent progress: the application may crash, or it may loop indefinitely. The same problem, of course, occurs when programming with MCAS because that API gives no guarantee that a series of MCASRead operations provide an atomic snapshot of the heap.

An application which suffers from these problems can be modified to validate the current transaction in appropriate places. This requires validation checks to be inserted immediately before critical operations which may cause a crash, and inside loops for which termination depends on transactional data. A failed validation causes the application to abort the current transaction and reattempt it, thus averting program failure or unbounded looping.

Our experience when implementing red-black trees over WSTM and OSTM was that determining where to place explicit validation checks is tedious and error-prone. We further observed that validation checks were only required in two types of situation: (*i*) to avoid a memory-protection fault, usually due to dereferencing a NULL pointer; and (*ii*) to prevent indefinite execution of a loop. Furthermore, we observed that each loop in the data structures implementation contained at least one STM operation.

Consequently, we adopted two techniques to automate the placement of lightweight validation in these settings. Firstly, when a transaction is started WSTM and OSTM save enough state to automatically return control to that point if the transaction becomes invalid: in a C/UNIX environment this can be done portably using the POSIX `setjmp` and `longjmp` routines. We install a signal handler which catches memory-protection faults and validates the in-progress transaction, if any. If the validation fails then the transaction is restarted.

Secondly, each STM operation probabilistically checks the consistency of one entry in the descriptor of the in-progress transaction. This avoids unbounded looping because the inconsistency will eventually be detected and the transaction automatically restarted. The probability of validation can be reduced to gain faster execution of STM operations at the expense of slower detection of inconsistencies.

An alternative implementation would be to perform full validation on *every* WSTMRead and OSTMOpen operation, thereby ensuring that the values seen within a transaction represent a mutually-consistent snapshot of part of the heap. This is effectively the approach taken by Herlihy *et al*'s obstruction-free STM and leads to the need to either make reads visible to other threads (making read parallelism difficult in a streamlined implementation) or explicit re-validation (leading to $O(n^2)$ behaviour when a transaction opens n objects in turn).

8.1.3 *Descriptor identification.* To allow implementation of the *IsMCASDesc*, *IsCCASDesc*, *IsWSTMDesc* and *IsOSTMDesc* predicates from Sections 5–7, there needs to be a way to distinguish pointers to descriptors from other valid memory values.

We do this by reserving the two low-order bits in each pointer that may refer to a descriptor. This limits CCAS and MCAS to only operate on pointer-typed locations, as dynamically distinguishing a descriptor reference from an integer with the same representation is not generally possible. However, OSTM descriptors are only ever installed in place of data-block pointers, so OSTM trivially complies with this restriction. Similarly, WSTM descriptor-pointers are only installed in orecs in place of version numbers: we use even values to indicate descriptor pointers and odd values to indicate version numbers.

Of course, other implementation schemes are possible, for instance using run-time type information or placing descriptors in distinct memory pools.

8.1.4 *Reclamation of dynamically-allocated memory.* Note that we are actually faced with two separate memory management problems: how to manage the memory within which descriptors are held and how to manage the memory within which application data structures are held. The latter problem has been subject to extensive recent work, such as SMR [Michael 2002b] and pass-the-buck [Herlihy et al. 2002], and either of those schemes (or others) can be used by the application.

That leaves the former problem of managing descriptors: so far we have assumed that they are reclaimed by garbage collection and we have benefited from this assumption by being able to avoid A-B-A problems that would otherwise be caused by re-use. Although general solutions such as SMR and pass-the-buck remain applicable, we can benefit from using a separate scheme because of the different workload. This is because our goal of disjoint-access parallelism means that when threads are performing non-overlapping updates then their descriptors do not become shared and so can be re-used directly.

We achieve this direct re-use with simple reference counting, placing a count in each MCAS, WSTM and OSTM descriptor and updating this to count the number of threads which may have active references to the descriptor. Michael and Scott's method is used to determine when reuse is safe [Michael and Scott 1995].

We manage CCAS descriptors by embedding a pool of them within each MCAS descriptor. In fact, embedding a small number of CCAS descriptors within each MCAS descriptor is sufficient because each one can be immediately reused as long as it is only introduced to any particular memory location at most once. This restriction is satisfied by allocating a single CCAS descriptor to each process that participates in an MCAS operation; each process then reuses its descriptor for each of the CCAS sub-operations that it executes. Unless contention is very high it is unlikely that recursive helping will occur often, and so the average number of processes participating in a single MCAS operation will be very small.

If excessive helping does ever exhaust the embedded cache of CCAS descriptors then further allocation requests must be satisfied by dynamic allocation. These dynamically-allocated descriptors are managed by the same reference-counting mechanism as MCAS and OSTM descriptors.

The same storage method is used for the per-transaction object lists maintained

by OSTM. Each transaction descriptor contains a pool of embedded entries that are sequentially allocated as required. If a transaction opens a very large number of objects then further descriptors are allocated and chained together to extend the node pool.

8.2 Performance evaluation

We evaluate the performance of the three abstractions for concurrent programming without locks by using them to build implementations of shared *set* data structures and then comparing the performance of these implementations against a range of lock-based designs. All experiments were run on a Sun Fire 15K server populated with 106 UltraSPARC III processors, each running at 1.2GHz. The server comprises 18 CPU/memory boards, each of which contains four processors and several gigabytes of memory. The boards are plugged into a backplane that permits communication via a high-speed crossbar interconnect. A further 34 processors reside on 17 smaller CPU-only boards.

Each experiment is specified by three adjustable parameters:

S	—	The search structure that is being tested
P	—	The number of parallel processes accessing the set
K	—	The average number of unique key values in the set

The benchmark program begins by creating P processes and an initial set, implemented by S, containing the keys 0, 2, 4, ..., $2K$. All processes then enter a tight loop which they execute for 5 wall-clock seconds. On each iteration they randomly select whether to execute a lookup ($p = 75\%$), update ($p = 12.5\%$), or remove ($p = 12.5\%$). This distribution is chosen because reads dominate writes in many observed real workloads; it is also very similar to the distributions used in previous evaluations of parallel algorithms [Mellor-Crummey and Scott 1991b; Shalev and Shavit 2003]. When 5 seconds have elapsed, each process records its total number of completed operations. These totals are summed and used to calculate the *result* of the experiment: the mean number of CPU-microseconds required to execute a random operation.

A timed duration of 5 is sufficient to amortise the overheads associated with warming each processor's data caches, and starting and stopping the benchmark loop. We confirmed that doubling the execution time to 10 seconds does not measurably affect the final result. We plot results showing the median of 5 benchmark runs showing error bars indicating the minimum and maximum results achieved.

In addition to gathering performance figures, our test harness can log the inputs, results and invocation and response timestamps for each operation. We used an off-line checked to ensure that these observations are linearizable. Although this problem is generally NP-complete [Wing and Gong 1993], a greedy algorithm which executes a depth-first search to determine a satisfactory ordering for the invocations works well in practice [Fraser 2003]. This was invaluable for finding implementation errors such as missing memory-ordering barriers, even when we were sure of the algorithmic correctness of the designs.

We compare 14 different set implementations: 6 based on red-black trees and 8 based on skip lists. Many of these are lock-based and were created for the purpose of running these tests to provide as strong contenders as possible; we have made their

source code publicly available for inspection and Fraser describes the contenders in more detail as part of his PhD dissertation [Fraser 2003]. Fraser also considers general binary search trees and develops a range of non-blocking and lock-based designs.

Where needed by lock-based algorithms we use Mellor-Crummey and Scott's (MCS) scalable queue-based spinlocks which avoid unnecessary cache-line transfers between processors that are spinning on the same lock [Mellor-Crummey and Scott 1991a]. Although seemingly complex, the MCS operations are highly competitive even when the lock is not contended; an uncontended lock is acquired or released with a single read-modify-write access. Furthermore, contended MCS locks create far less memory traffic than standard *test-and-set* or *test-and-test-and-set* locks.

Where multi-reader locks are required we use another queue-based design by the same authors which allows adjacently-queued readers may enter their critical regions simultaneously when the first of the sequence reaches the head of the queue [Mellor-Crummey and Scott 1991b].

In summary the 14 set implementations considered here are:

(1) Skip lists with per-pointer locks. Pugh describes a highly-concurrent skip list implementation which uses per-pointer mutual-exclusion locks [Pugh 1990]. Any update to a pointer must be protected by its lock. Deleted nodes have their pointers updated to link *backwards* thus ensuring that a search correctly backtracks if it traverses into a defunct node.

(2) Skip lists with per-node locks. Although per-pointer locking successfully limits the possibility of conflicting processes, the overhead of acquiring and releasing so many locks is an important consideration. We therefore include Pugh's design using per-node locks. The operations are identical to those for per-pointer locks, except that a node's lock is acquired before it is first updated and continuously held until after the final update to the node. Although this slightly increases the possibility of conflict between processes, in many cases this is more than repaid by the reduced locking overheads.

(3) Skip lists built directly from CAS. The direct-CAS design performs composite update operations using a sequence of individual CAS instructions, with no need for a dynamically-allocated per-operation 'descriptor'. This means that great care is needed to ensure that updates occur atomically and consistently. In outline, list membership is defined according to presence in the lowest level of the live. Insertion or deletion is performed on each level in turn as an independent linked list, using Harris's marking technique [Harris 2001] to logically delete a node from each level of the skip list in turn [Fraser 2003]. This implementation is used to show the performance gains that are possible using an intricate non-blocking system when compared with a one built from MCAS, WSTM or OSTM. The CAS-based design is notable, when compared with these abstractions, in that it does not need any temporarily allocated descriptors.

(4) Skip lists built using MCAS. Insertions and deletions proceed by building up batches of memory updates to make through a single MCAS invocation. As with Pugh's schemes, pointers within deleted nodes are reversed to aid concurrent searches.

(5-6) Skip lists built using WSTM. Skip lists can be built straightforwardly from single-threaded code using WSTM. We consider two variants: a non-blocking WSTM

built using double-word-width compare and swap and a version using the suspension scheme described in Section 6.4.

(7-8) Skip lists built using OSTM. Skip lists can be built straightforwardly from OSTM by representing each list node as a separate OSTM object. We consider two variants: the lock-free OSTM scheme described in Section 7 and Herlihy *et al*'s obstruction-free STM [Herlihy et al. 2003].

(9) Red-black trees with serialised writers. Unlike skip lists there has been little practical work on parallelism in balanced trees. Our first design [Fraser 2003] builds on Hanke's [Hanke 1999] and uses lock-coupling when searching down the tree, upgrading to a write mode when performing rebalancing (taking care to avoid deadlock by upgrading in down-the-tree order). A global mutaul-exclusion lock is used to serialise concurrent writers.

(10) Red-black trees with concurrent writers. Our second scheme allows concurrent writing by decomposing tree operations into a series of local updates on tree fragments [Fraser 2003]. It is similar to Hanke's relaxed red-black tree in that it decouples the steps of rebalancing the tree from the actual insertion or deletion of a node [Hanke et al. 1997]. Although lock-based, the style of the design is reminiscent of optimistic concurrency control because each local update is preceded by checking part of the tree in private to identify the sets of locks needed, retrying this stage if inconsistency is observed.

(11-12) Red-black trees built using WSTM. As with skip lists, red-black trees can be built straightforwardly from single-threaded code using WSTM. However, there is one caveat. In order to reduce the number of cases to consider during rotations, and in common with standard designs, we use a black sentinel node in place of NULL child pointers in the leaves of the tree. We use *write discard* to avoid updates to this introducing contention when making needless updates to the sentinel's parent pointer[1].

(13-14) Red-black trees built using OSTM. As with skip lists, each node is represented by a separate OSTM object, so nodes must be opened for the appropriate type of access as the tree is traversed. Again, write discard is used on the sentinel node.

We now consider our performance results under a series of scenarios. Section 8.2.1 looks at scalability under low contention. This shows the performance of our non-blocking systems when they are running on machines with few processors, or when they are being used carefully to reduce the likelihood that concurrent operations conflict. Our second set of results, in Section 8.2.2, considers performance under increasing levels of contention.

8.2.1 *Scalability under low contention.* The first set of results measure performance when contention between concurrent operations is very low. Each experiment runs with a mean of 2^{19} keys in the set, which is sufficient to ensure that

[1]Herlihy *et al*'s OSTM cannot readily support write discard because only one thread may have an OSTM object open for writing at a time. Their *early release* scheme applies only to read-only accesses. To avoid contention on the sentinel node we augmented their STM with a mechanism for registering objects with *non-transactional semantics*: such objects can be opened for writing but the shadow copy remains thread private and is discarded on commit or abort.

parallel writers are extremely unlikely to update overlapping sections of the data structure. A well-designed algorithm which provides disjoint-access parallelism will avoid introducing contention between these logically non-conflicting operations.

Note that all the graphs in this section show a significant drop in performance when parallelism increases beyond 5 to 10 processors. This is due to the architecture of the underlying hardware: small benchmark runs execute within one or two processor 'quads', each of which has its own on-board memory. Most or all memory reads in small runs are therefore serviced from local memory which is considerably faster than transferring cache lines across the switched inter-quad backplane.

Figure 16 shows the performance of each of the skip-list implementations. As expected, the STM-based implementations perform poorly compared with the other lock-free schemes; this demonstrates that there are significant overheads associated with the read and write operations (in WSTM) or with maintaining the lists of opened objects, constructing shadow copies of updated objects (in OSTM). Additionally, access-validation is necesssary in these cases, unlike lock-based schemes.

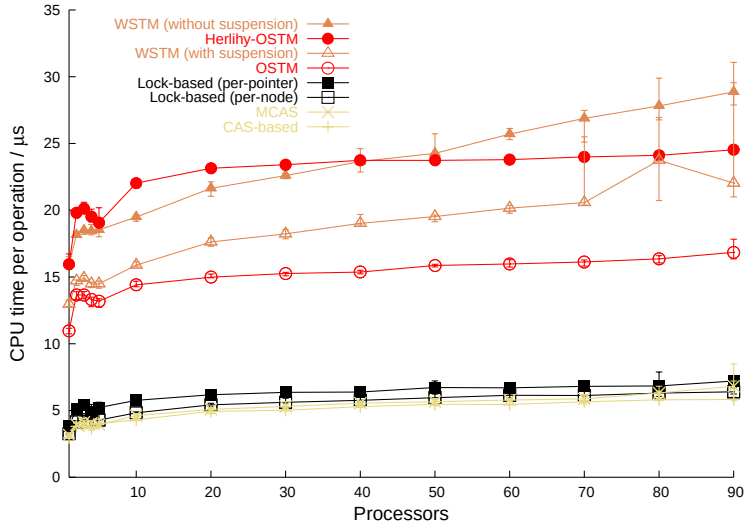
The lock-free CAS-based and MCAS-based designs perform extremely well because, unlike the STMs, they add only minor overheads on each memory access. Interestingly, under low contention the MCAS-based design has almost identical performance to the much more complicated CAS-based design — the extra complexity of using hardware primitives directly is not always worthwhile. Both schemes surpass the two lock-based designs, of which the finer-grained scheme is slower because of the costs associated with traversing and manipulating the larger number of locks.

Figure 17, presenting results for red-black trees, gives the clearest indication of the benefits of lock-free programming. Neither of the lock-based schemes scales effectively with increasing parallelism; indeed, both OSTM and WSTM-based trees out-perform the schemes using locking with only 2 concurrent processors. Of course, the difficulty of designing effective lock-based trees motivated the development of skip lists, so it is interesting to observe that a straightforward tree implementation, layered over STM does scale well and often performs better than our skip list implementations.

Surprisingly, the scheme that permits parallel updates performs hardly any better than the much simpler and more conservative design. This is because the main performance bottleneck in both schemes is contention when accessing the multi-reader lock at the root of the tree. Although multiple readers can enter their critical region simultaneously, there is significant contention for updating the shared synchronisation fields within the lock itself. Put simply, using a more permissive type of lock (i.e., multi-reader) does not improve performance because the bottleneck is caused by cache-line contention rather than lock contention.

In contrast, the STM schemes scale very well because transactional reads do not cause potentially-conflicting memory writes in the underlying synchronisation primitives. OSTM is considerably faster then Herlihy's design, due to better cache locality. Herlihy's STM requires a triple-indirection when opening a transactional object: thus three cache lines are accessed when reading a field within a previously-unopened object. In contrast my scheme accesses two cache lines; more levels of the tree fit inside each processor's caches and, when traversing levels that do not

(a)



(b)

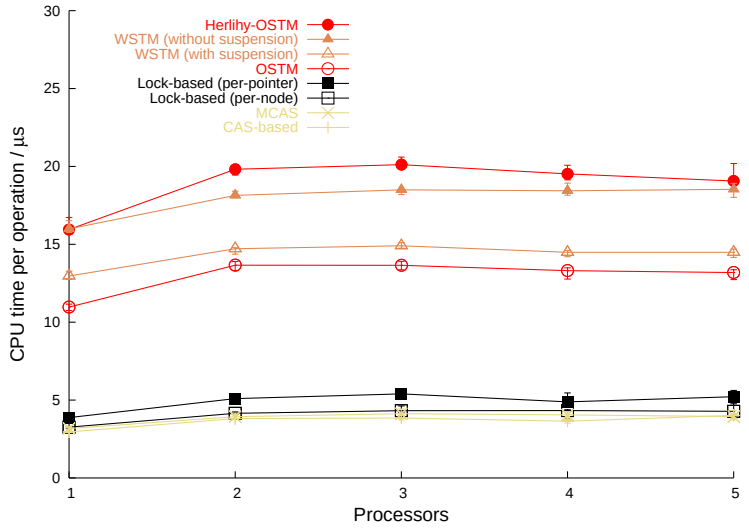


Fig. 16. Graph (a) shows the performance of large skip lists ($K = 2^{19}$) as parallelism is increased to 90 processors. Graph (b) is a 'zoom' of (a), showing the performance of up to 5 processors.

fit in the cache, 50% fewer lines must be fetched from main memory.

8.2.2 *Performance under varying contention.* The second set of results shows how performance is affected by increasing contention — a particular concern for non-blocking algorithms, which usually assume that conflicts are rare. This assumption allows the use of *optimistic* techniques for concurrency control; when conflicts do occur they are handled using a fairly heavyweight mechanism such as recursive helping or interaction with the thread scheduler. Contrast this with using locks, where an operation assumes the worst and 'announces' its intent before accessing

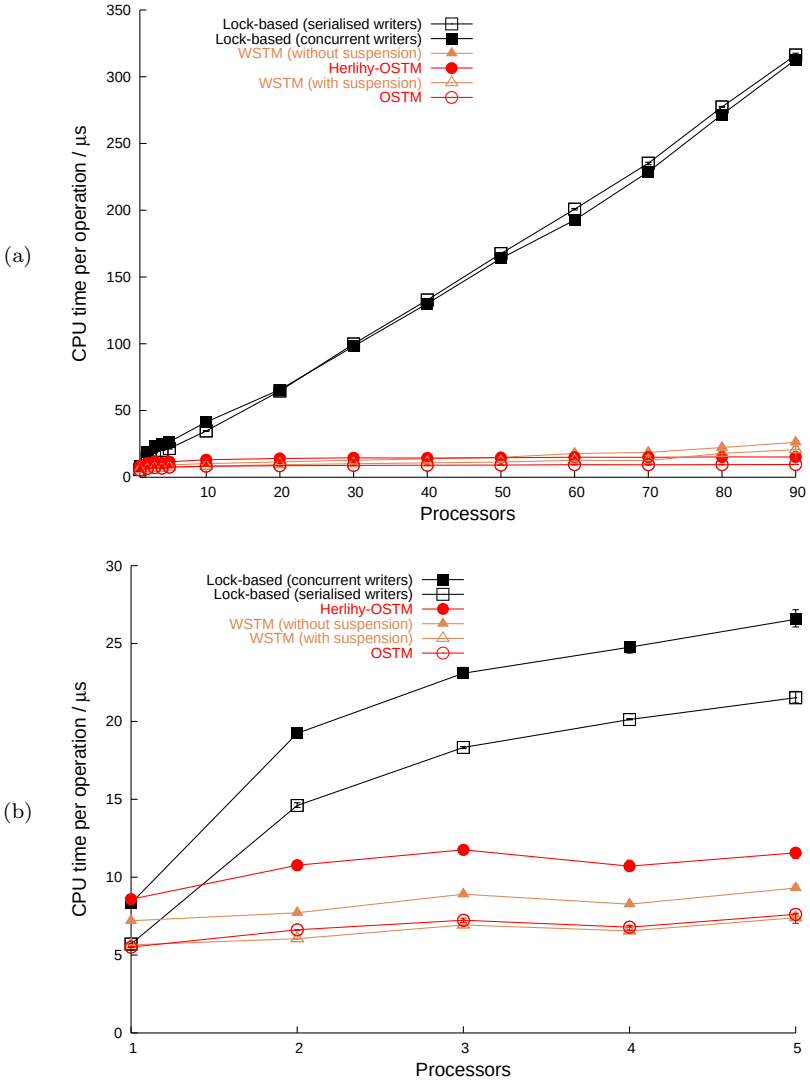


Fig. 17. Graph (a) shows the performance of large red-black trees ($K = 2^{19}$) as parallelism is increased to 90 processors. Graph (b) is a 'zoom' of (a), showing the performance of up to 5 processors.

shared data: that approach introduces unnecessary overheads when contention is low because fine-grained locking requires expensive juggling of acquire and release invocations. The results here allow us to investigate whether these overheads pay off as contention increases. All experiments are executed with 90 parallel processes ($P = 90$).

Figure 18 shows the effect of contention on each of the skip-list implementations. It indicates that there is sometimes a price for using high-level abstractions such

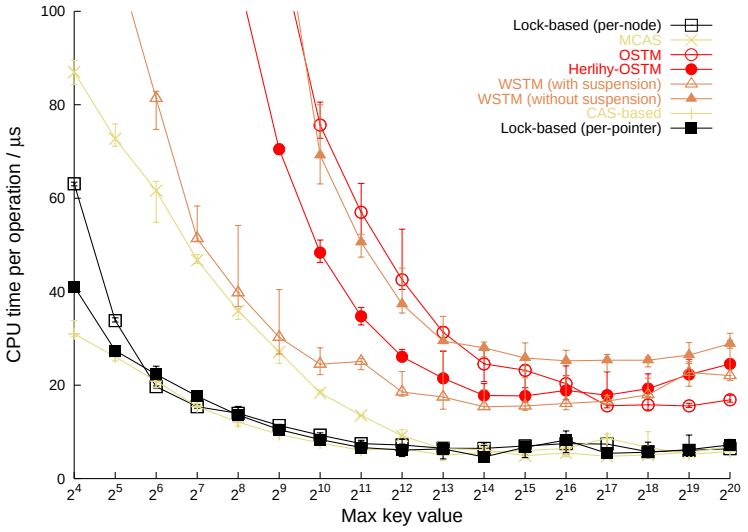


Fig. 18. Effect of contention on concurrent skip lists ($P = 90$).

as MCAS. The poor performance of MCAS when contention is high is because many operations must retry several times before they succeed: it is likely that the data structure will have been modified before an update operation attempts to make its modifications globally visible. In contrast, the carefully-implemented CAS-based scheme attempts to do the minimal work necessary to update its 'view' when it observes a change to the data structure. This effort pays off under very high contention; in these conditions the CAS-based design performs as well as per-pointer locks. These results also demonstrate a particular weakness of locks: the optimal granularity of locking depends on the level of contention. Here, per-pointer locks are the best choice under very high contention, but they introduce unnecessary overheads compared with per-node locks under moderate to low contention. Lock-free techniques avoid the need to make this particular tradeoff. Finally, note that the performance of each implementation drops slightly as the mean set size becomes very large. This is because the time taken to search the skip list begins to dominate the execution time.

Finally, Figure 19 presents results for red-black trees, and shows that locks are not always the best choice when contention is high. Both lock-based schemes suffer contention for cache lines at the root of the tree where most operations must acquire the multi-reader lock. The OSTM and WSTM scheme using suspension perform well in all cases, although conflicts still significantly affect its performance.

Herlihy's STM performs comparatively poorly under high contention when using an initial contention-handling mechanism which introduces exponential backoff to 'politely' deal with conflicts; other schemes may work better [Scherer and Scott 2004]. Furthermore, using the basic contention manager, the execution times of individual operations are very variable, which explains the performance 'spike' at the left-hand side of the graph. This low and variable performance is caused by

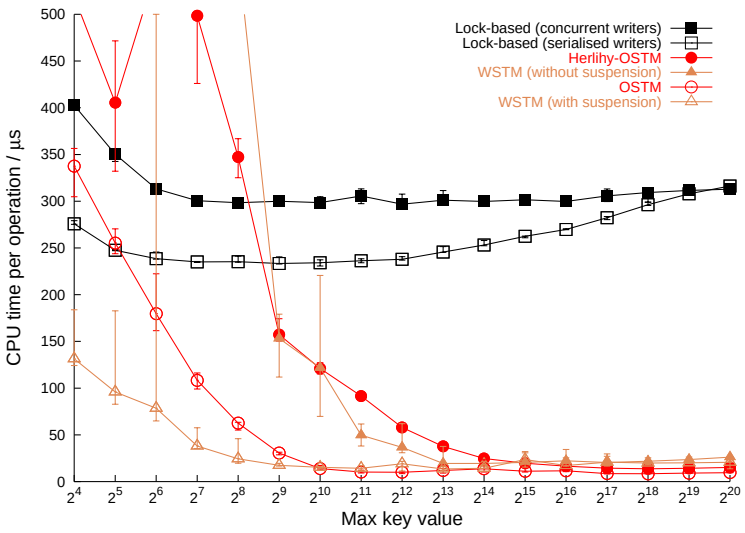


Fig. 19. Effect of contention on concurrent red-black trees ($P = 90$).

		Run-time performance	
Rank	**Ease of use**	**Low contention**	**High contention**
1	STM	CAS, MCAS	CAS, W-locks
2	RW-locks	—	—
3	MCAS	W-locks	MCAS
4	W-locks	STM	STM
5	CAS	RW-locks	RW-locks

Fig. 20. Effectiveness of various methods for managing concurrency in parallel applications, according to three criteria: ease of use for programmers, performance when operating within a lightly-contended data structure, and performance within a highly-contended data structure. The methods are ranked under each criterion, from best- to worst-performing.

sensitivity to the choice of back-off rate: our implementation uses the same values as the original authors, but these were chosen for a Java-based implementation of red-black trees and they do not discuss how to choose a more appropriate set of values for different circumstances.

9. CONCLUSION

The results presented in this paper demonstrate that well-implemented non-blocking algorithms can match or surpass the performance of state-of-the-art lock-based designs in many situations. Thus, not only do these non-blocking abstractions have many *functional advantages* compared with locks (such as freedom from deadlock and unfortunate scheduler interactions), but they can also be implemented on modern multiprocessor systems with *better performance* than traditional lock-based schemes.

Figure 20 presents a comparison of each of the synchronisation techniques that we have discussed. The comparative rankings are based on observation of how easy

it was to design practical search structures using each technique, and the relative performance results under varying levels of contention between concurrent update operations. *CAS*, *MCAS* and *STM* represent the three lock-free techniques. *RW-locks* represents data structures that require both read and write operations to take locks: these will usually be implemented using multi-reader locks. *W-locks* represents data structures that only use locks to synchronise write operations — some other method, usually an optimistic scheme, is used to ensure that readers are correctly synchronised with respect to concurrent updates.

In situations where ease of use is most important, STM and RW-locks are the best choices because they both ensure that readers are synchronised with concurrent updates: transactions or locking can be wrapped around a sequential implementation. STM is ranked above RW-locks because it avoids the need to consider issues such as granularity of locking and the order in which locks should be acquired to avoid deadlock. MCAS and W-locks have similar complexity: they both handle synchronisation between concurrent updates but an out-of-band method may be required to synchronise readers. Like STM, MCAS is ranked higher than W-locks because it avoids implementation issues that pertain only to locks. CAS is by far the trickiest abstraction to work with because some method must be devised to efficiently 'tie together' related updates to multiple memory locations; non-blocking implementations of specific data structures built using CAS remains a research topic [Harris 2001; Shalev and Shavit 2003].

When access to a data structure is not commonly contended, CAS and MCAS both perform very well. W-locks tend to perform slightly worse because of reduced cache locality compared with lock-free techniques, and the overhead of juggling locks when executing write operations. OSTM performs worse than CAS, MCAS and W-locks because of transactional overheads and the need to *double read* object headers to ensure that transactional reads are consistent during commit. WSTM performs similarly to OSTM when the number of objects opened using OSTM is comparable to the number of reads or writes performed using WSTM. RW-locks generally perform worst of all, particularly for a data structure which has only one point of entry: this *root* can easily become a performance bottleneck due to concurrent updates to fields within its multi-reader lock.

Under high contention, CAS-based designs perform well if they have been carefully designed to do the least possible work when an inconsistency or conflict is observed — however, this may require a very complicated algorithm. The extra space and time overheads of W-locks pay off under very high contention: MCAS performs considerably worse because memory locations are very likely to have been updated before MCAS is even invoked. OSTM also suffers because it, like MCAS, is an optimistic technique which detects conflicts *after* time has been spent executing a potentially expensive operation. However, it will still perform better than RW-locks in many cases because contention at the root of the data structure is still the most significant performance bottleneck for this technique.

In conclusion, using the programming abstractions and implementations that we have presented in this dissertation, it is now practical to deploy lock-free techniques, with all their attendant advantages, in many real-world situations where lock-based synchronisation would traditionally be the only viable option.

ACKNOWLEDGMENT

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

REFERENCES

Anderson, J. H. and Moir, M. 1995. Universal constructions for multi-object operations. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC '95)*. 184–193.

Anderson, J. H., Ramamurthy, S., and Jain, R. 1997. Implementing wait-free objects on priority-based systems. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing (PODC '97)*. 229–238.

Barnes, G. 1993. A method for implementing lock-free data structures. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*. 261–270.

Burrows, M. 2003. How to implement unnecessary mutexes. In *Computer Systems: Theory, Technology and Applications*. Springer-Verlag.

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

Fraser, K. 2003. Practical lock freedom. Ph.D. thesis, Computer Laboratory, University of Cambridge.

Greenwald, M. 1999. Non-blocking synchronization and system design. Ph.D. thesis, Stanford University. Also available as Technical Report STAN-CS-TR-99-1624, Stanford University, Computer Science Department.

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

Hanke, S. 1999. The performance of concurrent red-black tree algorithms. In *Proceedings of the 3rd Workshop on Algorithm Engineering*. Lecture Notes in Computer Science, vol. 1668. Springer-Verlag, 287–301.

Hanke, S., Ottmann, T., and Soisalon-Soininen, E. 1997. Relaxed balanced red-black trees. In *Proceedings of the 3rd Italian Conference on Algorithms and Complexity*. Lecture Notes in Computer Science, vol. 1203. Springer-Verlag, 193–204.

Harris, T. 2001. A pragmatic implementation of non-blocking linked lists. In *Proceedings of the 15th International Symposium on Distributed Computing (DISC '01)*. Springer-Verlag, 300–314.

Harris, T. and Fraser, K. 2003. Language support for lightweight transactions. In *Proceedings of the 18th Annual ACM-SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '03)*. 388–402.

Harris, T., Fraser, K., and Pratt, I. 2002. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Symposium on Distributed Computing (DISC '02)*. Springer-Verlag.

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

Herlihy, M., Luchangco, V., and Moir, M. 2002. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *Proceedings of the 16th International Symposium on Distributed Computing (DISC '02)*. Springer-Verlag.

Herlihy, M., Luchangco, V., and Moir, M. 2003. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE.

Herlihy, M., Luchangco, V., Moir, M., and Scherer, W. 2003. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC '03)*. 92–101.

Herlihy, M. and Moss, J. E. B. 1993. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA '93)*. ACM Press, 289–301.

Herlihy, M. and Wing, J. M. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems 12,* 3 (July), 463–492.

Israeli, A. and Rappoport, L. 1994. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the 13nd Annual ACM Symposium on Principles of Distributed Computing (PODC '94)*. 151–160.

Mellor-Crummey, J. and Scott, M. 1991a. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems 9,* 1, 21–65.

Mellor-Crummey, J. and Scott, M. 1991b. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 106–113.

Michael, M. M. 2002a. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*. ACM Press, 73–82.

Michael, M. M. 2002b. 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 (PODC '02)*.

Michael, M. M. and Scott, M. 1995. Correction of a memory management method for lock-free data structures. Tech. Rep. TR599, University of Rochester, Computer Science Department. Dec.

Moir, M. 1997. Transparent support for wait-free transactions. In *Distributed Algorithms, 11th International Workshop*. Lecture Notes in Computer Science, vol. 1320. Springer-Verlag, 305–319.

Pugh, W. 1990. Concurrent maintenance of skip lists. Technical Report CS-TR-2222, Department of Computer Science, University of Maryland. June.

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

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

Scherer, III, W. N. and Scott, M. L. 2004. Contention management in dynamic software transactional memory. In *Proceedings of the 2004 PODC Workshop on Concurrency and Synchronization in Java Programs*.

Shalev, O. and Shavit, N. 2003. Split-ordered lists: Lock-free extensible hash tables. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC '03)*. 102–111.

Shavit, N. and Touitou, D. 1995. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC '95)*. 204–213.

Turek, J., Shasha, D., and Prakash, S. 1992. Locking without blocking: Making lock-based concurrent data structure algorithms nonblocking. In *Proceedings of the 11th ACM Symposium on Principles of Database Systems*. 212–222.

Wing, J. M. and Gong, C. 1993. Testing and verifying concurrent objects. *Journal of Parallel and Distributed Computing 17,* 1 (Jan.), 164–182.

Received Month Year; revised Month Year; accepted Month Year