Concurrent Programming Without Locks

KEIR FRASER
University of Cambridge Computer Laboratory
and
TIM HARRIS
Microsoft Research Cambridge

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 locks can harbor problems such as priority inversion, deadlock, and convoying. Furthermore, scalable lock-based systems are not readily composable when building compound operations. In looking for solutions to these problems, interest has developed in *nonblocking* systems which have promised scalability and robustness by eschewing mutual exclusion while still ensuring safety. However, existing techniques for building nonblocking systems are rarely suitable for practical use, imposing substantial storage overheads, serializing nonconflicting operations, or requiring instructions not readily available on today's CPUs.

In this article we present three APIs which make it easier to develop nonblocking implementations of arbitrary data structures. The first API is a *multiword compare-and-swap* operation (MCAS) which atomically updates a set of memory locations. This can be used to advance a data structure from one consistent state to another. The second API 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 API is an *object-based software transactional memory* (OSTM). OSTM allows a simpler implementation than WSTM, but at the cost of reengineering the code to use OSTM objects.

We present practical implementations of all three of these APIs, built from operations available across all of today's major CPU families. We illustrate the use of these APIs by using them to build highly concurrent skip lists and red-black trees. 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 nonblocking data structures with performance comparable to, or better than, sophisticated lock-based designs.

Categories and Subject Descriptors: D.4.1 [**Operating Systems**]: Process Management—*Concurrency, mutual exclusion, synchronization*

This work was supported by donations from the Scalable Synchronization Research Group at Sun Labs Massachusetts. The evaluation was carried out using the Cambridge-Cranfield High Performance Computing Facility.

Authors' addresses: K. Fraser, Computer Laboratory, University of Cambridge, 15JJ Thomson Avenue, Cambridge, CB3 OFB, UK; T. Harris, Microsoft Research Cambridge, Roger Needham Building, 7JJ Thomson Ave., Cambridge CB3 0FB, UK; email: tharris@microsoft.com.

General Terms: Algorithms, Languages, Performance

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

ACM Reference Format:

Fraser, K. and Harris, T. 2007. Concurrent programming without locks. ACM Trans. Comput. Syst. 25, 2, Article 5 (May 2007), 61 pages. DOI = 10.1145/1233307.1233309 http://doi.acm.org/10.1145/1233307.1233309

1. INTRODUCTION

Mutual-exclusion locks are one of the most widely used and fundamental abstractions for synchronization. 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 benefits 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 favors the development of simple locking strategies which pessimistically serialize some nonconflicting operations.

—For liveness, programmers must be careful to avoid introducing deadlock and, consequently, they may cause software to hold locks for longer than would otherwise be necessary. Also, without scheduler support, programmers must be aware of priority inversion problems.

—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 article is concerned with the design and implementation of software which is safe for use on multithreaded multiprocessor shared-memory machines, but which does not involve the use of locking. Instead, we present three different APIs for making atomic accesses to sets of memory locations. These enable the direct development of concurrent data structures from sequential ones. We believe this makes it easier to build multithreaded systems which are correct. Furthermore, our implementations are *nonblocking* (meaning that even if any set of threads is stalled, the remaining threads can still make progress) and generally allow *disjoint-access parallelism* (meaning that updates made to nonoverlapping sets of locations will be able to execute concurrently).

To introduce these APIs, we shall sketch their use in a code fragment that inserts items into a singly-linked list which holds 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. For comparison, Figure 1 shows the corresponding insert operation when implemented for single-threaded use. In that figure, as in each of our examples, the insert operation proceeds by identifying nodes prev and curr between which the new node is to be placed.

Our three alternative APIs all follow a common optimistic style [Kung and Robinson 1981] in which the core sequential code is wrapped in a loop

```
typedef struct { int key; struct node *next; } node;
typedef struct { node *head; } list;
void list_insert_single_threaded (list *l, int k) {
    node *n := new node(k);
    node *prev := l→head;
    node *curr := prev→next;
    while ( curr→key < k ) {
        prev := curr;
        curr := curr→next;
    }
    n→next := curr;
    prev→next := n;
}
```

Fig. 1. Insertion into a sorted list.

which retries the insertion until it succeeds in committing the updates to memory.

Our first API provides *multiword compare-and-swap* (MCAS) which generalizes 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. Figure 2 shows how the insertion could be expressed using MCAS.

There are two fundamental changes from the sequential algorithm. Firstly, instead of updating shared locations individually, the code must call MCAS to perform the complete set of memory accesses that need to be made atomically. In the example there is only a single update to be made with MCAS, but a corresponding 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. Secondly, the code must call MCASRead whenever it reads from a location that might be subject to a concurrent update by MCAS in another thread.

MCAS and MCASRead present a rather low-level API: Programmers must be careful to use MCASRead where necessary and must also remember that the subsequent MCAS does not know which locations have been read. This can lead to cumbersome code in which the program keeps lists of the locations it has read from, and the values that it has seen in them, and then passes these lists to MCAS to confirm that the values represent a consistent view of shared memory.

The second abstraction provides a *word-based software transactional memory* (WSTM) which avoids some of these problems by allowing a series of reads and writes to be grouped as a software transaction which can be applied to the heap atomically [Harris and Fraser 2003]. Figure 3 shows our list example using WSTM. The changes from sequential code are that reads and writes to shared locations are performed through WSTMRead and WSTMWrite functions, and that this whole set of memory accesses is wrapped in a call to WSTMStartTransaction and a call to WSTMCommitTransaction calls.

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].

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

Fig. 2. Insertion into a sorted list managed using MCAS. In this case the arrays specifying the update need contain only a single element.

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

Fig. 3. Insertion into a sorted list managed using WSTM. The structure mirrors Figure 2 except that the WSTM implementation tracks which locations have been accessed based on the calls to WSTMRead and WSTMWrite.

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. In short examples, the code looks more verbose than WSTM, but the OSTM implementation is more straightforward and often runs more quickly.

While these techniques do not provide a silver bullet to designing scalable concurrent data structures, they represent a shift of responsibility away from the programmer: The API'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 API's caller remains responsible for ensuring scalability by making it unlikely that concurrent operations will need to modify overlapping sets of locations. However, this is a performance problem rather than a correctness or liveness one, and in our experience, even straightforward data structures, developed directly from

```
typedef struct { int key; ostm_handle<node*> *next_h; } node;
typedef struct { ostm_handle<node*> *head_h; } list;
void list_insert_ostm (list *l, int k) {
    node *n := new node(k);
    ostm_handle<node*> n_h := new ostm_handle(n);
    do {
        ostm_transaction *tx := OSTMStartTransaction();
        ostm_handle<node*> *prev_h := l→head_h;
        node *prev := OSTMOpenForReading(tx, prev_h);
        ostm_handle<node*> *curr_h := prev→next_h;
        node *curr := OSTMOpenForReading(tx, curr_h);
        while ( curr→key < k ) {
            prev_h := curr_h;        prev := curr;
            curr_h := prev → next_h; curr := OSTMOpenForReading(tx, curr_h);
        }
        n→next_h := curr_h;
        prev := OSTMOpenForWriting(tx, prev_h);
        prev→next_h := n_h;
    } while ( ¬OSTMCommitTransaction(tx) );
}
```

Fig. 4. Insertion into a sorted list managed using OSTM. The code is more verbose than Figure 3 because data is accessed by indirection through OSTM handles which must be opened before use.

sequential code, offer performance which 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. We define CAS to return the value it reads from the memory location.

```
atomically word CAS (word a, word e, word n, ) {
    word x := *a;
    if (x = e) *a := n;
    return x;
}
```

—*Linearizability.* In order for functions such as MCAS 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 called and when they return [Herlihy and Wing 1990].

—*Nonblocking progress guarantee.* In order to provide robustness against many liveness problems such as deadlock, implementations of our APIs should be nonblocking. This means that even if any set of threads is stalled, the remaining threads can still progress.

—*Disjoint-access parallelism.* Implementations of our APIs should not introduce contention in the sets of memory locations they access: Operations which access disjoint parts of a data structure should be able to execute in parallel [Israeli and Rappoport 1994].

Table I. Assessment of Our Implementations of These Three APIs Against Our Goals

	MCAS	WSTM	OSTM
Disjoint-access parallelism	when accessing disjoint sets of words	when accessing words that map to disjoint sets of *ownership records* under a hash function used in the implementation	when accessing disjoint sets of objects
Read parallelism	no	yes	yes
Space overhead (when no operations are in progress)	2 bits reserved in each word	fixed-size table (e.g., 65,536 double-word entries)	one word in each object handle
Composability	no	yes	yes

—*Read parallelism*. Implementations of our APIs should allow shared data that is read on different CPUs to remain in shared mode in those CPUs' data caches: Fetching a location from the cache of another CPU can be hundreds of times slower than fetching it from a local cache [Hennessy and Patterson 2003] and so we must preserve sharing where possible.

—*Dynamicity*. Implementations of our APIs should be able to support dynamically-sized data structures, such as lists and trees, in which constituent objects are allocated and reclaimed over time.

—*Practicable space costs*. Space costs should scale well with the number of threads and volume of data managed using the API. 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 reserving even this much if words are to hold unrestricted values, rather than being restricted to aligned pointers.

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

All of our APIs have concrete, linearizable, nonblocking implementations which can be used to build dynamically-sized data structures. Table I indicates the extent to which they meet our other goals.

We also have a number of non-goals.

Firstly, although our implementations of 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.

Secondly, we assume that a separate mechanism will be used to control contention between concurrent threads. This separation between progress in isolation and progress under contention follows Herlihy et al. [2003a, 2003b] and Scherer and Scott's [2005] recent work.

Finally, although our implementations support dynamically-sized data structures, our algorithms do not mandate the use of any particular method for determining when a particular piece of memory can be deallocated. In some settings this is achieved naturally by automatic garbage collection [Jones and Lins 1996] which can readily be extended to manage the data structures used by the implementations of our APIs [Harris et al. 2005]. Our examples in Figures 2–4 all assume a garbage collector. In Section 8.1.3 we describe the memory reclamation techniques that we used for our evaluation. Other authors have developed techniques that can be used in systems without a garbage collector: Herlihy et al.'s. "Repeat Offender" problem [2005] and Michael's "Safe Memory" reclamation [2002] both allow threads to issue tentative deallocation requests that are deferred until it is established that no other thread can access the memory involved.

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, MIPS, PowerPC, and SPARC processor families at `http://www.cl.cam.ac.uk/netos/lock-free`.

1.3 Structure of This Article

In Section 2 we present the three alternative APIs 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 Section 4 we describe our overall design method and the facets common to each of our designs. In Sections 5–7 we explore the details of these three APIs in turn and present our implementations of them, their relationship to previous work, and, where applicable, to contemporary work with similar goals of practicability.

In Section 8 we evaluate the performance of data structures built over our implementations of each of the APIs, both in comparison with one another and with sophisticated 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.

2. PROGRAMMING APIS

In this section we present the programming interfaces for using MCAS (Section 2.1), WSTM (Section 2.2), and OSTM (Section 2.3). These provide mechanisms for accessing and/or modifying 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 multithreaded use.

After presenting the APIs themselves, Section 2.4 discusses how they may be used in practice in sharedmemory multithreaded programs.

2.1 Multiword Compare-and-Swap (MCAS)

Multiword compare-and-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 shown 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 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 by calling MCASRead. This restriction is needed because, as we show in Section 5, the MCAS implementation places its own values in these locations while they are being updated. Furthermore, the MCAS implementation reserves two bits in each location that it may work on. In practice this means that these locations must hold aligned pointer values in which at least two low-order bits are ordinarily clear on a machine with 32-bit or 64-bit words. The full API is consequently

```
// 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[ ]);
// 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.

Using MCAS also allows expert programmers to reduce contention between concurrent operations by paring down the set of locations passed to each atomic update, or by decomposing a series of related operations into a series of MCAS calls. 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 potential pitfall for programmers directly using MCAS.

The API also precludes our goal of composability.

2.2 Word-Based Software Transactional Memory (WSTM)

Although MCAS eases the burden of ensuring correct synchronization of updates, many data structures also require consistency among groups of read operations and it is cumbersome for the application to track these calls to MCASRead and to use the results to build arrays of "no-op" updates to pass 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 algorithm which does not consider synchronization between concurrent searches may incorrectly fail.

Software transactional memories provide a way of dealing with these problems by grouping shared-memory accesses into transactions. These transactions 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.

Typically, 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 accessed

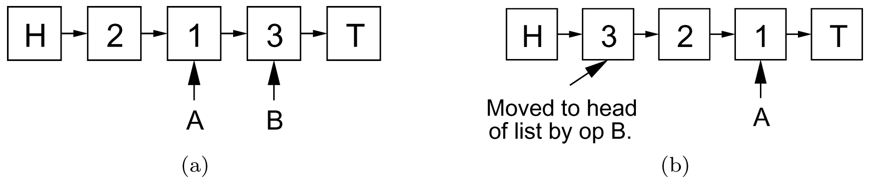


Fig. 5. The need for read consistency: A move-to-front linked list subject to two searches for node 3. In snapshot (a), search A is preempted while passing over node 1. Meanwhile, in snapshot (b), search B succeeds and moves node 3 to the head of the list. When A continues execution, it will incorrectly report that 3 is not in the list.

locations. However, as we show in Section 6.1, this is not a guarantee because false conflicts can be introduced if there are collisions under a hash function used in the WSTM implementation.

Within a transaction, data accesses are performed by WSTMRead and WSTMWrite operations. As with MCAS, the caller is responsible for 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

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

As we will show later, the 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, OSTM exposes OSTM *objects* through a level of indirection provided by OSTM *handles*. OSTM objects are allocated and deallocated by OSTMNew and OSTMFree, respectively, which behave analogously to 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 access to the underlying object: This is done by OSTMOpenForReading and OSTMOpenForWriting which, along with a transaction ID, take pointers to handles of type ostm_handle<t*> and return object pointers of type t* on which ordinary memory access operations can be invoked in that transaction. In the case of OSTMOpenForWriting, this return value refers to a *shadow* *copy* of the underlying object, that is, a private copy on which the thread can work before attempting to commit its updates.

Both OSTMOpenForReading and OSTMOpenForWriting are idempotent: If the object has already been opened in the same access mode within the same transaction, then the same pointer will be returned.

The caller must ensure that objects are opened in the correct mode: The OSTM implementation may share data between objects that have been opened for reading between multiple threads. The caller must also be careful if a transaction opens an object for reading (obtaining a reference to a shared copy C1) and subsequently opens the same object for writing (obtaining a reference to a shadow copy C2): Updates made to C2 will, of course, not be visible when reading from C1.

The OSTM interface leads to a different cost profile from WSTM: OSTM introduces a cost on opening objects for access and potentially producing shadow copies to work on, but subsequent data access is made directly (rather than through functions like WSTMRead and WSTMWrite). Furthermore, it admits a simplified nonblocking commit operation.

The OSTM API is

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

2.4 Programming with Our APIs

This section discusses some general questions that arise when developing software that uses any of our three APIs.

2.4.1 *Programming Languages.* This article concentrates on the mechanisms that can be used for making atomic nonblocking accesses to sets of memory locations. In practice there are two scenarios in which we envisage our APIs being used:

Firstly, our APIs may be used directly by expert programmers. For instance, when the code implementing a data structure is small and self-contained, then the shared-memory accesses in the code can be replaced directly with operations on WSTM, or the data structure's layout reorganized to use OSTM objects.

Secondly, and more generally, the mechanisms we have developed can form a layer within a complete system. For instance, transactional memory can be used as a basis for language features such conditional critical regions [Hoare 1972]. If this is implemented as part of a compiler, then it automates the typical way in which WSTM is used [Harris and Fraser 2003]. An alternative (in those languages which support it) is to use runtime code generation to add the level of indirection that programming using OSTM objects requires [Herlihy 2005]. Aside from CCRs, hybrid designs are possible which combine lock-based abstractions with optimistic execution over transactional memory [Welc et al. 2004].

A further direction, again beyond the scope of the current article, is using a language's type system to ensure that the *only* operations attempted within a transaction are made through an STM interface [Harris et al. 2005]. This can avoid programming errors in which irrevocable operations (such as network I/O) are performed in a transaction that subsequently aborts.

2.4.2 *Invalidity During Execution.* A nonobvious complication when using our APIs is that atomicity is only enforced at the point of a call to MCAS, WSTMCommitTransaction, or OSTMCommitTransaction. It remains possible for a thread to see a mutually inconsistent view of shared memory if it performs a series of MCASRead, WSTMRead, or OSTMOpen calls. This can happen due to atomic updates by other threads. Note that in all three APIs, arbitrary values cannot be returned: The value returned by MCASRead must have been current at some time during the call, and the values read through WSTMRead or OSTMOpenForReading must have been current at some point during the transaction.

If a thread does see a mutually inconsistent set of values, then its subsequent MCAS or STM commit operation will not succeed. However, it is possible that the inconsistency may cause an algorithm to crash, loop, or recurse deeply before it attempts to commit.

We do not seek to provide a general-purpose solution to this problem at the level of APIs in this article: The solution, we believe, depends on how the APIs are being used.

In the first scenario we discussed before—where one of our APIs is being used directly by an expert programmer—it is the responsibility of that programmer to consider the consequences of transactions that see inconsistent views of memory during their execution. Two different cases arise in the examples we have seen. Firstly, some simple algorithms are guaranteed to reach a call to MCAS or a STM commit operation even if they see a mutually inconsistent set of values; they can be used without modification. Secondly, other algorithms can loop internally, recurse deeply, or dereference NULL pointers if they see an inconsistent set of values. In these cases, we ensure that some validation is performed within every loop and function call and we use a signal handler to catch NULL-dereferences (explicit tests against NULL could be used in environments without signal handlers). We emphasize that these are only the cases that arise in the examples we have studied: Other programs using these APIs may need additional "hardening" against floating point exceptions, out-of-bound memory accesses, and so on, if these are possible due to sets of inconsistent reads.

As others have explored, an alternative approach would be to ensure validity throughout a transaction by performing work on WSTMRead and OSTMOpen operations. This ensures that the values seen within a transaction always form a mutually consistent snapshot of part of the heap. This is effectively the approach taken by Herlihy et al.'s [2003b] DSTM, and leads to the need either to make reads visible to other threads (making read parallelism difficult in a streamlined implementation) or to explicitly revalidate invisible reads (leading to $O(n^2)$ behavior when a transaction opens n objects in turn). Either of these approaches could be integrated with WSTM or OSTM if the API is to be exposed directly to programmers whilst shielding them from the need to consider invalidity during execution.

In the second scenario, where calls on the API are generated automatically by a compiler or language runtime system, we believe it is inappropriate for the application programmer to have to consider mutually inconsistent sets of values within a transaction. For instance, when considering the operational semantics of atomic blocks built over STM in Haskell [Harris et al. 2005], definitions where transactions run in isolation appear to be a clean fit with the existing language, while it is unclear how to define the semantics of atomic blocks that may expose inconsistency to the programmer. Researchers have explored a number of ways to shield the programmer from such inconsistency [Harris and Fraser 2003; Harris et al. 2005; Riegel et al. 2006; Dice et al. 2006]. These can broadly be classified as approaches based on hardening the runtime system against behavior due to inconsistent reads, and approaches based on preventing inconsistent reads from occuring. The selection between these goes beyond the scope of the current article.

2.4.3 *Optimizations and Hints.* The final aspect we consider is the availability of tuning facilities for a programmer to improve the performance of an algorithm using our APIs.

The key problem is *false contention*, where operations built using the APIs are deemed to conflict even though logically they commute. For instance, if a set of integers is held in numerical order in a linked list, then a thread transactionally inserting 15 between 10 and 20 will perform updates that conflict with reads from a thread searching through that point for a higher value.

It is not clear that this particular example can be improved automatically when a tool is generating calls on our APIs; realizing that the operations do not logically conflict relies on knowledge of their semantics and the set's representation. Notwithstanding this, the ideas of disjoint-access parallelism and read-parallelism allow the programmer to reason about which operations will be able to run concurrently without interfering with each other.

However, our APIs can be extended with operations for use by expert programmers. As with Herlihy et al.'s DSTM [2003b], our OSTM supports an additional *early release* operation that discards an object from the sets of accesses that the implementation uses for conflict detection. For instance, in our list example, a thread searching the list could release the lists, nodes as it traverses them, eventually trying to commit a minimal transaction containing only the node it seeks (if it exists) and its immediate predecessor in the list. Similarly, as we discuss in Section 6.5, WSTM supports *discard* operations to remove addresses from a transaction.

These operations all require great care: Once released or discarded, data plays no part in the transaction's commit or abort. A general technique for using them correctly is for the programmer to ensure that: (i) As a transaction runs, it always holds enough data for invalidity to be detected; and (ii) when a transaction commits, the operation it is performing is correct, given only the data that is still held. For instance, in the case of searching a sorted linked list, it would need to hold a pair of adjacent nodes to act as a "witness" of the operation's result. However, such extreme use of optimization APIs loses many of the benefits of performing atomic multiword updates (the linked-list example becomes comparably complex to a list built directly from CAS [Harris 2001] or sophisticated locking [Heller et al. 2005]).

3. RELATED WORK

The literature contains several designs for abstractions, such as MCAS, WSTM, and OSTM. However, many of the foundational designs have not shared our recent goals of practicality, for instance, much work builds on instructions such as strong-LL/SC or DCAS [Motorola 1985] which are not available as primitives in contemporary hardware. Our experience is that although this work has identified the problems which exist and has introduced terminology and conventions for presenting and reasoning about algorithms, it has not been possible to effectively implement or use these algorithms by layering them above software implementations of strong-LL/SC or DCAS. For instance, when considering strong-LL/SC, Jayanti and Petrovic's recent design reserves four words of storage *per thread* for each word that may be accessed [Jayanti and Petrovic 2003]. Other designs reserve N or $\log N$ bits of storage within each word when used with N threads: Such designs can only be used when N is small. When considering DCAS, it appears no easier to build a general-purpose DCAS operation than it is to implement our MCAS design.

In discussing related work, the section is split into three parts. Firstly, in Section 3.1 we introduce the terminology of nonblocking systems and describe the progress guarantees that they make. These properties underpin the liveness gurantees that are provided to users of our algorithms. Secondly, in Section 3.2 we discuss the design of "universal" transformations that build nonblocking systems from sequential or lock-based code. Finally, in Section 3.3, we present previous designs for multiword abstractions, such as MCAS, WSTM, and OSTM, and assess them against our goals.

3.1 Nonblocking Systems

Nonblocking algorithms have been studied as a way of avoiding the liveness problems that are possible when using traditional locks [Herlihy 1993]. A design is nonblocking if the suspension or failure of any number of threads cannot prevent the remainder of the system from making progress. This provides robustness against poor scheduling decisions, as well as against arbitrary thread termination. It naturally precludes the use of ordinary locks because unless a lock-holder continues to run, the lock will never be released.

Nonblocking algorithms can be classified according to the kind of progress guarantee that they make:

—*Obstruction-freedom* is the weakest guarantee: A thread performing an operation is only guaranteed to make progress so long as it does not contend with other threads for access to any location [Herlihy et al. 2003a]. 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 t2 encounters thread t1 obstructing it, then t2 helps t1 to complete t1's operation. Once that is done, t2 can proceed with its own operation and hopefully not be obstructed again. 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. It is seldom possible to directly develop wait-free algorithms that offer competitive practical performance. However, Fich et al. have recently developed a transformation which converts an obstruction-free algorithm into one that is wait-free in the unknown-bound semisynchronous model of computation [Fich et al. 2005].

Some previous work has used the terms "lock-free" and "nonblocking" interchangeably: We follow Herlihy et al.'s recent use of lock-freedom to denote a particular kind of nonblocking guarantee [Herlihy et al. 2003a]. In this article 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.

We must take care to deliberate over what it means for an implementation of an API like WSTM and OSTM to offer a given form of nonblocking progress: We care not just about the progress of individual operations on the API, but also about the progress of complete transactions through to successful WSTMCommitTransaction or OSTMCommitTransaction calls.

We say that an implementation of a transactional abstraction provides a given nonblocking progress guarantee if complete transactions running over it have that guarantee. For example, an obstruction-free transactional abstraction requires transactions to eventually commit successfully if run in isolation,[1] but allows a set of transactions to livelock, aborting one another if they contend. Similarly, a lock-free transactional abstraction requires some transaction to eventually commit successfully even if there is contention.

3.2 Universal Constructions

Universal constructions are a class of design technique that can transform a sequential data structure into one that is safe for concurrent usage. Herlihy's original scheme requires a shadow copy of the entire data structure to be taken.

[1]We say *eventually* because the transaction may have to be reexecuted before this occurs (just as an obstruction-free algorithm may involve internal retry steps to remove obstructions). Crucially, it cannot run in isolation an unbounded number of times before committing.

A thread then makes updates to this in private before attempting to make them visible by atomically updating a single "root" pointer of the structure [Herlihy 1993]. 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 original algorithm is replaced by an ownership reference which is either NULL 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 avoid blocking: instead, they execute instructions on behalf of the owner and then take ownership themselves. Interpreting a continuation is cumbersome: After each "instruction" is executed, a virtual program counter and a nonwrapping 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 pointers to *operation descriptors* [Barnes 1993]. Lock-based algorithms are converted to operate on shadow copies of the data structure; then, after determining the sequence of updates to apply, each "lock" is acquired in turn by making it point to the descriptor, the updates are performed on the structure itself, and finally the "locks" are released. Copying is avoided if contention is low by observing that the shadow 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 problem which places it beyond practical use. Another class of technique provides programming APIs which, although not automatic "fixes" to the problem of constructing nonblocking algorithms, make the task of implementing nonblocking data structures much easier compared with using atomic hardware primitives directly. The two best-known abstractions are multiword compare-and-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, which in turn is built from CAS [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. Although the space cost of implementing the required strong-LL/SC makes their design impractical, the identification of disjoint-access parallelism as a goal has remained a valuable contribution.

Anderson and Moir [1995] designed a wait-free version of MCAS that also requires strong-LL/SC. They improved on Israeli and Rappaport's space costs by constructing strong-LL/SC using $\log N$ reserved bits per updated memory location, rather than N. 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 streamlined 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 does not provide a read-parallel MCASRead.

Anderson et al. provide two 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 generally feasible.

Greenwald presents a simple MCAS design in his Ph.D. dissertation [1999]. This constructs a record describing the entire operation and installs it into a single shared location which indicates the sole in-progress MCAS. If installation is prevented by an existing MCAS operation, then the existing operation is helped to completion and its record is then removed. Once installed, an operation 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. This can be seen as a development of Turek's continuation-based scheme [Turek et al. 1992]. The use of a single shared installation point prevents the design from being disjoint-access parallel. Greenwald's subsequent technique of "two-handed emulation" generalized this scheme, 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 [1993]. They originally proposed a hardware design which leverages existing multiprocessor cache-coherency mechanisms.

Shavit and Touitou introduced the idea of implementing transactional memory in software [1995], showing how a lock-free transactional memory could be built from strong-LL/SC primitives. A notable feature is that they abort contending transactions rather than recursively helping them, as is usual in lock-free algorithms; lock-free progress 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 to subsequent STM designs.

Moir presents lock-free and wait-free STM designs with a dynamic programming interface [Moir 1997]. 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. The initial embodiment of this scheme required arbitrary-sized memory words and suffered 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.

Herlihy et al. have designed and implemented an obstruction-free STM concurrently with our work [Herlihy et al. 2003b]. It shares many of our goals. Firstly, the memory is dynamically sized: Memory blocks can be created and destroyed on-the-fly. Secondly, a practical implementation is provided which is built using CAS. Finally, the design is disjoint-access parallel and, in one implementation, transactional reads do not cause contended updates to occur in the underlying memory system. These features serve to significantly decrease contention in many multiprocessor applications, and are all shared with our lock-free OSTM. We include Herlihy et al.'s design in our performance evaluation in Section 8.

Recently, researchers have returned to the question of building various forms of hardware transactional memory (HTM) [Rajwar and Goodman 2002; Hammond et al. 2004; Ananian et al. 2005; Moore et al. 2005; McDonald et al. 2005; Rajwar et al. 2005]. While production implementations of these schemes are not available, and so it is hard to compare their performance with software systems, in many ways they can be seen as complementary to the development of STM. Firstly, if HTM becomes widely deployed, then effective STM implementations are necessary for machines without the new hardware features. Secondly, HTM designs either place limits on the size of transactions or fix policy decisions into hardware; STM provides flexibility for workloads that exceed those limits or benefit from different policies.

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 appears to occur atomically when implemented by a series of individual instructions accessing one word at-a-time. Our fundamental approach is to 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 the design of our implementation in a series of four steps:

(1) Define the format of the heap, the temporary data structures used, and how an application goes about allocating and deallocating memory for the 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 using a series of single-word accesses. This underpins the implementation of all functions other than the commit operations. In this step we are particularly concerned with ensuring nonblocking progress 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 contents 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 another, conflicting, commit operation. In this step we are concerned with ensuring nonblocking progress so that the progress is not prevented if, for example, the thread performing the existing commit operation has been preempted.

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, respectively.

4.1 Memory Formats

All three of our implementations introduce *descriptors* which: (i) set out the "before" and "after" versions of the memory accesses that a particular commit operation proposes to make, and (ii) provide 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 reused directly. This means that if a thread holds a reference to a given descriptor, then it can be sure that it has not been reused for another purpose.[2]

The second property is that, aside from its status field, a descriptor's contents are unchanged once it is made reachable from shared memory. This means that if one thread t1 encounters a descriptor allocated by another thread t2, then t1 can read a series of values from it and be sure of receiving mutually consistent results. For instance, in the case of an MCAS descriptor, t1 can read details both about a location that t2 accessed and the value that t2 proposes to write there.

The third property is that once the outcome of a particular commit operation has been decided, the descriptor's status field remains constant: If a thread

[2]This does not mean that the designs can only be used in languages that traditionally provide garbage collection. For instance, in our evaluation in Section 8.1.3, we use reference counting [Jones and Lins 1996] on the descriptors to allow prompt memory reuse and affinity between descriptors and threads.

wishes to retry a commit operation, for example, 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.

The combination of the first two properties is important because it allows us to avoid many A-B-A problems in which a thread is about to perform a CAS conditional on a location holding a value `A`, but then a series of operations by other threads changes the value to `B` and then back to `A`, allowing the delayed CAS to succeed. These two properties mean that there is effectively a one-to-one association between descriptor references and the intent to perform a given atomic update.

Our implementations rely on being able to distinguish pointers to descriptors from other values. In our pseudocode in Sections 5–7 we abstract these tests with predicates, for instance, IsMCASDesc to test if a pointer refers to an MCAS descriptor. We discuss ways in which these predicates can be implemented in Section 8.1.2.

4.2 Logical Contents

Each of our 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 a location is not in progress, then the latter is unowned and 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.

Each of our designs uses a different mechanism to represent the ownership relationship between locations and transactions. This forms the key distinction between them and we cover the details in Sections 5 (MCAS), 6 (WSTM), and 7 (OSTM).

4.3 Uncontended Commit Operations

The commit operations themselves are each structured in three stages. A first phase *acquires* exclusive ownership of the locations being updated, and a second *read-check* phase ensures that the locations which 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 occurs, in which the thread relinquishes ownership of the locations being updated.

There are four status values: UNDECIDED, READ-CHECK, SUCCESSFUL, and FAILED. 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 the relevant duration. At the decision point it is set to SUCCESSFUL if all required ownerships were acquired and the read-checks

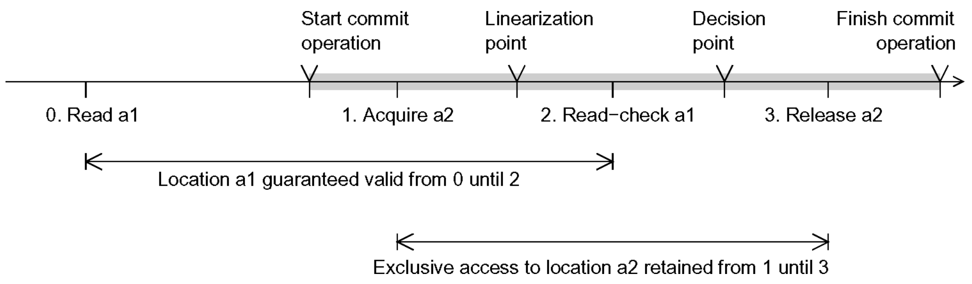


Fig. 6. Timeline for the three phases used in commit operations. The grey bar indicates when the commit operation is executed; prior to this, the thread prepares the heap accesses that it wants to commit. In this example location a1 has been read but not updated, and location a2 has been updated. The first phase acquires exclusive access to the locations being updated. The second phase checks that the locations read have not been updated by concurrent threads. The third phase releases exclusive access after making any updates. The read-check made at point 2 ensures that a1 is not updated between 0 and 2. The acquisition of a2 ensures exclusive access between 1 and 3.

succeeded; otherwise it is set to FAILED. These updates are always made using CAS operations. If a thread initiating a commit operation is helped by another thread, then both threads proceed through this series of steps, with the properties described in Section 4.1 ensuring that only one of these threads sets the status to SUCCESSFUL or FAILED.

In order to show that an entire commit operation appears atomic, we identify within its execution a *linearization point* at which it appears to operate atomically on the logical contents of the heap from the point of view of other threads.[3] There are two cases to consider, depending on whether an uncontended commit operation is successful:

Firstly, considering unsuccessful uncontended commit operations, the linearization point is straightforward: Some step of the commit operation observes a value that prevents the commit operation from succeeding, either a location that does not hold the expected value (in MCAS) or a value that has been written by a conflicting concurrent transaction (in WSTM and OSTM).

Secondly, considering successful uncontended commit operations, the linearization point depends on whether the algorithm has a read-check phase. Without a read-check phase the linearization point and decision point coincide: The algorithm has acquired ownership of the locations involved, and has not observed any values that prevent the commit from succeeding.

However, introducing a read-check phase makes the identification of a linearization point more complex. As Figure 6 shows, in this case the linearization point occurs at the *start* of the read-check phase, whereas the decision point (at which the outcome is actually signaled to other threads) occurs at the *end* of the read-check phase.

This choice of linearization point may appear perverse for two reasons:

(1) The linearization point comes *before* its decision point: How can an operation appear to commit its updates before its outcome is decided?

[3]In the presence of helping, the linearization point is defined with reference to the thread that successfully performs a CAS on the status field at the decision point.

The rationale for this is that holding ownership of the locations being updated ensures that these remain under the control of this descriptor from acquisition until release (1 until 3 in Figure 6). Similarly, read-checks ensure that any locations accessed in a read-only mode have not been updated[4] between points 0 and 2. Both of these intervals include the proposed linearization point, even though it precedes the decision point.

(2) If the operation occurs atomically at its linearization point, then what are the logical contents of the locations involved before the descriptor's status is updated at the decision point?

Following the definition in Section 4.2, the logical contents are dependent on the descriptor's status field, thus updates are not revealed to other threads until the decision point is reached. We reconcile this definition with the use of a read-check phase by ensuring that *concurrent readers help commit operations to complete*, retrying the read operation once the transaction has reached its decision point. This means that the logical contents do not need to be defined during the read-check phase because they are never required.

4.4 Contended Commit Operations

We now consider contended commit operations. In order to achieve nonblocking progress, we have to be careful about how to proceed when one thread t2 encounters a location that is currently owned by another thread t1. There are three cases to consider:

The first and most straightforward case is when t1's status is already decided, that is, if its status is SUCCESSFUL or FAILED. In this case, all of our designs rely on having t2 help t1 to complete its work, using the information in t1's descriptor to do so.

The second case is when t1's status is not decided and the algorithm *does not* include a READ-CHECK phase. In this case there are two general nonblocking strategies for handling contention with an UNDECIDED transaction:

—The first strategy is for t2 to cause t1 to abort if it has not yet reached its decision point; that is, if t1's status is still UNDECIDED. This leads to an obstruction-free progress property and the risk of livelock, unless contention management is employed to prevent t1 retrying its operation and aborting t2.

—The second strategy is for the threads to sort the locations that they require and for t2 to help t1 complete its operation, even if the outcome is currently UNDECIDED. This kind of *recursive helping* leads to a guarantee of lock-free progress because each recursive step involves a successively higher address, guaranteeing progress of the descriptor holding the highest contended address.

[4]Of course, the correctness of this argument does not allow the read-checks to simply consider the values in the locations because that would allow A-B-A problems to emerge if the locations are updated multiple times between 0 and 2. Our WSTM and OSTM designs which use read-check phases must check versioning information, rather than just values.

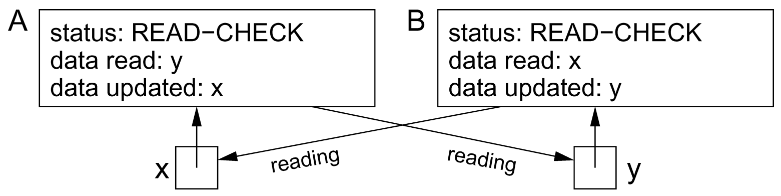


Fig. 7. An example of a dependent cycle of two operations A and B. Each needs the other to exit its read-check phase before it can complete its own.

The third and final case, and the most complicated, is when t1's status is not decided and the algorithm *does* include a READ-CHECK phase.

The complexity stems from the fact that, as we described in Section 4.3, a thread must acquire access to the locations that it is updating *before* it enters its READ-CHECK phase. This constraint on the order in which locations are accessed makes it impossible to eliminate cyclic helping by sorting accesses into a canonical order. Figure 7 shows an example: A has acquired data x for update and B has acquired data y for update, but A must wait for B before validating its read from y, while B in turn must wait for A before validating its read from x.

The solution is to abort at least one of the operations 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. For instance, with OSTM, this can be done by imposing a total order $\prec$ on all operations, based on the machine address of each transaction's descriptor. The loop is broken by allowing a transaction tx1 to abort a transaction tx2 if and only if: (i) both are in their read phase; (ii) tx2 owns a location that tx1 is attempting to read; and (iii) tx1 $\prec$ tx2. 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. MULTIWORD COMPARE-AND-SWAP (MCAS)

We now introduce our practical design for implementing the MCAS API. 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. Note that we define our MCAS API to work on memory locations containing *pointers*. As we described in Section 2.1, the MCAS implementation reserves two bits in each location that it may work on and so, in practice, these locations must hold aligned pointer values in which at least two low-order bits are ordinarily clear on a machine with 32-bit or 64-bit words. Sequentially, we define the MCAS operation as

```
atomically bool MCAS (int N, word **a[ ], word *e[ ], word *n[ ]) {
    for ( int i := 0; i < N; i++ ) if ( *a[i] ≠ e[i] ) return FALSE;
    for ( int i := 0; i < N; i++ ) *a[i] := n[i];
    return TRUE;
}
```

We initially present the implementation of MCAS using an intermediate *conditional compare-and-swap* operation. CCAS uses a second *conditional* memory location to control the execution of a normal CAS operation. If the conditional location holds the status value UNDECIDED, then the operation proceeds, otherwise CCAS has no effect. The conditional location may not itself be subject to updates by CCAS or MCAS. As with locations accessed by MCAS, two bits are reserved in locations updated by CCAS and so the function is defined to work on addresses holding pointers. Furthermore: (i) CCASRead operations must be used to read from locations that may be updated by a concurrent CCAS, and (ii) the implementation of CCAS must allow ordinary read, write, and CAS operations to be used on locations, so long as they are not being concurrently updated by CCAS.

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

This CCAS operation is a special case of the DCAS primitive that some processors have provided [Motorola 1985] and which has often been used in related work on building MCAS. However, unlike the more general DCAS (or even a double-compare single-swap), this restricted double-word operation has a straightforward implementation using CAS; we present this implementation in Section 5.4.

The implementation of MCAS is simpler than those of the two STMs because it does not involve a read-check phase. If the arrays passed to MCAS happen to specify the same value as e_i and n_i, then this is treated as an update between two identical values.

5.1 Memory Formats

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

```
typedef struct {
    word status;
    int N;
    word **a[MAX_N], *e[MAX_N], *n[MAX_N];
} mcas_descriptor;
```

A heap location is ordinarily unowned, in which case it holds the value logically stored there, or it refers to an MCAS (or CCAS) descriptor which is said to own it and which describes an MCAS (or CCAS) operation that it is being attempted on the location.

The type of a value read from a heap location can be tested using the IsMCASDesc and IsCCASDesc predicates: If either predicate evaluates true then the tested value is a pointer to the appropriate type of descriptor. As we describe in Section 8.1.2, we implement these functions by using reserved bits to distinguish the various kinds of descriptor from values being manipulated

```
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→e[i];
                    goto retry_read;
13              }
        return v;
15  }
```

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

by the application; This is why the MCAS implementation needs two reserved bits in the locations that it may update. However, for simplicity, in the pseudocode versions of our algorithms we use predicates to abstract these bitwise operations. Many alternative implementation techniques are available: For instance, some languages provide runtime type information, and in other cases descriptors of a given kind can be placed in given regions of the process' virtual address space.

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 SUCCESSFUL descriptor then the new value (n_i) is the logical contents.

The assumptions made about descriptor usage in Section 4.1 make it straightforward to determine the logical contents of a location because a series of words can be read from the descriptor without fear of it being deallocated or updated (other than the status field at the decision point).

Figure 8 presents this in pseudocode. If the location does not refer to a descriptor, then the 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 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 the before return forms the linearization point (line 8), and the recheck of ownership (lines 9 or 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.

Figure 10 presents the lock-free implementation of this algorithm in pseudocode. The first phase (lines 12–19) 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 the CCAS operation invoked on a_i must preserve the logical contents of the location: Either the CCAS fails (making no updates) or succeeds, installing a reference to a 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—this is needed in case a concurrent thread helps complete the MCAS operation via a call to mcas_help at line 18.

Note that this lock-free implementation must acquire updated locations in address order. As we explained in Section 4.4, this 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. If we do not want nonblocking progress, then we could omit the sorting step and abort an MCAS operation if it encounters contention (branch from line 17 to line 21 irrespective of the value seen in v).

The first phase terminates when the loop has completed each location (meaning that the descriptor has been installed in each of them (line 16)), when an unexpected nondescriptor value is seen (line 17), or when the descriptor's status is no longer UNDECIDED (line 16 or 17, because another thread has completed the first phase).

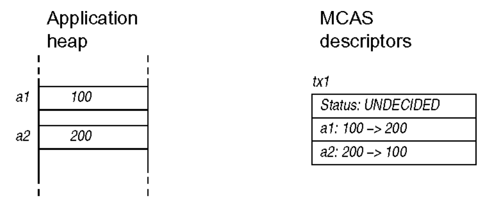
The first thread to reach the decision point for a descriptor must succeed in installing SUCCESSFUL or FAILED. If the MCAS has failed then the linearization point of the first CCAS that failed for this descriptor forms the linearization point of the MCAS operation: The unexpected value was the logical contents of the location and contradicts the expected value e_i for that location. Otherwise, if the MCAS has succeeded, note that when the status field is updated (line 23) then *all* of the locations a_i must hold references to the descriptor, and consequently the single status update changes the logical contents of all the locations. This is because the update is made by the first thread to reach line 23 for the descriptor and so no threads can yet have reached lines 25–27 nor started releasing the addresses.

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.

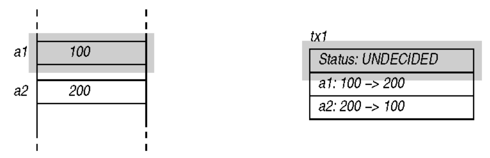
5.4 Building Conditional Compare-and-Swap

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

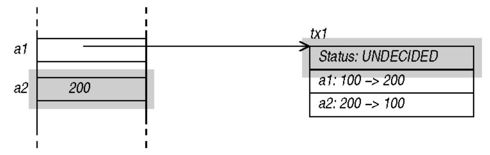
Figure 11 shows how CCAS and CCASRead can be implemented using CAS. Figure 12 illustrates this graphically.



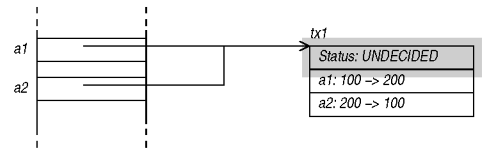
(a) The operation executes in private until it invokes MCAS. The MCAS descriptor holds the updates being proposed: In this case the contents of a1 and a2 are to be swapped.



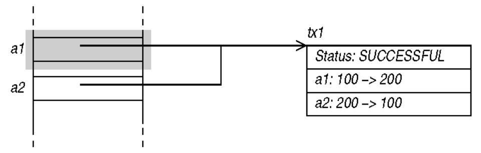
(b) CCAS is used to acquire ownership of addresses a1, replacing the value expected there with a reference to the MCAS descriptor. The update is conditional on the descriptor remaining UNDECIDED in order to guarantee that the location's logical contents do not change.



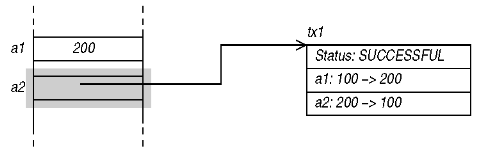
(c) Similarly, CCAS is used to acquire ownership of addresses a2.



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



(e) Ownership is released on a1, installing the new value.



(f) Similarly, ownership is released on a2, installing the new value.

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 through the MCAS descriptor.

```
bool MCAS (int N, word **a[ ], word *e[ ], word *n[ ]) {
    mcas_descriptor *d := new mcas_descriptor();
    (d→N, d→status) := (N, UNDECIDED);
    for ( int i := 0; i < N; i++ )
        (d→a[i], d→e[i], s→n[i]) := (a[i], e[i], n[i])
    address_sort(d); // Memory locations must be sorted into address order
    return mcas_help(d);
}
bool mcas_help (mcas_descriptor *d) {
    word *v, desired := FAILED;
    bool success;
    /* PHASE 1: Attempt to acquire each location in turn. */
    for ( int i := 0; i < d→N; i++ )
        while ( TRUE ) {
            v := CCAS(d→a[i], d→e[i], d, &d→status);
            if ( (v = d→e[i]) ∨ (v = d) ) break;
            if ( ¬IsMCASDesc(v) ) goto decision_point;
            mcas_help((mcas_descriptor *)v);
        }
    desired := SUCCESSFUL;
    /* PHASE 2: No read-phase is used in MCAS */
decision_point:
    CAS(&d→status, UNDECIDED, desired);
    /* PHASE 3: Release each location that we hold. */
    success := (d→status = SUCCESSFUL);
    for ( int i := 0; i < d→N; i++ )
        CAS(d→a[i], d, success ? d→n[i] : d→e[i]);
    return success;
}
```

Fig. 10. MCAS operation.

CCAS proceeds by attempting to install a *CCAS descriptor* in the location to be updated (line 8, Figure 12(a)). The descriptor is used to ensure that the location's logical contents match the expected value while the conditional location is being tested. If the descriptor fails to be installed at line 8 then either: (i) A descriptor for another CCAS was seen (line 10), which is helped to completion before retrying the proposed CCAS, or (ii) the expected value was not seen (line 9), in which case the failed CAS at line 8 forms the linearization point of the CCAS.

If the update location is successfully acquired, the conditional location is tested (line 24, Figure 12(b)). Depending on the contents of this location, the descriptor is either replaced with the new value or restored to the original value (line 25, Figure 12(c)). CAS is used so that this update is performed exactly once, even when the CCAS operation is helped to completion by other processes. A CCAS operation whose descriptor is installed successfully linearizes when the conditional location is read by the thread whose update at line 25 succeeds.

There is an interesting subtle aspect of the design and implementation of CCAS: It does not return a Boolean indicating whether it succeeded. Our MCAS algorithm does not need such a return value and, in fact, it is nontrivial to extend CCAS to provide one. The reason is that one thread's CCAS operation may be helped to completion by another thread and so it would be necessary to

```
 1  typedef struct {
        word **a, *e, *n, *cond;
 3  } ccas_descriptor;
    word *CCAS (word **a, word *e, word *n, word *cond) {
 5      ccas_descriptor *d := new ccas_descriptor();
        word *v;
 7      (d→a, d→e, d→n, d→cond) := (a, e, n, cond);
        while ( (v := CAS(d→a, d→e, d)) ≠ d→e ) {
 9          if ( ¬IsCCASDesc(v) ) return v;
            CCASHelp((ccas_descriptor *)v);
11      }
        CCASHelp(d);
13      return v;
    }
15  word CCASRead (word *a) {
        word *v := *a;
17      while ( IsCCASDesc(v) ) {
            CCASHelp((ccas_descriptor *)v);
19          v := *a;
        }
21      return v;
    }
23  void CCASHelp (ccas_descriptor *d) {
        bool success := (*d→cond = UNDECIDED);
25      CAS(d→a, d, success ? d→n : d→e);
    }
```

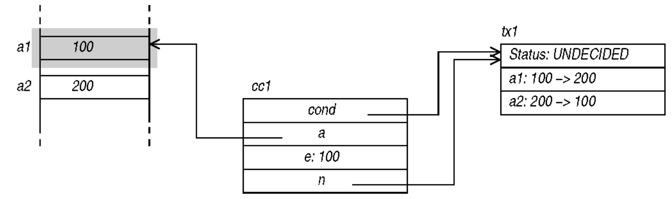
Fig. 11. Conditional compare-and-swap (CCAS). CCASRead is used to read from locations which may be subject to concurrent CCAS operations.

communicate the success/failure result back to the first thread. This cannot be done by extending the descriptor with a Boolean field for the result: There may be multiple threads concurrently helping the same descriptor, each executing line 25 with *different* result values.

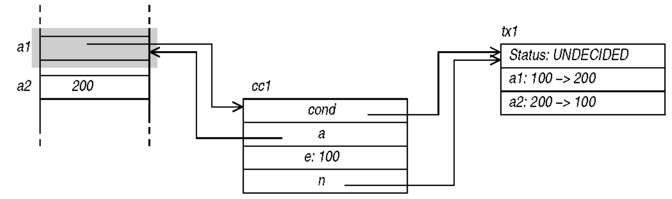
CCASRead proceeds by reading the contents of the supplied address. If this is a CCAS descriptor, then the descriptor's operation is helped to completion and the CCASRead retried. CCASRead returns a nondescriptor value once one is read. Notice that CCASHelp ensures that the descriptor passed to it has been removed from the address by the time that it returns, so CCASRead can only loop while other threads are performing new CCAS operations on the same address.

5.5 Discussion

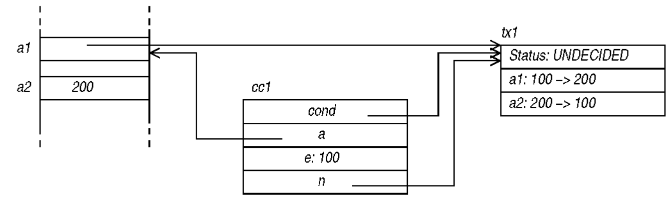
There are a number of final points to consider in our design for MCAS. The first is to observe that when committing an update to a set of N locations and proceeding without experiencing contention, the basic operation performs $3N + 1$ updates using CAS: $2N$ CAS operations are performed by the calls to CCAS, N CAS operations are performed when releasing ownership, and a further single CAS is used to update the status field. However, although this is more than a factor of three increase over updating the locations directly,



(a) CAS is used to try to install a pointer to the CCAS descriptor into the update location, in this case replacing the expected value 100.



(b) The conditional location is checked against 0 and the thread (or threads) acting on this CCAS descriptor individually decide the outcome and use CAS to store the new (if the check succeeds) or expected value (if the check fails).



(c) Note that the one-shot use of descriptors means that once one thread has removed the reference to the CCAS descriptor then any concurrent threads' CAS operations attempting to do so will fail: This is why there is no need for consensus between concurrent threads helping with a CCAS.

Fig. 12. The steps involved in performing the first CCAS operation needed in Figure 9. In this case the first location a1 is being updated from 100 to refer to MCAS descriptor tx1. The update is conditional on the descriptor tx1 being UNDECIDED.

it is worth noting that the three batches of N updates all act on the same locations: Unless evicted during the MCAS operation, the cache lines holding these locations need only be fetched once.

We did develop an alternative implementation of CCAS which uses an ordinary write in place of its second CAS. This involves leaving the CCAS descriptor linked into the location being updated and recording the success or failure of the CCAS within that descriptor. This $2N + 1$ scheme is not a worthwhile improvement over the $3N + 1$ design: It writes to more distinct cache lines and makes it difficult to reuse CCAS descriptors in the way we describe in Section 8.1.3. However, this direction may be useful if there are systems in which CAS operates substantially more slowly than an ordinary write.

Moir explained how to build an obstruction-free $2N + 1$ MCAS which follows the same general structure as our lock-free $3N + 1$ design [Moir 2002]. His design uses CAS in place of CCAS to acquire ownership while still preserving the logical contents of the location being updated. The weaker progress guarantee makes this possible by avoiding recursive helping: If t2 encounters t1 performing an MCAS then t2 causes t1's operation to abort if it is still UNDECIDED. This avoids the need to CCAS because only the thread initiating an MCAS can now update its status field to SUCCESSFUL: There is no need to check it upon each acquisition.

Finally, notice that algorithms built over MCAS will not meet the goal of read-parallelism from Section 1.1. This is because MCAS must still perform CAS operations on addresses for which identical old and new values are supplied: These CAS operations force the address's cache line to be held in exclusive mode on the processor executing the MCAS.

6. WORD-BASED SOFTWARE TRANSACTIONAL MEMORY

We now turn to the word-based software transactional memory (WSTM) that we have developed. WSTM improves on the MCAS API from Section 5 in three ways: (i) by removing the requirement that space be reserved in each location in the heap, (ii) by presenting an interface in which the WSTM implementation is responsible for tracking the locations accessed, rather than the caller, and (iii) by providing read parallelism: Locations that are read (but not updated) can usually remain cached in shared mode.

Unfortunately, the cost of this is that the WSTM implementation is substantially more complex. Due to this complexity we split our presentation of WSTM into three stages: firstly presenting the core framework without support for resolving contention (Sections 6.1–6.3), then showing how to provide a basic lock-based instantiation of this framework, and then showing how to provide one that supports obstruction-free transactions (Section 6.4).

The complexity of WSTM motivates the simpler obstruction-free OSTM in Section 7.

6.1 Memory Formats

WSTM is based on the idea of associating version numbers with locations in the heap and using updates to these version numbers as a way of detecting conflicts between transactions.

Figure 13 illustrates this with an example. The *application heap* comprises the memory locations holding data being manipulated through the WSTM API; in this case values 100, 200, 300, and 400 at addresses a1, a2, a101, and a102, respectively.

A table of *ownership records* (orecs) hold the information that WSTM uses to coordinate access to the application heap. The orec table has a fixed size. A hash function is used to map addresses to orecs. One possible hash function is to use a number of the low-order bits of the address: In this case addresses a1 and a101 map to r1 while addresses a2 and a102 map to r2.

In the basic design each orec either holds a *version number* or a reference to a *transaction descriptor*. It holds a version number when the orec is *quiescent*, that is, when no transaction is trying to commit an update to a location with which it is associated. For example, orec r2 is quiescent. Alternatively,

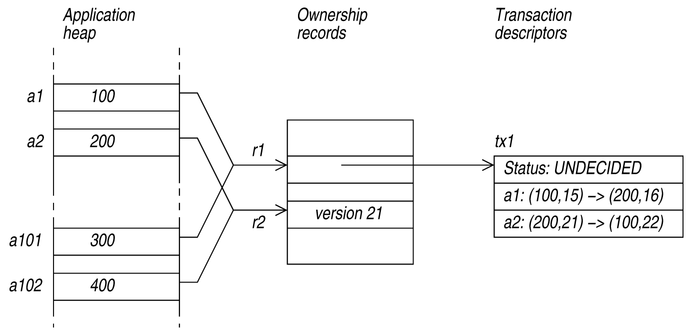


Fig. 13. The heap structure used in the lock-based WSTM. The commit operation acting on descriptor tx1 is midway through an update to locations a1 and a2: 200 is being written to a1 and 100 to a2. Locations a101 and a102 are examples of other locations which happen to map to the same ownership records, but which are not part of the update.

when an update is being committed, the orec refers to the descriptor of the transaction involved. In the figure, transaction tx1 is committing an update to addresses a1 and a2; it has acquired orec r1 and is about to acquire r2. Within the transaction descriptor, 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 value o_i at version number vo_i to 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, $vn_i = vo_i + 1$.

Figure 14 shows the definition of the data types involved. A wstm_transaction comprises a status field and a list of wstm_entry structures. As indicated in the figure, each entry provides the old and new values and old and new version numbers for a given memory access. In addition, the obstruction-free version of WSTM includes a prev_ownership field to coordinate helping between transactions.

The lock-based WSTM uses orecs of type orec_basic: Each orec is a simple union holding either a version field or a reference to the owning transaction. The obstruction-free WSTM requires orecs of type orec_obsfree holding either a version field or a pair containing an integer count alongside a reference to the owning transaction.[5] In both implementations the current usage of a union can be determined by applying the predicate IsWSTMDesc: If it evaluates true then the orec contains a reference to a transaction, else it contains a version number. As before, these predicates can be implemented by a reserved bit in the space holding the union.

A descriptor is *well formed* if, for each orec, 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, as is the new.

[5]The maximum count needed is the maximum number of concurrent commit operations. In practice, this means that orecs in the obstruction-free design are two-words wide. Both IA-32 and 32-bit SPARC provide double-word-width CAS. On 64-bit machines without double-word-width CAS, it is possible either to reserve sufficient high-order bits in a sparse 64-bit address space or to add a level of indirection between orecs and temporary double-word structures.

```
typedef struct {
    word status; // UNDECIDED, READ_CHECK, SUCCESSFUL, or FAILED
    list<wstm_entry*> *entries;
} wstm_transaction;
typedef struct {
    word *addr;
    word old_value, old_version;
    word new_value, new_version;
    orec_obsfree prev_ownership; // Only used in obstruction-free variant
} wstm_entry;
typedef union {
    word version;
    wstm_transaction* owner;
} orec_basic; // Used in the basic variant
typedef union {
    word version;
    <word, wstm_transaction*> <count, owner>;
} orec_obsfree; // Used in the obstruction-free variant
// Ensure tx outcome is decided - either FAILED or SUCCESSFUL
void force_decision(wstm_transaction *tx) {
    CAS (&tx → status, UNDECIDED, FAILED);
    CAS (&tx → status, READ_CHECK, FAILED);
}
// Mapping from addresses to orecs
orec *addr_to_orec(word *addr);
// Search a list of wstm_entry structures for the first one relating
// to a specified address (search_addr), or for the first one
// relating to any address associated with a specified orec
// (search_orec). In either case, a NULL return indicates that there
// is no match.
wstm_entry *search_addr(list<wstm_entry*> *entries, word *addr);
wstm_entry *search_orec(list<wstm_entry*> *entries, orec *orec_ptr);
// Filter a list of wstm_entry structures to obtain sub-lists
// relating to a given orec, or containing only reads (same
// old_version and new_version) or only updates (different old_version
// and new_version)
list<wstm_entry*> *filter_for_orec(list<wstm_entry*> *entries, orec *orec_ptr);
list<wstm_entry*> *filter_for_updates(list<wstm_entry*> *entries);
list<wstm_entry*> *filter_for_reads(list<wstm_entry*> *entries);
```

Fig. 14. Data structures and helper functions used in the WSTM implementation.

6.2 Logical Contents

As with MCAS, we proceed by defining the logical contents of a location in the heap. There are three cases to consider:

LS1 : If the orec holds a version number then the logical contents comes directly from the application heap. For instance, in Figure 13, the logical contents of a2 is 200.

LS2 : If the orec refers to a descriptor that contains an entry for the address, then that entry gives the logical contents (taking the new value from the entry if the descriptor is SUCCESSFUL, and the old value if it is

UNDECIDED or FAILED). For instance, the logical contents of a1 is 100 because the descriptor status is UNDECIDED.

LS3 : If the orec refers to a descriptor that does not contain an entry for the address then the logical contents come from the application heap. For instance, the logical contents of a101 is 300 because descriptor tx1 does not involve a101 even though it owns r1.

Figure 15 shows how the logical contents of a location are determined in the WSTMRead and WSTMWrite functions. As usual, since we do not define the logical contents during a READ-CHECK phase, we rely on threads encountering such a descriptor to help decide its outcome (reaching states SUCCESSFUL or FAILED and hence LS2 if an entry is found at line 15, or LS3 if not).

Both of these are built over get_entry, which finds (or adds) a wstm_entry structure to the given list of entries. get_entry begins by checking whether the given address is already in the list (lines 2–3). If not then a new entry is needed, holding the logical contents along with the version number associated with that value. Lines 5–29 determine this following the structure of cases LS1–3. The properties from Section 4.1 are key to allowing one thread to read another's transaction descriptor: Recall that aside from the status field, the descriptor is unchanged once made reachable in shared storage. Note that we call force_decision at line 14 to ensure that any other descriptor we encounter is in either the FAILED or SUCCESSFUL state; this ensures that: (i) The descriptor cannot change status while we are reading from it, and (ii) as in Section 4.4, we do not read from a descriptor in the READ-CHECK state, as the logical contents cannot be determined.

Lines 32–38 ensure that the descriptor will remain well formed when the new entry is added. This is done by searching for any other entries relating to the same orec (line 33). If there is an existing entry then the old version number is examined (line 34). If the numbers do not match then a concurrent transaction has committed an update to a location involved with the same orec: tx is doomed to fail (line 35). Line 36 ensures the descriptor remains well formed even if it is doomed to fail. Line 37 ensures that the new entry has the same new version as existing entries, for example, if there was an earlier WSTMWrite in the same transaction that was made to another address associated with this orec.

WSTMRead (lines 42–46) simply returns the new value for the entry for addr.

WSTMWrite (lines 47–53) updates the entry's new value (lines 48–50). It must ensure that the new version number indicates that the orec has been updated (lines 51–52), both in the entry for addr and, to ensure well-formedness, in any other entries for the same orec.

6.3 Uncontended Commit Operations

Figure 16 shows the overall structure of WSTMCommitTransaction built using helper functions for actually acquiring and releasing ownership of orecs. The structure follows the design method in Section 4: The orecs associated with updated locations are acquired (lines 25–27), then a read-check phase checks that the version numbers in orecs associated with reads are still current (lines 28–31). If both phases succeed then the descriptor status is attempted to

```
wstm_entry *get_entry(list<wstm_entry*> *entries, word *addr) {
  wstm_entry *entry := search_addr(entries, addr); // First try to find an existing entry
  if (entry ≠ NULL) return entry;
  // There's no existing entry: create a new one. First find the logical contents:
  word value, version;
  do {
    orec *orec_ptr := addr_to_orec(addr);
    orec orec_val := *orec_ptr;
    if (¬IsOSTMDesc(orec_val)) { // LS1
      value := *addr;
      version := orec_val.version;
    } else {
      owner := orec_val.owner;
      force_decision(owner);
      if ((owners_entry := search_addr(owner → entries, addr)) ≠ NULL) { // LS2
        if (owner → status = SUCCESSFUL)
          value := owners_entry → new_value;
          version := owners_entry → new_version;
        else
          value := owners_entry → old_value;
          version := owners_entry → old_version;
      } else { // LS3
        owners_entry := search_orec(owner → entries, orec_ptr);
        version := (owner → status = SUCCESSFUL) ? owners_entry → new_version
                                                 : owners_entry → old_version;
        value := *addr;
      }
    }
  } while (orec_val ≠ *orec_ptr); // Keep retrying until version or ownership unchanged
  // Tentatively create the new entry, same value & version as both the old and new state:
  entry := new wstm_entry(addr, value, version, value, version);
  // Ensure descriptor remains well-formed
  if ((aliasing_entry := search_orec(entries, orec_ptr)) ≠ NULL) {
    if (aliasing_entry → old_version ≠ version)
      tx → status := FAILED; // Inconsistent reads
    entry → old_version := aliasing_entry → old_version;
    entry → new_version := aliasing_entry → new_version;
  }
  append(entry, entries); // Add new entry to the list
  return entry;
}
word WSTMRead(wstm_transaction *tx, word *addr) {
  wstm_entry *entry;
  entry := get_entry(tx → entries, addr);
  return (entry → new_value);
}
void WSTMWrite(wstm_transaction *tx, word *addr, word new_value) {
  wstm_entry *entry;
  entry := get_entry(tx → entries, addr);
  entry → new_value := new_value;
  for ( entry in filter_for_orec(tx → entries, addr_to_orec(addr))) {
    entry → new_version := entry → old_version + 1;
  }
}
```

Fig. 15. WSTMRead and WSTMWrite functions built over get_entry.

```
 1  // Helper function signatures
    bool prepare_descriptor(wstm_transaction *tx);
 3  bool acquire_orec(wstm_transaction *tx, wstm_entry *entry);
    void release_orec(wstm_transaction *tx, wstm_entry *entry);

 5  // Common read-check function
    bool read_check_orec(wstm_transaction *tx, wstm_entry *entry) {
 7      orec *orec_ptr := addr_to_orec(entry → addr);
        orec seen := *orec_ptr;
 9      word version;
        if (IsWSTMDesc(seen)) {
11          wstm_transaction *owner := seen.owner;
            force_decision(owner);
13          owners_entry := search_orec(owner → entries, orec_ptr);
            version := (owner → status = SUCCESSFUL) ? owners_entry → new_version
15                                                    : owners_entry → old_version;
        } else {
17          version := seen.version;
        }
19      return (version = entry → old_version);
    }

21  // Main commit function
    bool WSTMCommitTransaction(wstm_transaction *tx) {
23      if (tx → status = FAILED) return false; /* Fail early if inconsistent */
        if (¬prepare_descriptor(tx)) return false;
25      word desired_status := FAILED;
        for ( wstm_entry *entry in filter_for_updates(tx → entries)) /* Acquire phase */
27          if (¬acquire_orec(tx, entry)) goto decision_point;
        CAS (&tx → status, UNDECIDED, READ_CHECK);
29      for ( wstm_entry *entry in filter_for_reads(tx → entries)) { /* Read phase */
            if (¬read_check_orec(tx, entry)) goto decision_point;
31      desired_status := SUCCESSFUL; // Acquire succeeded, read-check succeeded
    decision_point:
33      while ( ((status := tx → status) ≠ FAILED) ∧ (status ≠ SUCCESSFUL))
            CAS (&tx → status, status, desired_status);
35      if (tx → status = SUCCESSFUL)
            for ( wstm_entry *entry in filter_for_updates(tx → entries)) /* Make updates */
37              *(entry → addr) := entry → new_value;
        for ( wstm_entry *entry in filter_for_updates(tx → entries)) /* Release phase */
39          release_orec(tx, entry);
        return (tx → status = SUCCESSFUL);
41  }
```

Fig. 16. Basic implementation of WSTMCommitTransaction using helper functions to manage orecs.

be set to SUCCESSFUL (lines 33–34) and if successful, the updates are made (lines 35–37). Finally, ownership of the orecs is released (lines 38–39). Notice how the definition of LS2 means that setting the status to SUCCESSFUL atomically updates the logical contents of all of the locations written by the transaction.

The read-check phase uses read_check_orec to check that the current version number associated with an orec matches the old_version in the entries in a transaction descriptor. As with get_entry in Figure 15, if it encounters another transaction descriptor then it ensures that its outcome is decided (line 12) before examining it.

```
 1   bool **prepare_descriptor**(wstm_transaction *tx) {
         sort(tx → entries); // Sort entries to avoid deadlocks
 3       return TRUE;
     }
 5   bool acquire_orec(wstm_transaction *tx, wstm_entry *entry) {
         orec *orec_ptr := addr_to_orec(entry → addr);
 7       orec seen := *orec_ptr;
         // Already owned by us
 9       if (IsWSTMDesc(seen) ∧ (seen.owner = tx)) return TRUE;
         // To be acquired by us
11       do {
             seen := CAS(orec_ptr, entry → old_version, tx);
13       } while (IsWSTMDesc(seen));
         return (seen.version = entry → old_version);
15   }
     void release_orec(wstm_transaction *tx, wstm_entry *entry) {
17       bool succeeded := (tx → status = SUCCESSFUL);
         orec *orec_ptr := addr_to_orec(entry → addr);
19       orec seen := *orec_ptr;
         if (IsWSTMDesc(seen) ∧ (seen.owner = tx)) {
21           *orec_ptr.version := succeeded ? entry → new_version : entry → old_version;
         }
23   }
```

Fig. 17. Basic lock-based implementation of the helper functions for WSTMCommitTransaction.

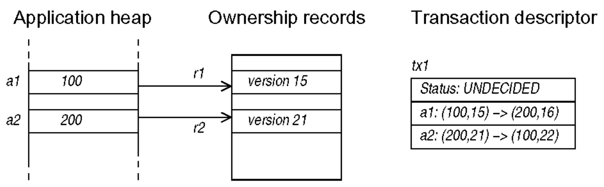
In our lock-based WSTM implementation the helper functions prepare_descriptor, acquire_orec, and release_orec are straightforward, as shown in Figure 16.

This implementation is based on using the orecs as mutual exclusion locks, allowing at most one transaction to own an orec at a given time. A transaction owns an orec when the orec contains a pointer to the transaction descriptor. To avoid deadlock, prepare_descriptor (lines 1–4) ensures that the entries are sorted, for instance, by address. Ownership acquisition involves two cases: (i) The orec is owned by tx because its descriptor holds multiple entries for the same orec (line 9), or (ii) the orec is not owned by tx, in which case CAS is used to replace the current transaction's old version number with a reference to its descriptor (lines 11–13). Note that the loop at line 13 will spin while the value seen in the orec is another transaction's descriptor. Releasing an orec is straightforward: Mutual exclusion allows us to directly update any orecs that we acquired (lines 19–22).

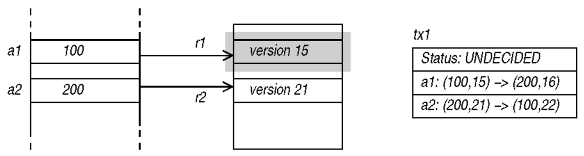
Figure 18 shows graphically how an uncontended commit operation can proceed for the transaction in Figure 13.

6.4 Obstruction-Free Contended Commit Operations

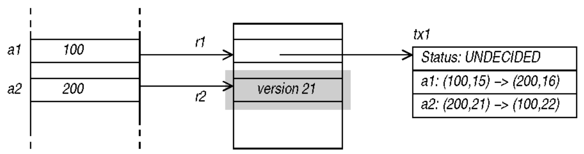
Designing a nonblocking way to resolve contention in WSTM is more complex than in MCAS. This is because in MCAS, it was possible to resolve contention by having one thread *help* another to complete its work. The key problem with WSTM is that a thread *cannot* be helped while it is writing updates to the heap (lines 35–37 of Figure 16, or the steps to be performed in Figure 18(e)).



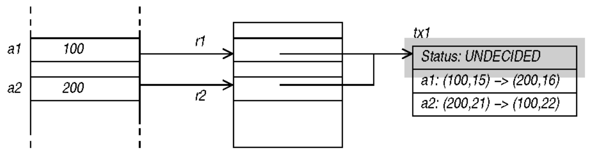
(a) The transaction executes in private until it attempts to commit.



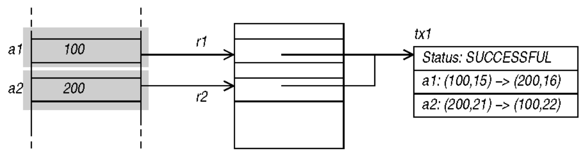
(b) CAS is used to acquire ownership record r1, replacing the expected version number with a pointer to the transaction descriptor.



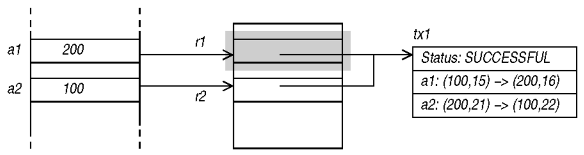
(c) Similarly, CAS is used to acquire ownership of record r2.



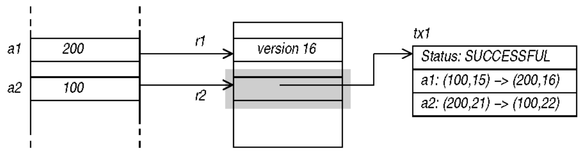
(d) CAS is used to set the status to SUCCESSFUL.



(e) The updates are written back to the heap. There is no need for the two writes to be atomic with one another.



(f) Ownership is released on r1, installing the new version number.



(g) Similarly, ownership is released on r2.

Fig. 18. An uncontended commit swapping the contents of a1 and a2, showing where updates are to be performed at each step.

If a thread is preempted just before one of the stores in line 37, then it can be rescheduled *at any time* and perform that *delayed update* [Harris and Fraser 2005], overwriting updates from subsequent transactions.

Aside from the lock-based scheme from the previous section and the complicated obstruction-free scheme we will present here, there are two further solutions that we note for completeness:

(1) As described in an earlier paper [Harris and Fraser 2005], operating system support can be used either to: (i) prevent a thread from being preempted during its update phase, or (ii) to ensure that a thread preempted while making its updates will remain suspended while being helped and then be resumed at a *safe point*, for example, line 40 in WSTMCommitTransaction with a new status value SUCCESSFUL_HELPED, indicating that the transaction has been helped (with the helper having made its updates and released its ownership).

(2) CCAS from Section 5 can be used to make the updates in line 37, conditional on the transaction descriptor still being SUCCESSFUL rather than SUCCESSFUL_HELPED. Of course, using CCAS would require reserved space in each word in the application heap, negating a major benefit of WSTM over MCAS.

The approach we take to making an obstruction-free WSTM without using CCAS or operating system support is to make delayed updates benign by ensuring that an orec remains owned by some transaction while it is possible that delayed writes may occur at locations associated with it. This means that the logical contents of locations that may be subject to delayed updates are taken from the owning transaction (under case LS2 of Section 6.2).

This is coordinated through the orec's count field that we introduced in Section 6.1: An orec's count is increased each time a thread successfully acquires ownership in the obstruction-free variant of acquire_orec. The count is decreased each time a thread releases ownership in the obstruction-free variant of release_orec. A count of zero therefore means that no thread is in its update phase for locations related to the orec.

The main complexity in our obstruction-free design comes from allowing ownership to transfer directly from one descriptor to another: Notice that if a thread committing one transaction, say t2 performing tx2, encounters an orec that has been acquired by another transaction, say tx1, then t2 cannot simply replace the <count, owner> pair <1, tx1> with <2, tx2>. The reason is that addresses whose logical contents were determined from tx1 under case LS2 may change because tx2 may not have entries for these addresses.

Figure 19 shows the obstruction-free prepare_descriptor function that enables safe ownership stealing. This is based on the idea that before a transaction tx starts acquiring any ownerships, its descriptor is extended to include relevant entries from the descriptors from which it may need to steal. This means that the logical contents of these entries' addresses will be unchanged if stealing occurs. Of course, good runtime performance depends on contention, hence stealing, being rare.

```
1   bool prepare_descriptor(wstm_transaction *tx) {
        // Build set of orecs to acquire
3       set<orec*> *orecs_to_acquire := new set<orec*>;
        for (wstm_entry *entry in filter_for_updates(tx → entries)) {
5           orec *orec_ptr := addr_to_orec(entry → addr);
            entry → prev_ownership := orec;
7           insert(orecs_to_acquire, orec_ptr);
        }
9       // Merge entries from the current owners of orecs_to_acquire
        for (orec *orec_ptr in orecs_to_acquire) {
11          orec seen := *orec_ptr;
            if (IsWSTMDesc(seen)) {
13              wstm_transaction *owner := seen.owner;
                force_decision(owner);
15              for (oe in filter_for_orec(owner → entries, orec_ptr)) {
                    if (owner → status = SUCCESSFUL) {
17                      value := oe → new_value;
                        version := oe → new_version;
19                  } else {
                        value := oe → old_value;
21                      version := oe → old_version;
                    }
23                  wstm_entry *our_entry := search_orec(tx → entries, orec_ptr);
                    if (version ≠ our_entry → old_version) goto fail;
25                  if (¬search_addr(tx → entries, oe → addr)) {
                        wstm_entry *new_entry := new wstm_entry(oe → addr,
27                              our_entry → old_version, value, our_entry → new_version, value);
                        new_entry → prev_ownership := seen;
29                      append(new_entry, tx → entries);
                    }
31              }
            }
33      }
        return TRUE;
35  fail:
        tx → status := FAILED;
37      return FALSE;
    }
```

Fig. 19. Obstruction-free implementation of the prepare_descriptor helper function for WSTMCommitTransaction.

prepare_descriptor works in two stages. Firstly, it builds up a set containing the orecs that it will need to acquire (lines 2–8), recording the previous value so that if stealing is necessary, it is possible to check that the ownership has not changed subsequent to the value being stashed in prev_ownership at line 6. Secondly, it examines whether each of these orecs is currently owned (lines 11–12). If it is owned then the owner is forced to a decision so that its descriptor can be examined (line 14), and the descriptor is searched for entries relating to the orec. For each such entry, the logical contents and version number are determined (lines 16–22), and it is checked that tx will remain wellformed if those entries are merged into it (lines 23–24), otherwise tx is doomed to fail (lines 35–37). Assuming tx would remain well formed, tx is checked for an existing entry for the same address (line 25). If there is an existing entry then tx's old value and version are the same as those of the owning transaction (by the

```
1   bool acquire_orec(wstm_transaction *tx, wstm_entry *entry) {
        orec *orec_ptr := addr_to_orec(entry → addr);
3       orec seen := *orec_ptr;
        // Already owned by us
5       if (IsWSTMDesc(seen) ∧ (seen.owner = tx)) return TRUE;
        // To be acquired by us (no previous owner)
7       if (¬IsWSTMDesc(entry → prev_owner.version)) {
            seen := CAS(orec_ptr, <0, entry → old_version>, <1, tx>);
9           bool success := (seen = <0, entry → old_version>);
            if (success) entry → prev_ownership := tx; // Record that we acquired it
11          return success;
        }
13      // To be acquired by us (from previous owner)
        word new_count := (entry → prev_ownership.count) + 1;
15      seen := CAS(orec_ptr, entry → prev_ownership, <new_count, tx>);
        bool success := (seen = entry → prev_ownership);
17      if (success) entry → prev_ownership := tx; // Record that we acquired it
        return success;
19  }
    void release_orec(wstm_transaction *tx, wstm_entry *entry) {
21      bool succeeded := (tx → status = SUCCESSFUL);
        orec *orec_ptr := addr_to_orec(entry → addr);
23      orec seen := *orec_ptr;
        orec want;
25      if (entry → prev_ownership ≠ tx) return; // Never acquired by tx
        do {
27          if (seen.count > 1) {
                want := <(seen.count - 1), seen.owner>;
29          } else { // seen.count = 1
              if (seen.owner = tx) {
31              want := <0, (succeeded ? entry → new_version : entry → old_version)>
              } else {
33              bool os := (seen.owner → status = SUCCESSFUL);
                for (wstm_entry *oe in filter_for_orec(seen.owner → entries, orec_ptr))
35                  *(entry → addr) := (os ? oe → new_value : oe → old_value);
                want := <(0, (os ? oe → new_version : oe → old_version)>
37            }
            }
39      } while (CAS(orec_ptr, seen, want) ≠ seen);
    }
```

Fig. 20. Obstruction-free implementation of the acquire_orec and release_orec helper functions for WSTMCommitTransaction.

check in line 24). Otherwise, if there is no such entry, a new entry is added to tx containing the logical contents of the address (lines 26–29).

Figure 20 shows how acquire_orec and release_orec are modified to enable stealing. A third case is added to acquire_orec: Lines 13–17 steal ownership, so long as the contents of the orec have not changed since the victim's entries were merged into tx in prepare_descriptor (i.e., the orec's current owner matches the prev_ownership value recorded during prepare_descriptor). Note that we reuse the prev_ownership field in acquire_orec to indicate which entries led to a successful acquisition; this is needed in release_orec to determine which entries to release. release_orec itself is now split into three cases. Firstly, if the count field will remain above 0, the count is simply decremented because other threads may still be able to perform delayed writes to locations controlled by the orec (lines 27–28). Secondly, if the count is to return to 0 and our descriptor tx is still the owner, we take the old or new version number, as appropriate (lines 30–31).

The third case is that the count is to return to 0, but ownership has been stolen from our descriptor (lines 33–36). In this case we must reperform the updates from the current owner before releasing the orec (lines 34–36). This ensures that the current logical contents are written back to the locations, overwriting any delayed writes from threads that released the orec earlier. Note that we do not need to call force_decision before reading from the current owner: The count is 1, meaning that the thread committing tx is the only one working on that orec and so the descriptor referred by the orec must have already been decided (otherwise there would be at least two threads working on the orec).

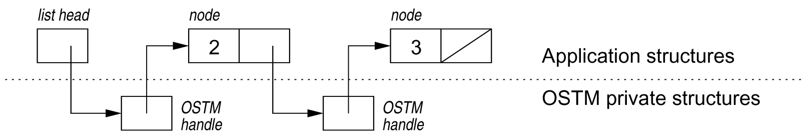
6.5 Discussion

The obstruction-free design from Figures 19 and 20 is clearly extremely complicated. Aside from its complexity, the design has an undesirable property under high contention: If a thread is preempted between calling acquire_orec and release_orec, then the logical contents of locations associated with that orec cannot revert to being held in the application heap until the thread is rescheduled.

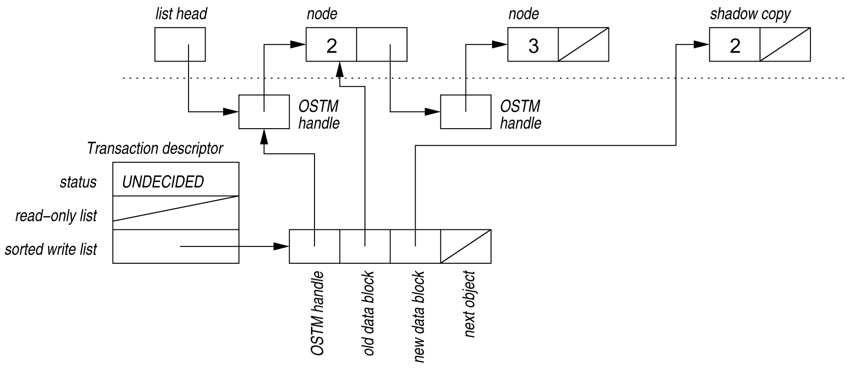
Although we do not present them here in detail, there are a number of extensions to the WSTM interface which add to the range of settings in which it can be used. In particular, we make note of a WSTMDiscardUpdate operation which takes an address and acts as a hint that the WSTM implementation is permitted (but not required) to discard any updates made to that address in the current transaction. This can simplify the implementation of some data structures in which shared "write-only" locations exist. For instance, in the red-black trees we use in Section 8, the implementation of rotations within the trees is simplified if references to dummy nodes are used in place of NULL pointers. If a single dummy node is used then updates to its parent pointer produce contention between logically nonconflicting transactions: In this case we can either use separate nodes or use WSTMDiscardUpdate on the dummy node's parent pointer.

7. OBJECT-BASED SOFTWARE TRANSACTIONAL MEMORY

We now turn to the third of our APIs: OSTM. Following previous transactional memory designs [Moir 1997], and concurrently with Herlihy et al.'s work [2003b], our design organizes memory locations into objects which act as the unit of concurrency and update. Rather than containing pointers, data structures contain opaque references to OSTM handles which may be converted to directly-usable machine pointers by opening them as part of a transaction. As a transaction runs, the OSTM implementation maintains sets holding the handles that it has accessed. When the transaction attempts to commit, these handles are validated to ensure that they represent a consistent snapshot of the objects' contents, and any updates to the objects are made visible to other threads.



(a) Example OSTM-based linked-list structure used by pseudocode in Figure 4. List nodes are chained together via OSTM handles which are private to the STM. References to OSTM handles 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 above. The transaction has accessed one object (node 2) which it has opened for writing. The read-only list is therefore empty, while the sorted write list contains one entry describing the modified node 2.

Fig. 21. Memory formats used in the OSTM implementation.

This means that OSTM offers a very different programming model than WSTM because OSTM requires data structures to be reorganized to use handles. Introducing handles allows the STM to more directly associate its coordination information with parts of the data structure, rather than using hash functions within the STM to maintain this association. This may be an important consideration in some settings: There is no risk of false contention due to hash collisions. Also note that while WSTM supports obstruction-free transactions, OSTM guarantees lock-free progress.

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.

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 reused and so a pointer uniquely identifies a particular use of a specific block of memory. Outside of a transaction context, shared references to an OSTM object point to a word-sized OSTM handle. Figure 21(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 a per-transaction descriptor which indicates the current status of the transaction and lists of objects that have been opened in read-only and read-write modes. To guarantee system progress when faced with contending transactions, the latter list must be maintained in sorted order: Hence we call it the *sorted write* list. Each list entry holds pointers to an OSTM handle and a data block. In addition, entries in the sorted write list hold a pointer to a thread-local shadow copy of the data block. Figure 21(b) illustrates the use of transaction descriptors and OSTM handles by showing a transaction in the process of deleting a node from an ordered linked list.

Ordinarily, OSTM handles 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*. If the transaction in Figure 21(b) were attempting to commit, then the OSTM handle for node 2 will be updated to contain a pointer to the transaction's descriptor. Concurrent reads can still determine the object's current value by searching the sorted write list and returning the appropriate data block, depending on the transaction's status. Once again notice that unlike MCAS, a commit is coordinated without needing to use reserved values in the application's data structures, and so full word-size values can be held in OSTM objects.

As usual, a predicate IsOSTMDesc distinguishes between references to a data block and those to a transaction descriptor.

7.2 Logical Contents

As with MCAS and WSTM, we proceed by defining the logical contents of an object. However, the definition is more straightforward than with WSTM because we avoid the aliasing problems attributable to the many-to-one relationship between orecs and heap words.

There are two cases which can be distinguished by applying the predicate IsOSTMDesc to an OSTM handle:

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

LS2 : If the OSTM handle 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 time 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 handle of each object opened in read-write mode is acquired in some global total order (e.g., arithmetic ordering of OSTM-handle pointers) by using CAS to replace the data-block pointer with a pointer to the transaction descriptor.

—*Read-check phase.* The handle 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.

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

7.4 Pseudocode

Figures 22 and 23 present pseudocode for the OSTMOpenForReading, OSTMOpenForWriting, and OSTMCommitTransaction operations. Both OSTMOpen operations use obj_read to find the most recent data block for a given OSTM handle; we therefore describe this helper function first. Its structure follows the definitions from Section 7.2. In most circumstances, the logical contents are defined by LS1: The latest data-block reference can be returned directly from the OSTM handle (lines 6 and 17). However, if the object is currently owned by a committing transaction then the correct reference is found by searching the owner's sorted write list (line 9) and selecting the old or new reference based on the owner's current status (line 15). As usual, LS2 is defined only for UNDECIDED, FAILED, and SUCCESSFUL descriptors and so if the owner is in its read phase, then the owner 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).

OSTMOpenForReading proceeds by checking whether the object is already open by that transaction (lines 20–23). If not, a new list entry is created and inserted into the read-only list (lines 24–28).

OSTMOpenForWriting proceeds by checking whether the object is already open by that transaction; if so, the existing shadow copy is returned (lines 32–33). If the object is present on the read-only list then the matching entry is removed (line 35). If the object is present on neither list then a new entry is allocated and initialized (lines 37–38). A shadow copy of the data block is made (line 40) and the list entry inserted into the sorted write list (line 41).

OSTMCommitTransaction itself is divided into three phases. The first phase attempts to acquire each object in the sorted write list (lines 4–9). If a more recent data-block reference is found then the transaction is failed (line 7). If the object is owned by another transaction then the obstruction is helped to completion (line 8). The second phase checks that each object in the read-only list has not been updated subsequent to being opened (lines 11–12). If all objects were successfully acquired or checked then the transaction will attempt to commit successfully (lines 15–16). Finally, each acquired object is released (lines 17–18)

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

t *obj_read (ostm_transaction *tx, ostm_handle<t*> *o) {
    t *data := o→data;
    if ( IsOSTMDesc(data) ) {
        ostm_transaction *other := (ostm_transaction *)data;
        obj_entry<t*> *hnd := search(other→sorted_write_list, o);
        if ( other→status = READ_CHECK )
            if ( (tx→status ≠ READ_CHECK) ∨ (tx > other) )
                OSTMCommitTransaction(other); // Help other
            else
                CAS(&other→status, READ_CHECK, FAILED); // Abort other
        data := (other→status = SUCCESSFUL) ? hnd→new : hnd→old;
    }
    return data;
}

t *OSTMOpenForReading (ostm_transaction *tx, ostm_handle<t*> *o) {
    obj_entry<t*> *hnd := search(tx→read_list, o);
    if ( hnd ≠ NULL ) return hnd→new;
    hnd := search(tx→sorted_write_list, o);
    if ( hnd ≠ NULL ) return hnd→new;
    hnd := new obj_entry<t*>();
    hnd→obj := o;
    hnd→old := obj_read(tx, o);
    hnd→new := hnd→old;
    insert(tx→read_list, hnd);
    return hnd→new;
}

t *OSTMOpenForWriting (ostm_transaction *tx, ostm_handle<t*> *o) {
    obj_entry<t*> *hnd := search(tx→sorted_write_list, o);
    if ( hnd ≠ NULL ) return hnd→new;
    if ( (hnd := search(tx→read_list, o)) ≠ NULL ) {
        remove(tx→read_list, o); // Upgrading to write
    } else {
        hnd := new obj_entry<t*>();
        (hnd→obj, hnd→old) := (o, obj_read(tx, o));
    }
    hnd→new := clone(hnd→old);
    insert(tx→sorted_write_list, hnd);
    return hnd→new;
}
```

Fig. 22. OSTM's OSTMOpenForReading and OSTMOpenForWriting interface calls. Algorithms for read and sorted write lists are not given here. Instead, search, insert, and remove operations are assumed to exist, for example, acting on linked lists of obj_entry structures.

and data-block reference returned to its previous value if the transaction failed, otherwise it is updated to its new value.

7.5 Discussion

Our lock-free OSTM was developed concurrently with an obstruction-free design by Herlihy et al. [2003b]. We include both in our experimental evaluation.

```
bool OSTMCommitTransaction (ostm_transaction *tx) {
    word data, status, desired_status := FAILED;
    obj_entry *hnd;
    for ( hnd in tx→sorted_write_list ) /* Acquire phase */
        while ( (data := CAS(&hnd→obj→data, hnd→old, tx)) ≠ hnd→old ) {
            if ( data = tx ) break;
            if ( ¬IsOSTMDesc(data) ) goto decision_point;
            commit_transaction((ostm_transaction *)data);
        }
    CAS(&tx→status, UNDECIDED, READ_CHECK);
    for ( hnd in tx→read_list ) /* Read phase */
        if ( (data := obj_read(tx, hnd→obj)) ≠ hnd→old ) goto decision_point;
    desired_status := SUCCESSFUL;
decision_point:
    while ( ((status := tx→status) ≠ FAILED) ∧ (status ≠ SUCCESSFUL) )
        CAS(&tx→status, status, desired_status);
    for ( hnd in tx→sorted_write_list ) /* Release phase */
        CAS(&hnd→obj→data, tx, (status = SUCCESSFUL) ? hnd→new : hnd→old);
    return (status = SUCCESSFUL);
}
```

Fig. 23. OSTM's OSTMCommitTransaction interface calls.

The two designs are similar in their use of handles as a point of indirection and the use of transaction descriptors to publish the updates that a transaction proposes to make.

The key difference lies in how transactions proceed before they attempt to commit. In our scheme, transactions operate entirely in private and so descriptors are only revealed when a transaction is ready to commit. In Herlihy et al.'s DSTM design, their equivalent to our OSTMOpen operation causes the transaction to acquire the object in question. This allows a wider range of contention management strategies because contention is detected earlier than with our scheme. Conversely, it appears difficult to make DSTM transactions lock-free using the same technique as our design: In our scheme, threads can help one another's commit operations, but in their scheme it appears it would be necessary for threads to help one another's entire transactions if one thread encounters an object opened by another.

It is interesting to note that unlike MCAS, there is no clear way to simplify our OSTM implementation by moving from lock freedom to obstruction freedom. This is because the data pointers in OSTM handles serve to uniquely identify a given object-state and so lock-freedom can be obtained without need for CCAS so as to avoid A-B-A problems when acquiring ownership.

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 pseudocode 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 three particular implementation problems: supporting nested transactions for composition (Section 8.1.1), distinguishing descriptors from application data (Section 8.1.2), and managing the memory within which descriptors are contained (Section 8.1.3).

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 enclosing transactions be aborted.

8.1.2 *Descriptor Identification.* To allow implementation of the *IsMCASDesc*, *IsCCASDesc*, *IsWSTMDesc*, and *IsOSTMDesc* predicates from Sections 5–7, there must 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 so as 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 always only 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 which are under the implementation's control: An implementation could use even values to indicate descriptor pointers and odd values to indicate version numbers.

An implementation using reserved bits would need to extend our pseudocode to perform bit-setting and bit-masking operations when manipulating descriptor references. Of course, other implementation schemes are possible, for instance, using runtime type information or placing descriptors in distinct memory pools.

8.1.3 *Reclamation of Dynamically-Allocated Memory.* We face two separate memory management problems: how to manage the memory within which descriptors are held and the storage within which application data structures are held. The latter problem has been subject to extensive recent work, such as SMR [Michael 2002] and pass-the-buck [Herlihy et al. 2005]. Both these schemes require mutator threads to publish their private references. Our implementation avoids this small mutator overhead by a scheme similar to that of Kung and Lehman [1980], in which tentative deallocations are queued until all threads pass through a quiescent state, after which it is known that they hold no private references to defunct objects.[6]

This 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 the A-B-A problems that would otherwise be caused by reuse. We use Michael and Scott's reference-counting garbage collection method [1995], 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.

We manage CCAS descriptors by embedding a fixed number of them within each MCAS descriptor. This avoids the overheads of memory management headers and reference counts that would otherwise be associated with the small CCAS descriptors. Since the logical contents of a CCAS descriptor is computable from the contents of the containing MCAS descriptor, our CCAS descriptors contain nothing more than a back pointer to the MCAS descriptor. Apart from saving memory, the fact that a CCAS descriptor contains no data specific to a particular CCAS suboperation allows it to be safely reused by the thread to which it was originally allocated. Since we know the number of threads participating in our experiments, embedding a CCAS descriptor per thread within each MCAS descriptor is sufficient to avoid any need to fall back to dynamic allocation. In situations where the number of threads is unbounded, dynamically-allocated descriptors can be managed by the same reference-counting mechanism as MCAS descriptors. However, unless contention is very high, it is unlikely that recursive helping will occur often, and so the average number of threads participating in a single MCAS operation will not exhaust the embedded supply of descriptors.

A similar 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 list.

8.2 Performance Evaluation

We evaluate the performance of our implementations of the three APIs 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 CPUs, each running at 1.2GHz. The server comprises 18 CPU/memory boards, each of which contains four CPUs and several gigabytes of memory. The boards are plugged into a backplane that permits communication via a high-speed crossbar interconnect. An addition 34 CPUs reside on 17 smaller CPU-only boards.

[6]As we explain in Section 8.2, this article considers workloads with at least one CPU available for each thread. Schemes like SMR or pass-the-buck would be necessary for prompt memory reuse in workloads with more threads than processors.

The set data type that we implement for each benchmark supports lookup, add, and remove operations, all of which act on a set and a key. *lookup* returns a Boolean indicating whether the key is a member of the set; *add* and *remove* update membership of the set in the obvious manner.

Each experiment is specified by three adjustable parameters:

S	—	the search structure that is being tested;
P	—	the number of parallel threads accessing the set; and
K	—	the initial (and mean) number of key values in the set.

The benchmark program begins by creating P threads and an initial set, implemented by S, containing the keys $0, 2, 4, \ldots, 2(K-1)$. All threads then enter a tight loop which they execute for five wallclock seconds. On each iteration they randomly select whether to execute a lookup ($p = 75\%$), add ($p = 12.5\%$), or remove ($p = 12.5\%$) on a random key chosen uniformly from the range $0 \ldots 2(K-1)$. This distribution of operations 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]. Furthermore, by setting equal probabilities for add and remove in a key space of size $2K$, we ensure that K is the mean number of keys in the set throughout the experiment. When five seconds have elapsed, each thread records its total number of completed operations. These totals are summed to get a system-wide total. The *result* of the experiment is the system-wide amount of CPU time used during the experiment, divided by the system-wide count of completed operations. When plotting this against P, a *scalable* design is indicated by a line parallel with the x-axis (showing that adding extra threads does not make each operation require more CPU time), whereas *faster* designs are indicated by lines closer to the x-axis (showing that less CPU time is required for each completed operation).

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

For brevity, our experiments only consider nonmultiprogrammed workloads where there are sufficient CPUs for running the threads in parallel. The main reason for this setting is that it enables a fairer comparison between our nonblocking designs and those based on locks. If we did use more threads than available CPUs, then when testing lock-based designs, a thread could be preempted at a point where it holds locks, potentially obstructing the progress of the thread scheduled in its place. Conversely, when testing nonblocking designs, even obstruction freedom precludes preempted threads from obstructing others: In all our designs, the time taken to remove or help an obstructing thread is vastly less than that of a typical scheduling quantum. Comparisons of multiprogrammed workloads would consequently be highly dependent on how well the lock implementation is integrated with the scheduler.

In addition to gathering the performance figures presented here, our benchmark harness can run in a testing mode, logging the inputs and results, as well as the invocation and response timestamps for each operation. Although logging every operation incurs a significant performance penalty, this mode of operation would never be enabled in normal use. We used an offline tool to check that these particular 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, complementing other techniques (such as proofs and model checking) which ensure algorithmic correctness.

We compare 14 different set implementations: 6 based on red-black trees and 8 on skip lists. Many of these are lock-based and, in the absence of published designs, 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 these contenders in more detail as part of his Ph.D. dissertation [Fraser 2003]. Fraser also considers general binary search trees and develops a range of nonblocking and lock-based designs.

As well as our own STM designs, we also include Herlihy's DSTM coupled with a simple "polite" contention manager that uses exponential backoff to deal with conflicts [Herlihy et al. 2003b]. We include this primarily because it is a configuration widely studied in the literature and so serves as a good common point for comparison between different designs. As others have shown, DSTM results are sensitive to contention management strategies and so our results should not be seen as a thorough comparison between DSTM and other STMs.

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 CPUs 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 multireader locks are required, we use another queue-based design by the same authors which allows adjacently-queued readers to 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 conflicts, 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 initially updated and continuously held until after the final update to the node. Although this slightly increases the possibility of conflict between threads, 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. This means that great care is needed to ensure that updates occur atomically and consistently. Fraser [2003] and Sundell et al. [2004] both provide pseudocode algorithms for nonblocking versions of the usual skip-list operations. In outline, list membership is defined according to presence in the lowest level. Insertion or deletion is performed on each level in turn as an independent linked list, using Harris's marking technique [2001] to logically delete a node from each level of the skip list. This implementation is used to show the performance gains that are possible using an intricate nonblocking system when compared with one built from MCAS, WSTM, or OSTM. Of course, the STM-based implementations do allow composability, whereas the CAS-based design does not.

(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) and (6) *Skip lists built using WSTM.* Skip lists can be built straightforwardly from single-threaded code using WSTM. We consider two variants: the obstruction-free WSTM built using double-word-width compare-and-swap (Section 6.4), and a version using operating system support [Harris and Fraser 2005].

(7) and (8) *Skip lists built using OSTM.* Skip lists can be built straightforwardly using 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 [2003b]. We couple the obstruction-free STM with a "polite" contention manager which introduces exponential backoff to deal with conflicts [Scherer III and Scott 2005].

(9) *Red-black trees with serialized writers.* Unlike skip lists, there has been little practical work on parallelism in balanced trees. Our first design [Fraser 2003] builds on Hanke's [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 mutual-exclusion lock is used to serialize 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]. This 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) and (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 sentinel nodes introducing contention when making needless updates to the sentinel's parent pointer.

(13) and (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. We consider implementations using OSTM from Section 7 and Herlihy et al.'s obstruction-free STM [2003b] coupled with the "polite" contention manager. As before, we use write discard on the sentinel node.[7]

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 nonblocking systems when running on machines with few CPUs, 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.

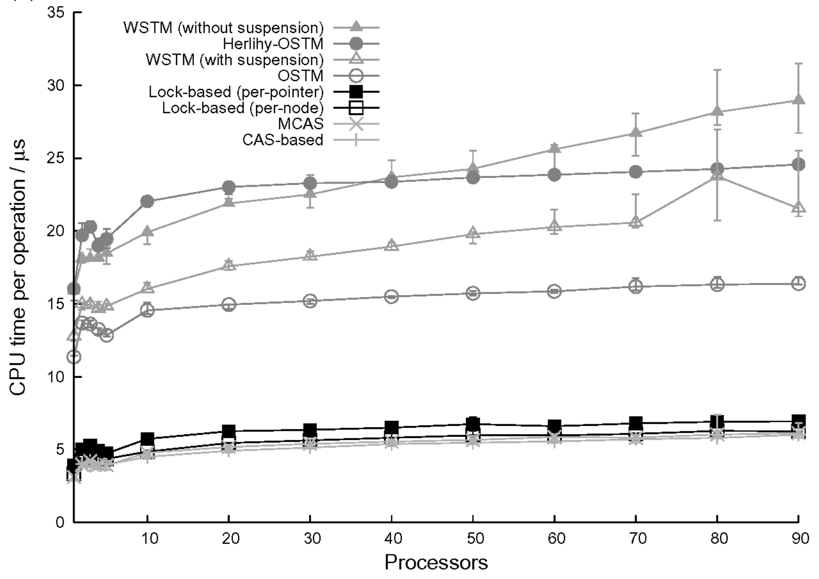
8.2.1 *Scalability Under Low Contention.* The first set of results measures performance when contention between concurrent operations is very low. Each experiment runs with a mean of 2^{19} keys in the set, which is sufficient to ensure that parallel writers are extremely unlikely to update overlapping sections of the data structure. A well-designed algorithm which provides disjoint-access parallelism will avoid introducing contention between these logically nonconflicting operations.

Note that all graphs in this section show a significant drop in performance when parallelism increases beyond five to ten threads. This is due to the architecture of the underlying hardware: Small benchmark runs execute within one or two CPU "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 interquad backplane.

Figure 24 shows the performance of each skip-list implementation. As expected, STM-based implementations perform poorly compared with other lock-free schemes; this demonstrates that there are significant overheads associated with the read and write operations (in WSTM), with maintaining the lists of opened objects and constructing shadow copies of updated objects (in OSTM), or

[7]Herlihy et al.'s DSTM cannot readily support write discard because only one thread may have a DSTM 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 *nontransactional semantics*: Such objects can be opened for writing by multiple threads, but the shadow copies remain thread private and are discarded on commit or abort.

(a)



(b)

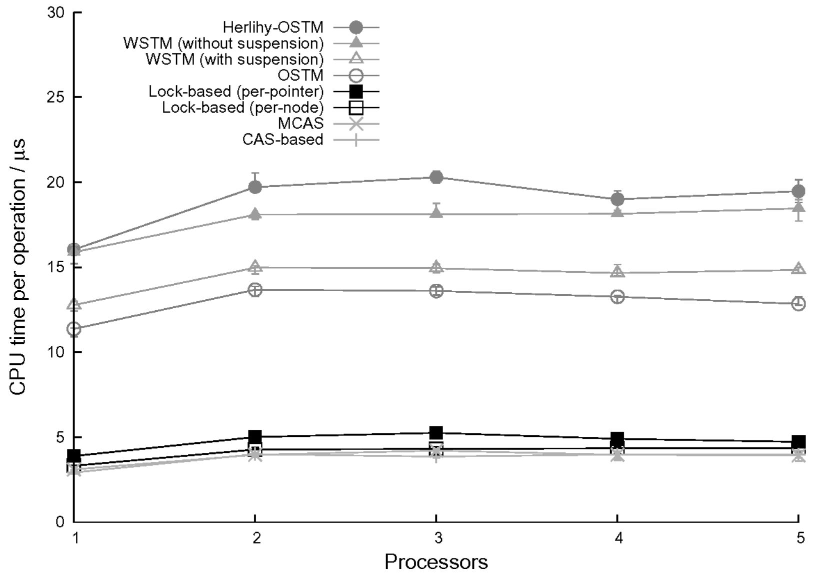


Fig. 24. Graph (a) shows the performance of large skip lists ($K = 2^{19}$) as parallelism is increased to 90 threads. Graph (b) is a "zoom" of (a), showing the performance of up to five threads. As with all our graphs, lines marked with boxes represent lock-based implementations, circles are OSTMs, triangles are WSTMs, and crosses are implementations built from MCAS or directly from CAS. The ordering in the key reflects that of the lines at the right-hand side of the graph: Lower lines are achieved by faster implementations.

with commit-time checks that verify whether the values read by an optimistic algorithm represent a consistent snapshot.

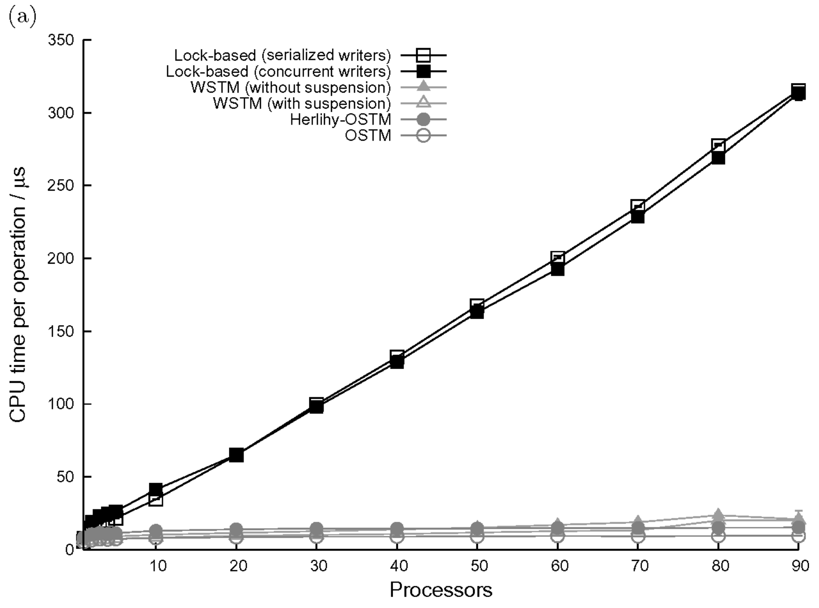
Lock-free CAS-based and MCAS-based designs perform extremely well because, unlike the STMs, they add only minor overheads on each memory access. Interestingly, under low contention, the MCAS-based design has almost identical performance to the much more complicated CAS-based design, thus indicating that the extra complexity of directly using hardware primitives 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 a larger number of locks.

Figure 25, presenting results for red-black trees, gives the clearest indication of the kinds of setting where our different techniques are effective. Neither of the lock-based schemes scales effectively with increasing parallelism; indeed, both OSTM- and WSTM-based trees outperform the schemes using locking with only two concurrent threads. 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 a skip list implemented over the same STM.

Surprisingly, the lock-based scheme that permits parallel updates performs hardly any better than the much simpler and more conservative design with serialised updates. 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. We suspect that, under low contention, OSTM is faster then Herlihy's design due to better cache locality. Herlihy's STM requires a double indirection when opening a transactional object: thus three cache lines are accessed when reading a field within a previously-unopened object. In contrast our scheme accesses two cache lines; more levels of the tree fit inside each CPU's caches and, when traversing levels that do not 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. This is a particular concern for nonblocking 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 shared data: That approach introduces unnecessary overheads when contention is low because fine-grained locking



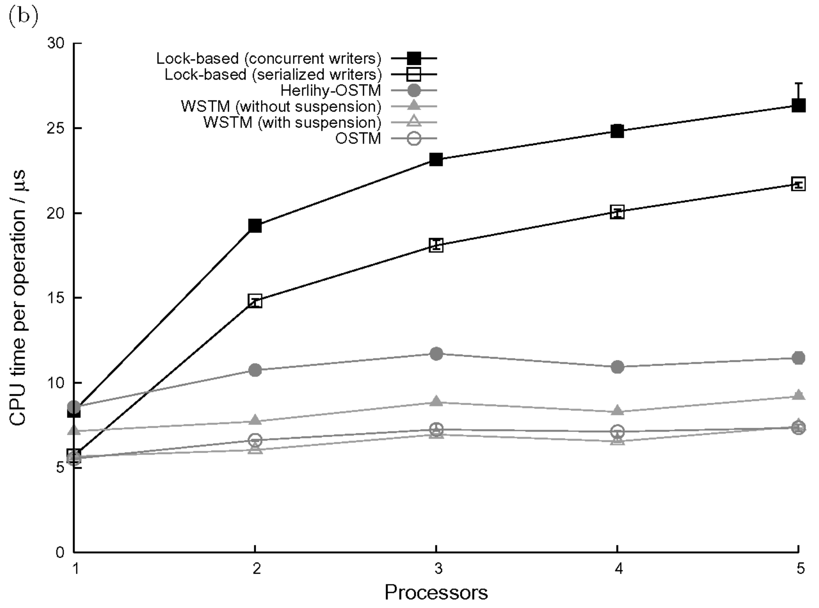


Fig. 25. Graph (a) shows the performance of large red-black trees ($K = 2^{19}$) as parallelism is increased to 90 threads. Graph (b) is a "zoom" of (a), showing the performance of up to five threads.

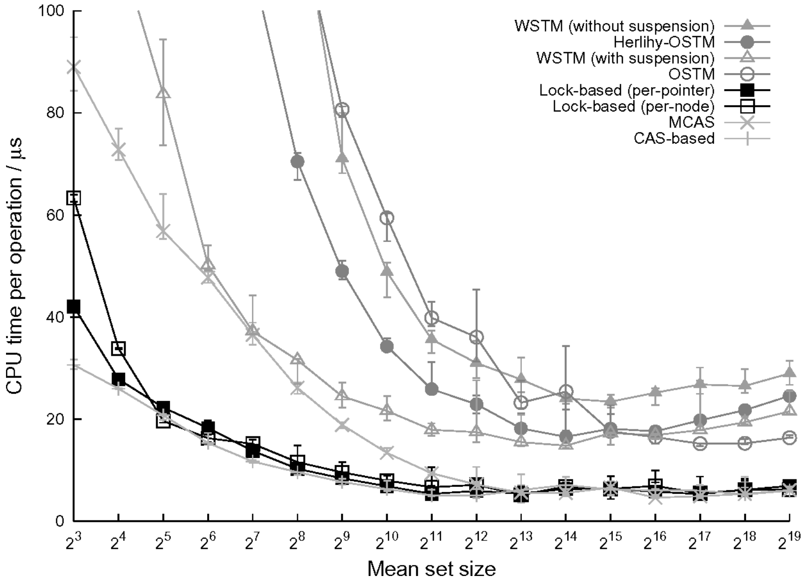


Fig. 26. Effect of contention on concurrent skip lists ($P = 90$).

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 threads ($P = 90$). Contention is varied by adjusting the benchmark parameter K and hence the mean size of the data structure under test. Although not a general-purpose contention metric, this is sufficient to allow us to compare the performance of a single data structure implemented over different concurrency-control mechanisms.

Figure 26 shows the effect of contention on each of the skip-list implementations. It indicates that there is sometimes a price for using MCAS, rather than programming directly using CAS. The comparatively poor performance of MCAS when contention is high is because many operations must retry several times before succeeding: 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. Of course, we could postulate a hybrid implementation in which the programmer strives to perform updates using multiple smaller MCAS operations. This could provide an attractive middle ground between the complexity of using CAS and the "one-fails-then-all-fail" contention problems of large MCAS operations.

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 introduce unnecessary overheads compared with per-node locks under moderate to low contention.

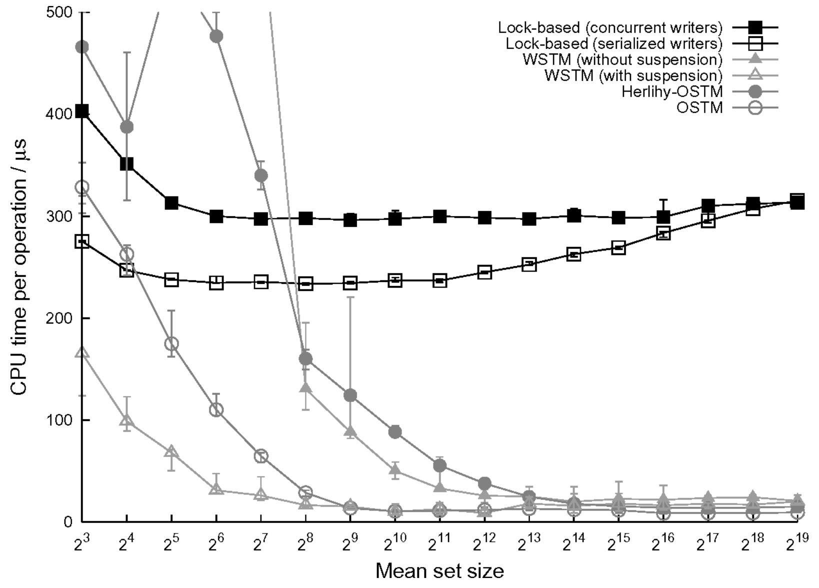


Fig. 27. Effect of contention on concurrent red-black trees ($P = 90$).

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.

Figure 27 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 multireader lock. For this workload, the OSTM and WSTM schemes using suspension perform well in all cases, although conflicts still significantly affect performance.

Herlihy's STM performs comparatively poorly under high contention when using an initial contention-handling mechanism that introduces exponential backoff to "politely" deal with conflicts. Other schemes may work better [Scherer III and Scott 2005] and, since DSTM is obstruction-free, would be expected to influence its performance more than OSTMs. Once again, the results here are intended primarily to compare our designs with lock-based alternatives and we include DSTM in this sample configuration because it has been widely studied in the literature. Marathe et al. perform further comparisons of DSTM and OSTM [2004].

Note that when using this 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 sensitivity to the choice of backoff 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

We have presented three APIs for building nonblocking concurrency-safe software and demonstrated that these can match or surpass the performance of state-of-the-art lock-based alternatives. Thus, not only do nonblocking systems 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 without the runtime overheads that have traditionally been feared.

Furthermore, APIs such as STM have benefits in ease of use compared with traditional direct use of mutual-exclusion locks. An STM avoids the need to consider issues such as granularity of locking, the order in which locks should be acquired to avoid deadlock, and composability of different data structures or subsystems. This ease of use is in contrast to traditional implementations of nonblocking data structures based directly on hardware primitives such as CAS.

In conclusion, using the APIs that we have presented in this article, it is now practical to deploy lock-free techniques, with all their attendant advantages, in situations where lock-based synchronization would traditionally be the only viable option.

9.1 Future Work

The work discussed in this article leads to many directions for future exploration. As we briefly discussed in Section 2.4, we have already made progress with some of these, most notably integrating transactional memories into managed runtime environments and investigating the programming language abstractions that can be provided as a consequence [Harris and Fraser 2003; Harris 2004; Welc et al. 2004; Ringenburg and Grossman 2005; Harris et al. 2005].

Another direction relates to the direct use of our APIs by expert programmers. In particular, in the case of OSTM, it is necessary for the programmer to be careful to ensure that the APIs do not write to objects that they have only opened for reading. We do not currently check that programmers follow this rule, but we could imagine tools to help with this. For instance, in a debugging mode, OSTM could ensure that only the data blocks of objects that a transaction has opened for writing are held on pages with write-permission enabled. This could be achieved by using a fresh page or pages for the data blocks of each transaction; this space cost, along with the time needed for system calls to modify page permissions, means that it is not suitable for production use.

A further avenue of exploration is whether there are new primitives that future processors might provide to improve the performance of nonblocking algorithms such as our MCAS, WSTM, and OSTM implementations. It is clear that mutliword atomic updates, such as DCAS, can simplifiy the complex algorithms we have designed; it is less clear that such primitives can actually enhance performance.

A final direction is the integration of software transactional memory with hardware support of the kind originally conceived by Herlihy and Moss [1993] and recently subject to vigourous research [Rajwar and Goodman 2002; Hammond et al. 2004; Ananian et al. 2005; Moore et al. 2005; McDonald et al. 2005]. These hardware schemes are attractive for short-running transactions in which all of the accesses can be contained in whatever structures future hardware may provide. Conversely, software-based schemes are attractive for longer-running transactions when these limits are exceeded or when blocking operations are to be provided.

ACKNOWLEDGMENTS

We thank the anonymous referees for their particularly thorough reading of earlier drafts of this article.

SOURCE CODE

Source code for the MCAS, WSTM, and OSTM implementations described in this article is available at `http://www.cl.cam.ac.uk/netos/lock-free`. Also included are the benchmarking implementations of skip lists and red-black trees, and the offline linearizability checker described in Section 8.2.

REFERENCES

ANANIAN, C. S., ASANOVIĆ, K., KUSZMAUL, B. C., LEISERSON, C. E., AND LIE, S. 2005. Unbounded transactional memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA)*. (San Franscisco, CA). 316–327.

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)*. 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)*. 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.

DICE, D., SHALEV, O., AND SHAVIT, N. 2006. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*.

FICH, F., LUCHANGCO, V., MOIR, M., AND SHAVIT, N. 2005. Obstruction-Free algorithms can be practically wait-free. In *Distributed Algorithms*, P. Fraigniaud, Ed. Lecture Notes in Computer Science, vol. 3724. Springer Verlag, Berlin. 78–92.

FRASER, K. 2003. Practical lock freedom. Ph.D. thesis, Computer Laboratory, University of Cambridge. Also available as Tech. Rep. UCAM-CL-TR-639, Cambridge University.

GREENWALD, M. 1999. Non-Blocking synchronization and system design. Ph.D. thesis, Stanford University. Also available as Technical Rep. 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)*. 260–269.

HAMMOND, L., CARLSTROM, B. D., WONG, V., HERTZBERG, B., CHEN, M., KOZYRAKIS, C., AND OLUKOTUN, K. 2004. Programming with transactional coherence and consistency (TCC). In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM Press, New York. 1–13.

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, Berlin. 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, Berlin. 193–204.

Harris, T. 2001. A pragmatic implementation of non-blocking linked lists. In *Proceedings of the 15th International Symposium on Distributed Computing (DISC)*. Springer Verlag, Berlin. 300–314.

Harris, T. 2004. Exceptions and side-effects in atomic blocks. In *Proceedings of the PODC Workshop on Synchronization in Java Programs*. 46–53. Proceedings published as Memorial University of Newfoundland CS Tech. Rep. 2004-01.

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 and Applications (OOPSLA)*. 388–402.

Harris, T. and Fraser, K. 2005. Revocable locks for non-blocking programming. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM Press, New York. USA, 72–82.

Harris, T., Marlow, S., Peyton-Jones, S., and Herlihy, M. 2005. Composable memory transactions. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM Press, New York. 48–60.

Heller, S., Herlihy, M., Luchangco, V., Moir, M., Scherer, B., and Shavit, N. 2005. A lazy concurrent list-based set algorithm. In *9th International Conference on Principles of Distributed Systems (OPODIS)*.

Hennessy, J. L. and Patterson, D. A. 2003. *Computer Architecture—A Quantitative Approach*, 3rd ed. Morgan Kaufmann, San Francisco, CA.

Herlihy, M. 1993. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst. 15*, 5 (Nov.), 745–770.

Herlihy, M. 2005. SXM1.1: Software transactional memory package for C#. Tech. Rep., Brown University and Microsoft Research. May.

Herlihy, M., Luchangco, V., Martin, P., and Moir, M. 2005. Nonblocking memory management support for dynamic-sized data structures. *ACM Trans. Comput. Syst. 23*, 2, 146–196.

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, Los Alamitos, CA. 522–529.

Herlihy, M., Luchangco, V., Moir, M., and Scherer, W. 2003b. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC)*. 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)*. ACM Press, New York. 289–301.

Herlihy, M. and Wing, J. M. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (Jul.), 463–492.

Hoare, C. A. R. 1972. Towards a theory of parallel programming. In *Operating Systems Techniques*. A.P.I.C. Studies in Data Processing, vol. 9. Academic Press, 61–71.

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

Jayanti, P. and Petrovic, S. 2003. Efficient and practical constructions of LL/SC variables. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*. ACM Press, 285–294.

Jones, R. and Lins, R. 1996. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons.

Kung, H. T. and Lehman, P. L. 1980. Concurrent manipulation of binary search trees. *ACM Trans. Database Syst. 5*, 3 (Sept.), 354–382.

Kung, H. T. and Robinson, J. T. 1981. On optimistic methods for concurrency control. *ACM Trans. Database Syst. 6*, 2, 213–226.

Marathe, V. J., Scherer III, W. N., and Scott, M. L. 2004. Design tradeoffs in modern software transactional memory systems. In *Proceedings of the 7th Workshop on Languages, Compilers, and Run-Time Support for Scalable Systems*.

McDonald, A., Chung, J., Chafi, H., Cao Minh, C., Carlstrom, B. D., Hammond, L., Kozyrakis, C., and Olukotun, K. 2005. Characterization of TCC on chip-multiprocessors. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*.

Mellor-Crummey, J. and Scott, M. 1991a. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst. 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. 2002. 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)*.

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. December.

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

Moir, M. 2002. Personal communication.

Moore, K. E., Hill, M. D., and Wood, D. A. 2005. Thread-Level transactional memory. Tech. Rep.: CS-TR-2005-1524, Deptartment of Computer Sciences, University of Wisconsin, Motorola Inc., Phoenix, AZ. 1–11.

Motorola. 1985. *MC68020 32-Bit Microprocessor User's Manual*.

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

Rajwar, R. and Goodman, J. R. 2002. Transactional lock-free execution of lock-based programs. *ACM SIGPLAN Not. 37*, 10 (Oct.), 5–17.

Rajwar, R., Herlihy, M., and Lai, K. 2005. Virtualizing transactional memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*. IEEE Computer Society, Los Alamotos, CA. 494–505.

Riegel, T., Felber, P., and Fetzer, C. 2006. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*.

Ringenburg, M. F. and Grossman, D. 2005. AtomCaml: First-Class atomicity via rollback. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM Press, New York. 92–104.

Scherer III, W. N. and Scott, M. L. 2005. Advanced contention management for dynamic software transactional memory. In *Proceedings of the 24th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*. ACM Press, New York. 240–248.

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)*. 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)*. 204–213.

Sundell, H. and Tsigas, P. 2004. Scalable and lock-free concurrent dictionaries. In *Proceedings of the ACM Symposium on Applied Computing*. ACM Press, New York, NY. 1438–1445.

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.

Welc, A., Jagannathan, S., and Hosking, T. 2004. Transactional monitors for concurrent objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. 519–542.

Wing, J. M. and Gong, C. 1993. Testing and verifying concurrent objects. *J. Parallel Distrib. Comput. 17*, 1 (Jan.), 164–182.

Received July 2004; revised January 2007; accepted January 2007