A Qualitative Survey of Modern Software Transactional Memory Systems*

Virendra J. Marathe and Michael L. Scott

TR 839

Department of Computer Science
University of Rochester
Rochester, NY 14627-0226
{vmarathe, scott}@cs.rochester.edu

June 2004

Abstract

Software Transactional Memory (STM) can be defined as a generic nonblocking synchronization construct that allows correct sequential objects to be converted automatically into correct concurrent objects. In STM, a transaction is defined as a sequence of instructions that atomically modifies a set of concurrent objects. The original STM proposed by Shavit and Touitou worked on static transactions (wherein the concurrent objects being accessed by a transaction were pre-determined). Recent STM research has focused on support for more realistic dynamic transactions.

In this paper we present a qualitative survey of modern STM systems that support dynamic transactions. More concretely, we describe the designs of three STM systems—a hash table based STM system (Hash table STM) for shared memory words due to Harris and Fraser, and two purely object-based STM systems, one due to Herlihy et al., the other due to Fraser. We also present a detailed analysis of the Hash table STM and a qualitative comparison between the two object-based STM systems. We identify a scalability drawback (that may be unacceptable in some applications) in the Hash table STM and propose an LL/SC based variant that overcomes this drawback. The qualitative comparison between the two object-based STM systems helps us understand their various design peculiarities and the potential tradeoffs involved. Specifically, we discuss object ownership acquire semantics, levels of indirection to access concurrent objects, space utilization, transaction search overhead during conflict resolution, transaction validation semantics, and contention management versus helping.

*This work was supported in part by NSF grants numbers EIA-0080124, CCR-9988361, and CCR-0204344, by DARPA/AFRL contract number F29601-00-K-0182, and by financial and equipment grants from Sun Microsystems Laboratories.

1 Introduction

A *concurrent object* is a data object shared by multiple processes in a concurrent system. The classic lock-based synchronization algorithms for concurrent access to these objects suffer from several important drawbacks like deadlocks, priority inversion, convoying and lack of fault tolerance. Due to these drawbacks, the last two decades have seen an increasing interest in *nonblocking* synchronization algorithms. Software Transactional Memory [Shavit 95] is one such nonblocking synchronization construct.

1.1 Non-Blocking Synchronization Algorithms

In nonblocking synchronization algorithms, processes do not need to *wait* (e.g. spin) to gain access to a concurrent object during contention. Instead of waiting, a concurrent process may either abort its own atomic operation (retrying later optionally), or abort the atomic operation of the conflicting process. More formally, nonblocking synchronization algorithms permit asynchronous and concurrent access (including updates) to concurrent objects, but guarantee consistent updates using atomic operations like *Compare&Swap (CAS)* and *Load-Linked/Store-Conditional (LL/SC)* [Case 78, Jensen 87, Herlihy 90, Herlihy 91]. In contrast, blocking synchronization algorithms use mutually exclusive critical sections to serialize access to concurrent objects. Nonblocking synchronization algorithms have been classified into three main categories based on their algorithmic progress guarantees:

- *Wait-freedom* [Herlihy 91] – Wait-freedom is the strongest property of a nonblocking synchronization algorithm in terms of progress guarantees of concurrent processes. This property guarantees that *all* processes contending for a common set of concurrent objects make progress in a finite number of their individual time steps. The definition of the wait-free property rules out the occurrence of deadlocks as well as starvation.

- *Lock-freedom* – Lock-freedom is a weaker progress guarantee property of a nonblocking synchronization algorithm. It guarantees that given a set of concurrent processes contending for a set of concurrent objects, *at least one* process makes progress in a finite number of execution time steps of any other concurrent process. Lock-freedom rules out the occurrence of deadlocks but not starvation.

- *Obstruction-freedom* [Herlihy 03a] – In a concurrent system, a nonblocking synchronization algorithm is said to be obstruction free if it guarantees progress to a process in a finite number of its own steps in the absence of contention. This is the weakest progress guarantee property of a non-blocking synchronization algorithm. Obstruction-freedom rules out the occurrence of deadlocks, but *livelocks* may occur if a group of processes keep preempting or aborting each others atomic operations and consequently no one makes any progress.

Blocking synchronization algorithms guarantee consistency by enforcing mutually exclusive access to critical sections. Critical sections are guarded by locks that may be acquired by concurrent processes exclusively. Once a lock has been acquired by a process, other concurrent processes trying to acquire the same lock are forced into a *wait-state* until the lock is released by its owner process. They may spin, yield, or ask the scheduler to block them; they cease to make forward progress in any case. This wait-state is the fundamental cause of the various problems mentioned above in blocking synchronization algorithms. Nonblocking synchronization algorithms are free from this wait-state of concurrent processes. Obstruction-freedom introduces the livelock problem, but it

can be effectively minimized using simple techniques like exponential backoff, or higher throughput techniques of *contention management* [Herlihy 03b]. Herlihy et al. have proved that effective contention management is crucial to achieve high throughput for any obstruction-free nonblocking synchronization algorithm, in particular Software Transactional Memory systems [Shavit 95]. The problems of deadlock, priority inversion and convoying do not occur in nonblocking synchronization algorithms. Fault tolerance is also ensured using mechanisms like *helping* [Shavit 95] and *stealing* [Harris 03].

It may apparently seem that the strongest wait-freedom property is most desirable in any nonblocking synchronization algorithm. On the other hand, ensuring just obstruction-freedom leads to greater simplification and flexibility in the design of nonblocking synchronization algorithms, and subsequent performance benefits. Thus, for practical reasons, obstruction-free synchronization algorithms turn out to be faster than wait-free and lock-free alternatives.

Software Transactional Memory (STM) is a nonblocking synchronization construct that has been studied for over a decade. It has both obstruction-free [Herlihy 03b, Harris 03] and lock-free [Shavit 95, Fraser 03] implementations. This paper contains a brief survey of these Software Transactional Memory systems followed by more detailed analysis and comparison among the most recent STM systems [Fraser 03, Harris 03, Herlihy 03b]. In Section 2 we introduce the general idea of STM systems and briefly discuss the design of the first ever STM system proposed by Shavit and Touitou. We also point out the fundamental limitations of this algorithm. In Section 3 we discuss the hash table enabled *word-based* STM design of Harris and Fraser [Harris 03]. In word-based STMs each individual shared memory word is a concurrent object. We identify potential problems with this design and propose a variant design that addresses these problems. In Section 4 we overview the *object-based* STM systems of Fraser [Fraser 03] and of Herlihy et al. [Herlihy 03b]. We present a qualitative comparison between these two approaches in Section 5. Finally we conclude with a statement on future directions in our work.

2 Software Transactional Memory (STM)

Software Transactional Memory can be defined as a generic nonblocking synchronization construct that allows correct sequential objects to be converted automatically into correct concurrent objects. The original idea of Transactional Memory was proposed by Herlihy and Moss as a novel architectural support mechanism for nonblocking synchronization [Herlihy 93]. A similar mechanism was proposed concurrently by Stone et al. [Stone 93]. A *transaction* is defined as a finite sequence of instructions (satisfying the linearizability [Herlihy 90] and atomicity properties) that is used to access or modify concurrent objects. Herlihy and Moss [Herlihy 93] proposed the implementation of transactional memory by simple extensions to multiprocessor cache coherence protocols. Their transactional memory provides an instruction set for accessing shared memory locations by transactions.

Subsequently, Shavit and Touitou proposed a software equivalent of transactional memory, the *Software Transactional Memory* [Shavit 95]. Their system mechanism is as follows : A transaction makes updates to a concurrent object only after a system-wide declaration of its update intention. This declaration helps other transactions recognize that some transaction is about to make updates to a particular concurrent object. The declaring transaction is said to be the *owner* of the corresponding concurrent object. The declaration can be trivially done by storing a reference in the concurrent object to its current owner transaction. After making the intended update to the owned concurrent object, a transaction relinquishes its ownership. The processes of acquiring and releasing ownerships of concurrent objects can be done atomically (in a nonblocking fashion)

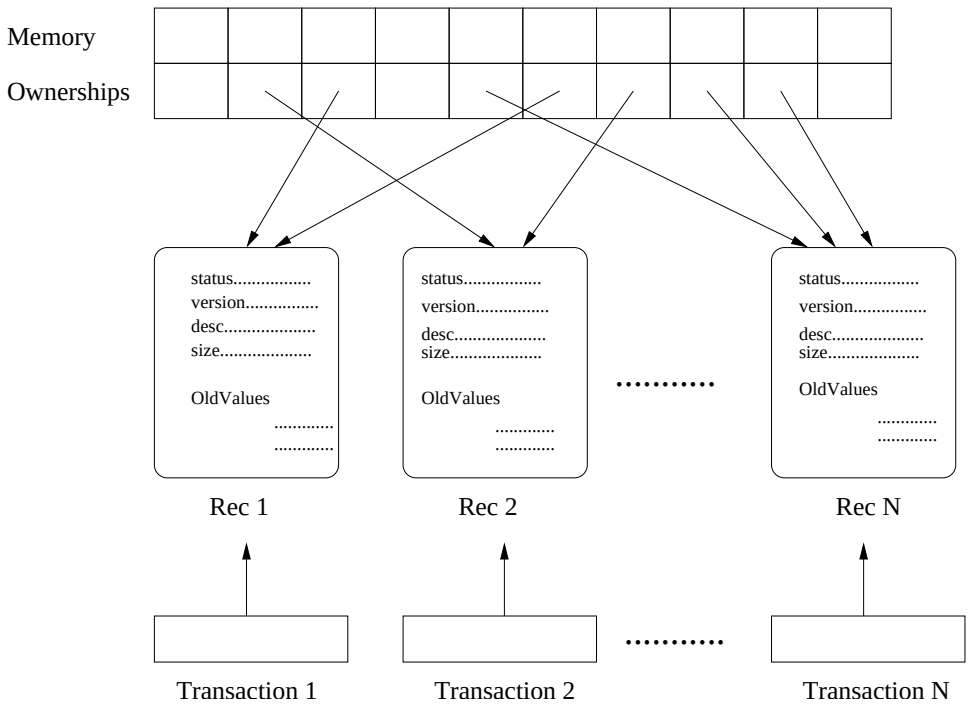

Figure 1: STM Implementation: shared data structures (courtesy [Shavit 95])

using operations like CAS and LL/SC. In order to ensure atomic updates to concurrent objects, transactions need exclusive ownerships of these objects. In their design, Shavit and Touitou used shared memory words as concurrent objects. At most one transaction may own a shared memory word at any given time. The ownership information of a shared memory word is stored in another corresponding word called the *ownership record*. Each memory word has a distinct associated ownership record. The ownership record either stores a null value (denoting that no transaction owns the corresponding shared memory word currently) or a reference to its owner's *transaction record*. A transaction record is a data structure that stores information about the corresponding transaction's software transactional memory accesses. A transaction owns a single transaction record at a time. All the transactions are given shared access to all the existing transaction records, but each transaction record is owned by only one transaction.

Figure 1 gives a clearer idea about the above description. If a transaction fails to acquire an ownership (because the memory location is owned by some other transaction at that time), it aborts and releases its already acquired ownerships. If the transaction succeeds in acquiring all the desired ownerships, it atomically changes its state to COMMITTED,[1] makes its updates and releases the acquired ownerships. This is a nonblocking multi-word CAS operation. To avoid livelocks, ownerships are acquired in some global total order of the shared memory word addresses. In order to ensure fault tolerance a *non-recursive helping*[2] mechanism is used forcing conflicting

[1]Conventionally, a transaction may be in three states : ACTIVE (when the transaction is still executing), ABORTED (when the transaction's atomic operation gets aborted), and COMMITTED (when the transaction commits its updates)

[2]In helping, when a transaction A detects a conflict with another transaction B, A makes B's updates on its behalf. This may lead to recursion and possibly livelocks. Non-recursive helping allows helping up to a specified level only. In the original STM design, the helping level was restricted to one.

transactions to help the owner of the ownership record under conflict. Total global ordering and helping ensure lock-freedom in this STM system. A single process executes one transaction at a time. Each process owns at most one transaction record at any given time.

This original proposal of STM has two significant limitations. The first limitation is that to ensure ordered access the system mandates pre-knowledge of all the memory words that a transaction accesses. This restricts on-the-fly decision making of memory words to be accessed by the transactions. More recent work [Fraser 03, Harris 03, Herlihy 03b] has focused on providing software transactional memory algorithms for concurrent objects dynamically (more about that later). The other limitation is that each shared memory location requires an associated ownership record. Even though the ownership record may be just one word in length, the memory requirement of this STM doubles. An efficiency drawback that has been pointed out [Harris 03] is the contention overhead during helping. Harris and Fraser propose a hash table based STM system [Harris 03] that increases the spanning range of ownership records phenomenally and uses *stealing* as an alternative to helping.

3 A Hash Table Based STM

Harris and Fraser have recently proposed a word-based STM [Harris 03] that uses a hash table for storing ownership records. Their system abstracts away the intricate details of their design by providing an interface to access the STM. This interface consists of the following functions:

```
void STMStart( )
stm_word STMRead(addr a)
void STMWrite(addr a, stm_word w)
void STMAbort( )
boolean STMCommit( )
boolean STMValidate( )
void STMWait( )
```

STMStart begins a new transaction. STMRead and STMWrite are used to read and write to the shared memory words. STMAbort aborts an ACTIVE transaction and STMCommit commits an ACTIVE transaction. STMValidate verifies the consistency of the current ACTIVE transaction. STMWait enforces *blocking* semantics during contention. Since we are concerned only with the nonblocking aspect of this system we shall not consider the STMWait semantics in our discussion.

3.1 The Hash Table STM Design

Figure 2 shows the schematic design of the Hash table STM system proposed by Harris and Fraser [Harris 03]. This STM system consists of three main data structures:

- *Application Heap.* This is the shared memory that holds the actual data the concurrent processes intend to use.
- *Hash table of ownership records (orecs).* The shared memory locations hash into the orecs hash table. Each orec stores either a version number (signifying that the orec is currently not owned by any transaction) or a reference to the transaction descriptor of the transaction that currently owns the orec. Consequently, when a transaction owns an orec, it semantically owns *all* the shared memory locations hashing into that orec.

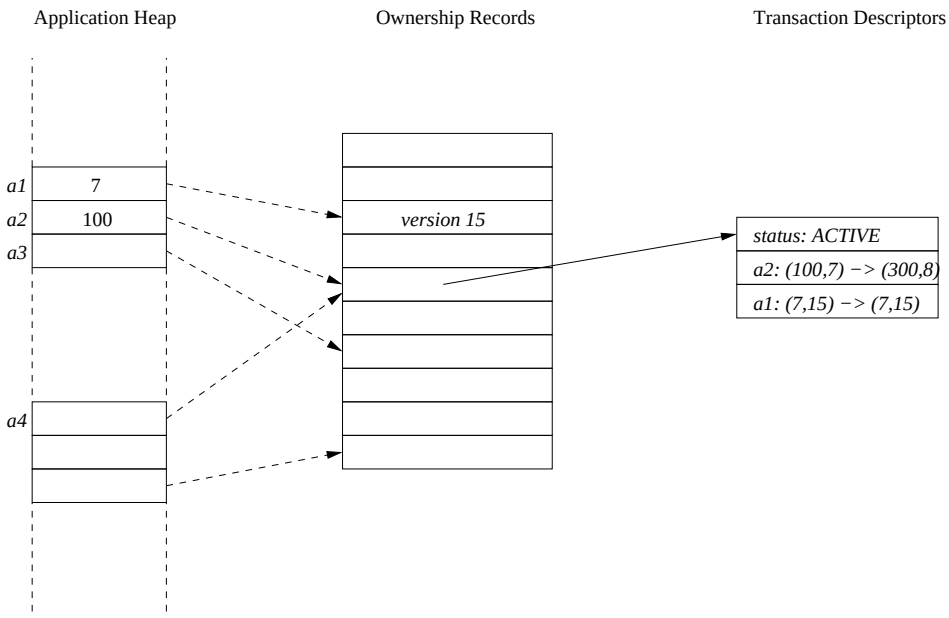

Figure 2: The STM Heap Structure showing an active transaction (courtesy [Harris 03])

- *Transaction Descriptors.* Each transaction uses its transaction descriptor to access the shared memory. The transaction descriptor consists of *transaction entries*, one for each shared memory location accessed by the transaction. Each transaction entry in turn consists of five elements: the address of the shared memory word, the contents in that memory word, the version number of the corresponding orec, the potential new value of the memory word (to be written back at commit time), and the potential new version number to be stored in the orec.

An STMRead or STMWrite creates a transaction entry (corresponding to the accessed memory location) if one does not already exist in the transaction descriptor. A process may execute just one transaction at a time; and each transaction is represented by one transaction descriptor. To maintain consistency, a transaction descriptor must be *well-formed*, i.e. it must either contain at most one entry corresponding to an orec, or all the entries corresponding to an orec must have the same old and new version numbers. It must be noted that a transaction does not try to acquire ownership of an orec during an STMRead or an STMWrite. Acquiring orecs is done only during execution of the STMCommit operation. As a result, the commit operation is a multi-word CAS. The multi-word CAS is highlighted here since it turns out to be an important distinction between STM systems. The DSTM [Herlihy 03b] for example does not require a multi-word CAS at commit time.

The most important and involved part of the Hash table STM system is the STMCommit operation. To commit, a transaction has to acquire exclusive ownership of all the orecs referred in its transaction descriptor. The transaction does so by using an atomic CAS operation on each orec. After acquiring all the ownerships, the transaction atomically sets its status to COMMITTED, and then goes into a *release phase*. In the release phase, the transaction writes back each transaction entry's new value and version number to the corresponding memory location and orec respectively.

A conflict occurs when a transaction finds another transaction's descriptor in the orec that it tries to read (during STMRead or STMWrite) or acquire (during STMCommit).

Read conflict. The conflicting transaction may be in the COMMITTED, ABORTED or ACTIVE state. If the conflicting transaction is ACTIVE the current transaction aborts it. This design decision makes this algorithm *obstruction-free* in nature. Now the conflicting transaction is either ABORTED or COMMITTED. The current transaction has to create a transaction entry for the shared memory word it intends to access. The orec version number is obtained from the conflicting transaction's descriptor. The valid memory word contents can be obtained either from the conflicting transaction's descriptor (if that transaction contains an entry for the same word) or the shared memory heap (if the conflicting transaction does not contain an entry for the same word).

Acquire conflict. The conflicting transaction may be in any of the three states during an acquire conflict. If the conflicting transaction is ACTIVE, the current transaction aborts it. Now the conflicting transaction is either ABORTED or COMMITTED. The current transaction now verifies the consistency of the version number of the orec under conflict with the latest valid version number of the same orec in the conflicting transaction's descriptor. If an inconsistency is detected, the current transaction aborts itself and releases all its previously acquired ownerships. Otherwise it has to ensure that the valid updates from the conflicting transaction's descriptor are made to the orec under conflict and the corresponding memory locations. It may do so by *helping* the transaction [Shavit 95]. But Harris and Fraser argue that helping is expensive since it causes a lot of contention. Instead of helping, they have proposed a *stealing* mechanism. In the stealing mechanism, the current transaction merges the transaction entries (from the conflicting transaction's descriptor) corresponding to the orec under conflict into its own transaction descriptor. After merging happens, the transaction sets the orec to point to its own transaction descriptor using an atomic CAS operation.

During its acquire phase, if the current transaction is ABORTED (by some other other transaction) it releases all the already acquired orecs. After a successful acquire phase the current transaction atomically updates its status to COMMITTED (using a CAS operation). After committing, the current transaction starts updating the memory locations corresponding to its transaction entries, and releasing the corresponding orecs using a CAS operation. Stale updates may occur during the release phase of a transaction. Stale updates happen when the stealer transaction makes updates to the shared memory before the victim transaction (another transaction from which the stealer has stolen an orec) does, and the victim transaction subsequently makes its *stale* updates to the shared memory potentially overwriting the most recent updates made by the stealer. Stale updates are avoided by a *redo* operation : When the current transaction discovers a potential threat of stale updates (on detecting an orec theft) it redoes the updates to the memory words corresponding to the stolen orec from the stealer transaction's descriptor only if the stealer is no longer ACTIVE. The combination of merging (during stealing) and redo (during releasing) operations guarantees that the most recent valid update is always made to the shared memory.

In the presence of moderate to heavy contention orecs will be moved around before their most recent consistent version numbers are restored. To ensure consistency in updates to orecs, a reference count is used for each orec. The orec is now a composite of a version number and a reference count. The reference count in an orec is incremented by a stealer during the steal operation and also by the first transaction that acquires it directly from the orec hash table. The reference count is decremented in the release phase. If the count goes down to zero the current transaction updates the orec with the new valid version number of that orec from its last stealer transaction's descriptor. A combined atomic update to the reference count and the version number is made using an atomic two-word wide CAS operation (also called CASX).

3.2 Design and Efficiency Issues

In Section 3.1 we described the Hash table STM system. The direct application of this STM system is on shared memory at the granularity of atomic word accesses. In some concurrent data structures, high granularity of parallel access may be required. A purely object-based solution (this will be clear in Section 4) would be expensive in this situation in terms of memory overhead. Additionally, the Hash table STM system may be extended to object level granularity by mapping object pointers into the orec hash table. In this section, we will qualitatively evaluate the algorithm and also propose an alternative based on the LL/SC instruction to avoid a potential scalability issue with the CAS based design.

3.2.1 Contention Management

In the Hash table STM system a transaction aborts a conflicting ACTIVE transaction giving rise to the possibility of livelocks. The system is hence obstruction-free. The conflict resolution policy, between ACTIVE transactions, applied in this system is "Aggressive". Conflict resolution in obstruction-free synchronization algorithms has been more formally defined as *Contention Management* [Herlihy 03b].

Herlihy et al. have shown that contention management policies play an important role in gaining higher throughput in obstruction-free synchronization algorithms [Herlihy 03b]. A simple "Polite" contention management policy (a transaction employs exponential backoff on detecting contention and aborts the conflicting transaction after reaching a maximum backoff limit) performs significantly better than the "Aggressive" policy. The Hash table STM system gives opportunities for better contention management at two different execution points of a transaction's life cycle, the read conflict point and the acquire conflict point (as discussed in Section 3.1). Employment of a better contention management policy will certainly increase the throughput of the Hash table STM system.

3.2.2 The Bounded Memory Blow-up Problem

During the STMCommit operation, on detecting conflicts a transaction steals the orecs under conflict from the conflicting transactions. In the process of stealing, the transaction merges into its descriptor the conflicting transaction's entries corresponding to the orec under conflict. The merging step has a potentially significant scalability issue. The merging step merges into the stealer the transaction entries of memory words that are not even accessed by the stealer. Consequently, the stealer may end up possessing several transaction entries it is not concerned with, that coincidentally hash into an orec that the transaction has stolen. This kind of *false sharing* leads potentially to long *merge chains* of transaction entries in a transaction descriptor.

Let the ratio of the application heap size to the orec hash table size be $M : 1$. Correspondingly, if the hashing of memory locations to the orec hash table is uniform, each orec covers approximately M different shared memory words. In the worst-case scenario, for each memory location that a transaction may update, it may end up possessing $M - 1$ extra transaction entries. If a transaction needs to acquire N orecs, it will end up possessing $N \times M$ transaction entries in the process. This bounded memory blow-up may be tolerable if M is small. But in practice, M may be large. Although the worst case scenario may occur rarely, it should be noted that conflicts on orecs increase with increasing contention. During low contention the bounded memory blow-up problem may not be significant. From moderate to high contention the merge chains may grow unacceptably. The bounded memory blow-up problem is particularly intolerable to applications that may use fixed

sized transaction descriptors.

A side effect of the bounded memory blow-up problem is the direct impact on the efficiency of the *release* phase in the STMCommit operation. With longer chains of transaction entries, transactions in the release phase that have their orecs stolen have to redo the updates from each stolen orec's most recent owner transaction's descriptor. The longer the extended chains of transaction entries, the slower the transaction release phase is going to be. An alternative to the *merge-redo* mechanism in the STMCommit operation is the helping mechanism. Harris and Fraser have argued that helping leads to thrashing of the cache in the scenario of heavy contention [Harris 03]. For example, if process A and process B are executing on different processors simultaneously and try to access the same set of memory locations, they will keep fighting for these memory locations, thus wasting CPU cycles and memory bus bandwidth. But the bounded memory blow-up problem may even undo the potential efficiency advantages gained by avoiding helping. This needs to be analyzed empirically.

3.3 An Alternative LL/SC Based Approach of Stealing

The stealing mechanism has been employed to ensure nonblocking semantics to the algorithm. A reference counter has also been introduced as a part of the orec to avoid the stale updates problem. The update of the version number (transaction descriptor reference if the orec is acquired by a transaction) and the reference count of an orec must be done together atomically. The current design uses two separate words to represent a version number and the reference count of an orec. An atomic update to these two words can be done using a two-word CAS operation (or a CASX as per Sun Microsystems terminology). Unfortunately not all architectures provide a CASX-like operation.

An alternative to using CASX is to embed the reference count and the version number together in a one-word long orec. But embedding the reference count and the version/reference information in a single word is still not a very clean alternative solution. We propose another alternative approach that is free of the CASX operation, and provides a different view of incorporating obstruction-freedom in the algorithm. This approach is based on the use of the LL/SC (Load Linked / Store Conditional) instruction, instead of CAS and CASX.

Our approach is basically along the same lines of using a hash table, but uses the idea of *helping* during stealing instead of the current *merge-redo* mechanism. Everything else, except for the stealing mechanism, remains the same in our new variant. In our stealing mechanism, the stealer transaction does not merge transaction entries to its descriptor. Instead, it writes the values, corresponding to the orec under conflict, from the conflicting transaction's descriptor entries (either the old or the new values depending on whether the conflicting transaction has aborted or committed respectively) to the memory locations during stealing itself. The writing is done using LL/SC as follows : The stealer does a Load-Linked on the memory location, double-checks the orec (to check if the orec has not been stolen in between), and then does a Store-Conditional to the memory location. No merging happens, only helping happens. After successfully doing a LL/SC from a transaction entry, the stealer may mark the entry as invalid (so that no one else does an update subsequently). This marking may be useful to reduce the cache thrashing caused while helping.

An important side-effect of our new stealing mechanism is that there is no need for using reference counts anymore. Consequently, we do not require an operation semantically equivalent to CASX and we greatly reduce the space overhead of the orec hash table. A significant amount of complexity in the stealing process is removed. Another important outcome is that merging never

happens; hence the bounded memory blow-up problem is eliminated. Our proposed modification may also be implemented using CAS. But CAS is susceptible to the ABA problem.[3] Version numbering can be employed to remove the ABA problem. But usage of version numbers may mandate use of the CASX operation. More importantly, version numbers are needed for every memory word. This defeats the basic memory efficiency motivation for using a hash table in this STM system. The semantics of LL/SC more elegantly support the helping mechanism than the CAS instruction variants. We can implement the existing stealing mechanism using LL/SC as well. In general a CAS instruction can be simulated with a LL/SC instruction. A CAS is always preceded by a read; correspondingly Load Linked implements the same read semantics and additionally stores the address of the word in the LL/SC register in the processor. Both these operations are executed atomically. The Store Conditional that follows updates the memory word, Load Linked previously, after verifying that the word has not been modified in between by some other concurrent process. This is done simply by confirming that the address of the word is still in the LL/SC register. The underlying cache coherence protocol is responsible for flushing out the LL/SC register if the corresponding word is overwritten by another processor. Overlooking these implementation details, at highest level of abstraction LL/SC appears to be semantically equivalent to the CAS instruction. A detailed comparative analysis between LL/SC and CAS is outside the scope of this paper.

4 Object-based Software Transactional Memory Systems

Recent trends in STM system designs have shown an inclination toward supporting object level synchronization. Since many applications allocate and use data structures dynamically, object-based STM systems are of more practical importance than word-based STM systems. Word-based STM systems are more suitable for data structures that may require concurrent access at a high level of granularity (e.g. multi-dimensional arrays). Conventional concurrent systems employing object-based atomic operations use blocking synchronization techniques. Synchronization of highly complex data structures, like red-black trees [Herlihy 03b], using blocking mechanisms requires very careful (and highly complex) designing of the concurrent system to avoid pitfalls like deadlock, convoying, etc. Very recently, efforts have been taken in the direction of developing nonblocking object-based STM systems [Fraser 03, Herlihy 03b]. Support for practical and dynamic data structures has been the biggest motivation toward the development of these object-based STM systems. In this section we will discuss the designs of two dynamic object-based STM systems of Herlihy et al. [Herlihy 03b] and Fraser [Fraser 03].

4.1 Dynamic Software Transactional Memory (DSTM)

Herlihy et al. have proposed an obstruction-free STM system called the DSTM [Herlihy 03b] for concurrent access to dynamic objects. The obstruction-freedom property provides DSTM with greater design simplicity, flexibility and resulting efficiency than comparable lock-free STM systems. Although obstruction-freedom does not guarantee progress, Herlihy et al. have introduced a novel idea of contention management that ensures progress in practice. Additionally, Herlihy et al. have introduced other novel ideas of early release and invisible reads that are useful in decreasing overall contention.

[3] A CAS is generally preceded by a read on the memory location to be updated. There is always a window between a read and its corresponding CAS. In this window, the memory location may be overwritten several times by other concurrent processes. The ABA problem occurs when the last writer to the memory location writes a value expected by the process doing the subsequent CAS on that memory location. The result is a stale update.

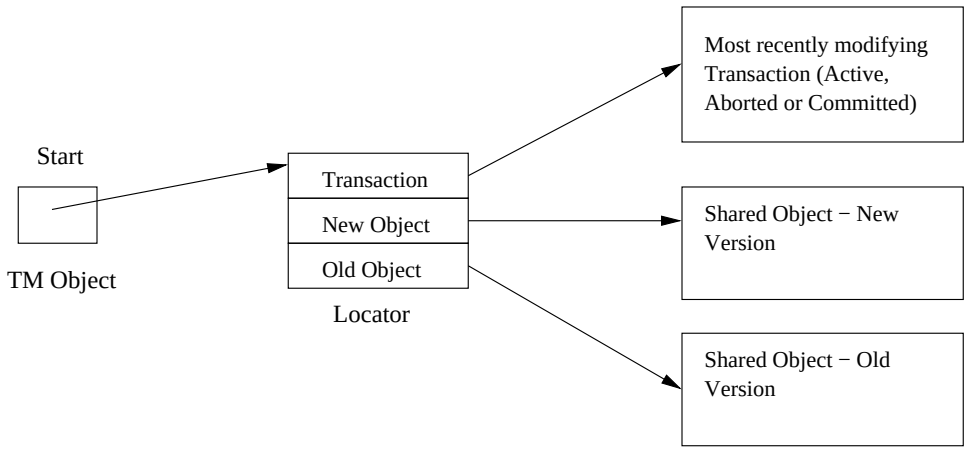

Figure 3: Transactional Memory Object (TM Object) Structure

4.1.1 The DSTM Design

Figure 3 depicts the architecture of a dynamic *Transactional Memory Object (TM Object)*. The TM Object acts as a wrapper around the concurrent data object. The TM Object contains a pointer to a *locator* object. The locator stores a pointer to a descriptor of the most recent transaction that tried to modify the TM Object and a pointer to old and new versions of the data object. A transaction descriptor pointed to by a locator may be in any of the three states : ACTIVE, ABORTED or COMMITTED. The most recent valid version of the data object is determined by the state of the most recently modifying transaction at any given time.

- If the transaction is ACTIVE or ABORTED, the most recent valid version of the data object is the old version referenced by the locator.
- If the transaction is COMMITTED, the most recent valid version of the data object is the new version referenced by the locator.

The locator can be viewed as a variant of the orec (defined by Harris and Fraser [Harris 03]) except for a few key design differences.

- The locator is referenced by the TM Object whereas the orec corresponding to a memory word is found by a hash function.
- The locator points to the old and new versions of the data object, whereas the orec either stores a version number or points to a transaction descriptor that stores the old and new versions of the data object.
- The locator does not require a version number. Rather it always stores a pointer to the most recent valid version of the data object.

The locator is the main key to the design of the DSTM.

A transaction must access a data object via its wrapper TM Object. To access the data object the transaction *opens* the corresponding TM Object. During the open operation, the transaction declares its "interest" in using the corresponding data object to the concurrent system. The transaction first locally creates a new copy of the locator object. The transaction pointer within this

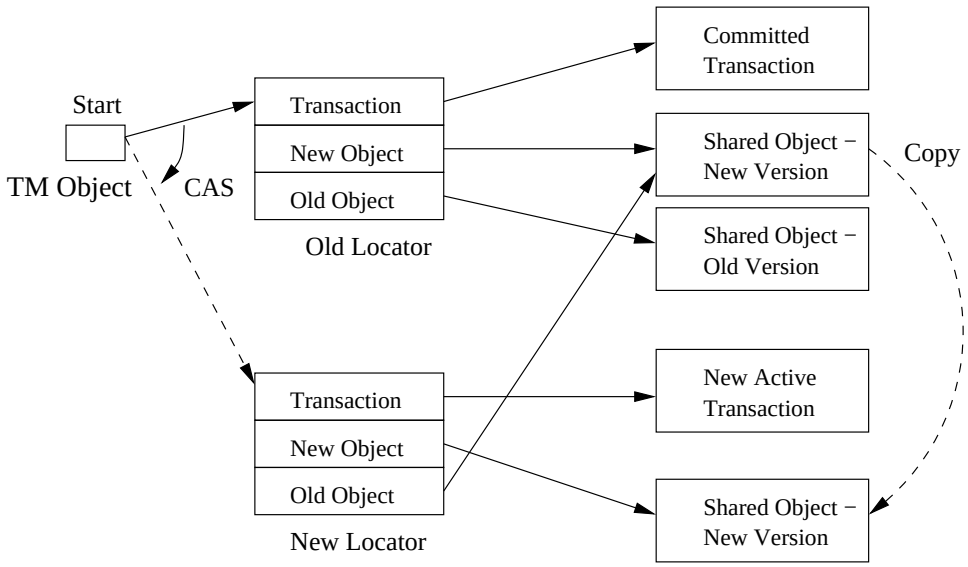

Figure 4: Opening of a TM Object recently modified by a committed transaction [Herlihy 03b]

locator points to creator transaction itself. The new locator's contents depend on the state of the most recently modifying transaction in the old locator as follows :

- If the old locator refers to an ABORTED transaction, the new locator's old object pointer points to the old object version referenced by the old locator. The new locator's new object pointer points to a copy of this old object version.
- If the old locator refers to a COMMITTED transaction, the new locator's old object pointer points to the new object version referenced by the old locator. The new locator's new object pointer points to a copy of this new object.
- If the old locator points to a transaction that is still ACTIVE it means that another transaction is potentially interested in modifying the same TM Object (this means that the old locator of the TM Object under conflict was created by the conflicting transaction). We have a conflict. This conflict is resolved using a novel contention management scheme. The result is that one of the two conflicting transactions is ABORTED. If the transaction that detects the conflict is not ABORTED (that means that the conflicting transaction was ABORTED) its new local locator's old object pointer points to the old object version referenced by the old locator. The new locator's new object pointer points to a copy of the old object version. This is an example of an early conflict resolution in the life cycle of a transaction. As we shall see later, early conflict resolution is a key difference DSTM and the competing system of Fraser, FSTM [Fraser 03 (Section 4.2)].

The transaction then replaces the old locator referenced by the TM Object with the new local locator using an atomic CAS operation. A successful CAS guarantees that the current transaction is visible to the entire concurrent system. A failure in CAS implies that some other transaction has opened (acquired) the TM Object in between. At this point, the current transaction must retry to acquire the TM Object. Figure 4 depicts the open operation of a TM Object after a recent commit.

Initially the status of a transaction is ACTIVE. It can open (or acquire) multiple TM Objects, make modifications to the new versions, and do an atomic CAS on its status to COMMITTED,

provided its status is still ACTIVE. In the presence of contention a transaction finds the status of the most recently modifying transaction (pointed by the locator of the TM Object) as ACTIVE. The transaction has the option to either abort itself or the current owner of the TM Object. This scenario may lead to livelocks. Hence DSTM turns out to be obstruction-free. In the face of contention a transaction goes through a novel *contention management* protocol to decide whether to abort itself or the TM Object's current ACTIVE owner transaction. The contention manager is important from the perspective of performance. Herlihy et al. have proposed a few rudimentary contention managers like the Aggressive manager (that aborts any conflicting transaction), and the Polite Manager (that uses exponential backoff to acquire ownership of a TM Object). The performance of the Polite contention manager is significantly better than that of the Aggressive contention manager. Choosing the best possible contention manager is crucial to the performance of a DSTM implementation. Scherer and Scott have recently proposed and evaluated more sophisticated contention management protocols [Scherer 04].

Herlihy et al. have also introduced the technique of *early release* to reduce contention. The idea is that a transaction may release an open object before committing. Such an optimization may be necessary for efficiency reasons in data structures like trees (where for example the root node should not be kept open for a long time by any transaction that is not updating it, since all the accesses to the tree are made via the root node). It is the programmer's responsibility to ensure consistency in the presence of early releases.

Many transactions require read-only access to concurrent objects. The open operation as described above may cause unnecessary contention in the presence of multiple transactions trying to open a TM Object in read-only mode. To avoid this unnecessary contention separate semantics are required for the read-only open operation. Each transaction maintains a separate *read-list* of the objects that it opens in read-only mode. This list is not reachable (visible) from the corresponding TM Object locators. While this approach reduces unnecessary contention it may lead to the transaction having an inconsistent view of memory. While a transaction, say A, maintains a read-only reference to a TM Object in its read-list, another transaction, say B, may modify that object without the knowledge that the modified object is being used by transaction A. Consequently transaction A will have an inconsistent view of the memory. To avoid this inconsistent state of a transaction *incremental validation* of the transaction is required. Whenever a transaction tries to open a TM Object, all the objects in its read-list are verified for consistency. Validation also happens immediately before the commit point.

4.2 FSTM

A lock-free object-based STM was recently developed at the University of Cambridge by Fraser for his doctoral thesis [Fraser 03]. The lock-freedom property in this STM system has been achieved by the use of order-based recursive helping.

4.2.1 The FSTM Design

Each concurrent object is wrapped in an *Object Header*. A transaction may gain access to concurrent objects by *opening* their object headers. The open call returns a pointer to the concurrent object encapsulated within the object header. A transaction may open several object headers at a time. Each transaction uses a *transaction descriptor* data structure to maintain the list of in-use concurrent objects. Along with a transaction status flag, the transaction descriptor contains a read-only list for the concurrent objects opened in read-only mode and a read-write list for the concurrent objects opened in read-write mode. The two lists contain *object handles*. An object handle

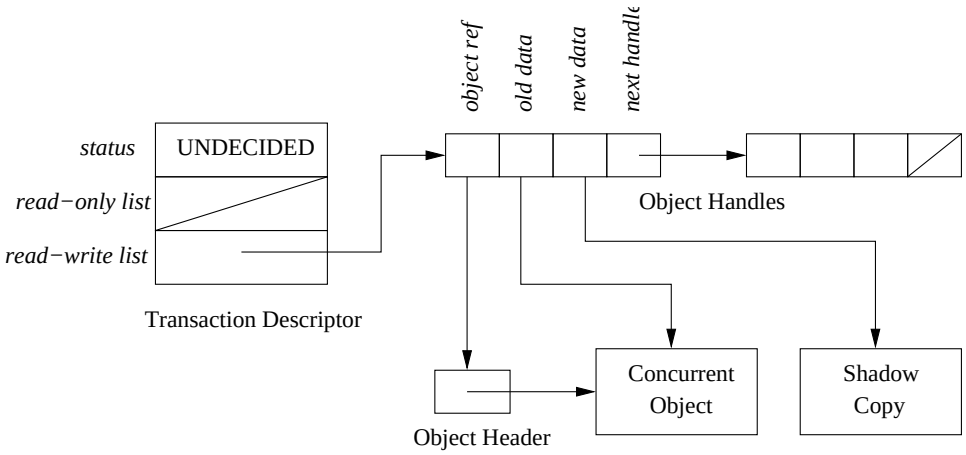

Figure 5: The basic Transactional Memory Structure in FSTM

consists of references to an object header, the concurrent object referenced by the object header, and a *shadow copy* of the concurrent object. The transaction does all its updates to this shadow copy of the concurrent object. It should be noted that the shadow copy is local to the transaction. In contrast to the conventions of DSTM, multiple transactions may open the same concurrent object and have their own local shadow copy for that object. A transaction may be in any of the four different states : UNDECIDED, ABORTED, COMMITTED, or READ-CHECKING. The first three states are straightforward to understand. We will discuss the READ-CHECKING state later in this section. Figure 5 shows an example transaction descriptor used by a transaction to access a concurrent object in read-write mode. The object header is a simple pointer to the concurrent object. What is not clear from the figure is that the object header may also point to a transaction descriptor when the corresponding transaction owns the object header. This will become clearer later in this section.

Whenever a transaction needs access to a concurrent object it first *opens* the object using that object's header. The open operation creates an object handle in the transaction's descriptor. If the object is opened in read-only mode the transaction adds the corresponding object handle to its read-only list. Otherwise it adds the object handle to its read-write list. If the handle is added to the read-only list there is no need to create a shadow copy of the concurrent object. Note that the open operation does not make the opening transaction visible to other transactions in the concurrent system; the commit operation does this. Conflicts are detected only with transactions that are trying to commit. Such transactions are recursively helped by the current transaction that detects these conflicts. It should be noted that recursive helping (by a transaction opening concurrent objects) does not lead to cyclical helping[4] since the helping transaction does not have any acquired concurrent objects.

The commit operation is basically a multi-word CAS. There are three phases in the commit operation: the *acquire phase* (where the transaction gains exclusive ownership of concurrent objects opened), the *decision point* (where the transaction either commits or aborts), and the *release phase* (where the transaction releases the ownership of all the acquired concurrent objects). In the acquire phase a transaction acquires each concurrent object in its read-write object handle list, in

[4]Cyclical helping occurs when helping transactions form a help-cycle that results in a livelock and no one makes progress subsequently.

some global total order, using an atomic CAS operation for each object. If a CAS fails because another transaction has modified a concurrent object the current transaction has to abort. On the other hand, if a conflict is encountered with an uncommitted transaction, the current transaction recursively helps the conflicting transaction. The global total ordering for acquiring concurrent objects avoids cyclical recursive helping that may lead to livelocks. After the acquire phase is successfully completed the transaction decides to commit or abort. After this point the READ-CHECKING phase decides whether the transaction is committed or aborted. Subsequently the transaction releases all the acquired concurrent objects in its release phase.

Acquiring and releasing the concurrent objects opened in read-only mode is unnecessary. In fact, it may result in unnecessary contention between transactions opening common objects in read-only mode. A typical example of such a scenario is concurrent accesses to tree-based data structures, where all the requests go through the root node. In FSTM, this may lead to unnecessary helping and aborting of transactions. Hence, before committing, the transaction needs only to verify the consistency of the concurrent objects in its read-only list (i.e. the transaction does not acquire the concurrent objects in its read-only list, it just verifies whether the objects have not been changed since they were last read by the transaction). Fraser [Fraser 03] calls this the *read phase*. If a conflict is detected with a COMMITTED transaction, the current transaction has to abort. If the conflict occurs with an UNDECIDED or ABORTED transaction, the current transaction traverses the read-write list of the conflicting transaction's descriptor and verifies that the object has not yet been changed. This approach clearly leads to non-serializable transactions.

Fraser gives a classic example [Fraser 03] of non-serializable transactions : Consider two concurrent transactions, T1 and T2, which access concurrent objects such that T1 opens some objects in read-write mode that T2 has opened in read-only mode and vice-versa. If both pass through their read-phase simultaneously (detecting that the objects under conflict have not been changed since they were last read) and commit, they cannot be serialized.

To avoid the problem of non-serializability introduced due to the read phase Fraser has introduced a new READ-CHECKING state for a transaction. After the acquire phase a transaction atomically switches over to the READ-CHECKING state and begins its read phase. In the read phase, the transaction walks through its descriptor's read-only list verifying the consistency of the read-only objects. If a conflict is detected with a transaction in UNDECIDED state the current transaction simply verifies that the old data object in the object handle (of the conflicting transaction's descriptor) for the concurrent object under conflict is not different from the old data object referenced by the object handle within the read-only list of the current transaction. If the data object is different, the current transaction has to abort. It should be noted that the current transaction (in its READ-CHECKING state) does not help the conflicting transaction (in UNDECIDED state). The insight in this design decision is that the transactions can still be serialized (where the transaction in READ-CHECKING state happens before the transaction in UNDECIDED state). This is a non-trivial technique of overlooking apparent conflicts and still achieving consistency. If the current transaction detects a conflict with a transaction in READ-CHECKING state then it either helps or aborts the conflicting transaction. The helping decision is made based on some global total ordering of transactions. A conflicting transaction that precedes the current transaction in the global total ordering is helped by the current transaction. A conflicting transaction that succeeds the current transaction in the global total ordering is aborted by the current transaction. Here the total global ordering of transactions ensures that there will be no cyclical helping in the system and progress is guaranteed for at least one transaction. Consequently the algorithm is lock-free.

5 A Qualitative Comparison

In the last two sections we have discussed the designs of the most recent STM systems. This section strives to make a qualitative comparison between the design peculiarities of these STM systems. The Hash table STM is more appropriate to word-based non-blocking synchronization, but can easily be extended to support objects as well : a shared memory location hashing into the ownership records hash table may contain a pointer to a concurrent object instead of simple word-sized data. We have already gone through a significant qualitative analysis of the Hash table STM in Section 3. Several of the design decisions in FSTM and Hash table STM are similar (e.g. multi-word CAS commit operation). Hence our comparison will mostly bring out differences between DSTM and the other two STM systems. The arguments applicable to helping in FSTM are obviously not applicable to Hash table STM. Of the three STM systems only FSTM exhibits lock-free semantics; the other two are obstruction-free algorithms.

5.1 Object Acquire Semantics

In the DSTM design, a transaction atomically acquires a concurrent object using a CAS operation while *opening* it for read-write access. As a result, for the acquired object, the transaction becomes visible to the concurrent system fairly early in its lifetime, long before it tries to commit. We can term this strategy *eager acquire*. On the other hand, in FSTM and Hash table STM, transactions acquire concurrent objects (to be updated) not while opening the object, but at commit time. As a result, a transaction is visible to the concurrent system only at its commit time. This strategy can be termed *lazy acquire*. An obvious observation about these acquire semantics is that eager acquire enables earlier detection of conflicts than lazy acquire does. Early detection of conflicts enables their early resolution. Conversely, lazy acquire semantics lead to extraneous expenditure of computational resources before conflicts are detected. The cost of late conflict detection may be significant in concurrent systems tending to have *long* transactions. The overhead of late conflict detection in lazy acquire needs to be studied empirically.

On the flip side, eager acquire may lead to unnecessary abortion of transactions. For example, consider a transaction A that has acquired an object O and is still in its ACTIVE state. Another transaction, say B (that has already acquired another concurrent object, say O'), comes along and tries to open object O. B detects a conflict and wins over A in the contention management protocol. B subsequently aborts A. Meanwhile another transaction, say C, tries to open object O'. C detects a conflict with B. If the contention management framework lets C win over B, C aborts B. C subsequently progresses to its completion. In this process, transaction A is aborted unnecessarily. In the same scenario, use of lazy acquire could result in the success of transactions A and C (provided C aborts B before B detects a conflict with A). Lazy acquire turns out to be more effective in this case since it narrows down the window of visible conflicts significantly. It is necessary to empirically evaluate the overhead of the superfluous abortions in eager acquire. The tradeoffs between eager and lazy acquire semantics may be crucial to the efficiency of the STM systems reviewed in this paper.

Hash table STM can be modified to incorporate eager acquire semantics as well. The acquire phase in the commit operation may be implemented in the open operation (STMRead and STMWrite functions in the case of Hash table STM) instead. The lazy acquire semantics could be incorporated in DSTM as well. The open operation (where a transaction acquires all the concurrent objects it intends to modify) would not contain the CAS operation. As a result, the commit operation would become a heavyweight multi-word CAS operation (as in Hash table STM and FSTM). On the other hand, eager acquire semantics cannot be incorporated in the FSTM design. The property of lock-

freedom is preserved in FSTM by ensuring a global total order for acquiring concurrent objects. This global total order guarantees that there will be no helping cycles. For FSTM to incorporate eager acquire semantics a transaction would have to have *pre-knowledge* about the concurrent objects it intends to open so that the objects could be opened in the global total order. Consequently, a transaction would not be allowed to make on-the-fly decisions about which concurrent objects to access and modify. The overall dynamism of FSTM would be crippled. This is yet another example of the design flexibility that obstruction-freedom allows while lock-freedom does not. On the flip side, the eager acquire problem in FSTM also shows the greater design flexibility that lazy acquire enables as compared to the eager acquire semantics. In fact, lazy acquire is a key aspect of the lock-freedom property of FSTM.

5.2 Indirection Overhead

The commit operation of DSTM relies on a single CAS operation on the status field of the transaction's descriptor. This is a lightweight commit operation as compared to the multi-word CAS commit of FSTM and Hash table STM. In the absence of contention, if a transaction updates N concurrent objects, DSTM requires $N + 1$ CAS operations for the transaction to commit. In comparison, in FSTM and Hash table STM the transaction requires $2N + 1$ CAS operations. This CAS-count difference stems from the design choices of placement of the ownership records (orecs) in these STM systems. The DSTM places the orec (the *locator* in DSTM terminology) within the TM Object, and the orec points to the most recent version of the concurrent object. FSTM and Hash table STM place the orec (object handle in FSTM's case) in the transaction descriptor. DSTM decreases the cost of a commit operation at the expense of an additional level of indirection (due to locators) in the TM Object. During low contention, this indirection overhead may result in slower transactions in DSTM than in FSTM and Hash table STM. The indirection overhead may be significant in concurrent systems that mostly contain reads and rarely any updates to the concurrent objects. A concrete empirical evaluation needs to be done to measure the overhead of the indirection in DSTM in the presence of low contention. It should be noted that the indirection overhead factor and the eager/lazy acquire semantics are independent design decisions and can be evaluated separately.

Correction, Feb. 2006: It's actually 2N+2; 2N to install and clean up pointers to descriptors; 1 to enter read-checking mode; 1 to commit.

5.3 Space Usage

The TM Object in the DSTM points to a locator object. The locator in turn contains three references : one to the transaction that most recently tried to modify the TM Object, one to the old version of the concurrent object, and one to the new version of the concurrent object. The state of the transaction determines which object version is the most recent valid version. The space occupied by a TM Object is a composite of all these objects (including the locator and the reference to the locator). The object reference in FSTM either contains a pointer to the most recent valid version of the concurrent object or a reference to the transaction that currently owns that object reference. As can be clearly seen, the space requirement of DSTM is more than twice as that of FSTM. For small concurrent objects (a common case with data structures like linked lists) this space overhead could be significant. In case of very large concurrent objects, this overhead may be prohibitively high because of the excessive space used up by invalid copies of the objects pointed to by the corresponding locators.

Correction, Feb. 2006: DSTM does NOT necessarily require double space: A reader that finds a locator whose owner transaction has committed or aborted can zero out the pointer to the no-longer-valid data copy, making that copy available for garbage collection. An ordinary write suffices for the update: once the owner transaction has finished the locator is no longer subject to modification or reinterpretation.

5.4 Search Overhead

In a transaction, FSTM and Hash table STM maintain lists of acquired objects in the transaction's descriptor. When a transaction detects a conflict, it has to search the descriptor of the conflicting transaction for the concurrent object under conflict. This search overhead is absent in DSTM. In FSTM this overhead could be eliminated by storing a pointer in the object reference to the object handle (corresponding to that object reference) in the owner transaction's read-write list. The object handle in turn would contain a reference to its creator transaction. This solution cannot be applied to Hash table STM since an orec may refer to several transaction entries in a descriptor. Note that the this search overhead does not apply to objects opened in read-only mode since they are never acquired.

5.5 Contention Management Versus Helping

FSTM employs recursive helping to ensure lock-freedom. Although lock-freedom may be a desirable property in a nonblocking synchronization algorithm, ensuring just obstruction-freedom simplifies the design of the algorithm and renders more design simplicity, flexibility and efficiency of execution. Helping in particular is considered to cause high contention for cache blocks between processors. Employing a good contention management policy may lead to excellent performance in an obstruction-free synchronization algorithm. An empirical comparison between contention management and helping is yet to be made. In fact, contention management schemes are still being actively researched.

5.6 Transaction Validation

In STM systems a transaction may update multiple concurrent objects atomically. Although the update is semantically atomic the implementation typically requires a multi-word CAS operation. A transaction may enter an inconsistent state during its execution long before it detects the inconsistency. More specifically, it may begin to work with data whose apparent values will never occur concurrently in a linearizable execution. Such yet to be detected inconsistencies may lead to serious problems like memory access violations and infinite loops. Hence transaction validation becomes necessary. Transaction validation at every step [Herlihy 03b] may be employed to ensure that the transaction is consistent during its execution. Herlihy et al. [Herlihy 03b] have included transaction validation in each STM operation. As a result, validation happens during each STM open operation. This incremental validation guarantees that the transaction never works with mutually inconsistent data. Although incremental validation is the only transaction validation option from a theoretical perspective it has some computational overhead. Alternatively, the application programmer may be given the responsibility of transaction validation. Along with providing a separate function of transaction validation to the programmer, Fraser proposes a mechanism based on exception handling [Fraser 03] to reduce the incremental validation cost. On a memory access violation, the exception handler is designed to validate the transaction that caused the exception. The responsibility of detecting other inconsistencies is left to the application programmer.

The need for incremental validation depends on the concurrent system in which the STM is used. In concurrent systems using complex data structures like red-black trees the decision of making incremental validations at different points in the application code may be highly non-trivial. In such cases, built-in incremental validation in the STM operations becomes the choice for the application programmer. Ideally, an STM system should provide the application programmer with both the STM APIs : the one with built-in validation STM operations, and the one with a

separate transaction validation function. Additionally, the cost of incremental validations needs to be empirically measured so as to determine whether the overhead is significant enough to consider the other alternative.

6 Future Directions

In this paper we have discussed the designs of recent STM systems. We have also done some isolated (in the case of Hash table STM) and some comparative qualitative analyses of these STM systems. To get a better insight into our perceptions about the design peculiarities we need to conduct experiments that test the tradeoffs among the specific STM properties that we mentioned in Section 5. Additionally, we need to study general data structures with respect to each of the STM systems and determine which is best supported by which STM system. Finally, we need to compare the performance of STM to that of high quality lock-based algorithms to assess the marginal cost of STM's semantic advantages (fault tolerance, lack of priority inversion, etc.)

References

[Case 78] R. P. Case and A. Padegs. *Architecture of the IBM System 370.* Communications of the ACM, 21(1):73–96, January 1978.

[Fraser 03] K. Fraser. *Practical Lock-Freedom.* Ph.D. Thesis, King's College, University of Cambridge, September 2003.

[Harris 03] T. Harris, and K. Fraser. *Language Support for Lightweight Transactions.* In Proceedings of OOPSLA'03, October 2003.

[Herlihy 90] M. Herlihy, and J. M. Wing. *Linearizability: A Correctness Condition for Concurrent Objects.* ACM Transactions on Programming Languages and Systems, 12(3):463–492, July 1990.

[Herlihy 91] M. Herlihy. *Wait-Free Synchronization.* ACM Transactions on Programming Languages and Systems, 13(1):124–149, January 1991.

[Herlihy 93] M. Herlihy, and J. E. B. Moss. *Transactional Memory: Architectural Support for Lock-Free Data Structures.* In Proceedings of the 20^{th} International Symposium on Computer Architecture, May 1993.

[Herlihy 03a] M. Herlihy, V. Luchangco, and M. Moir. *Obstruction-Free Synchronization: Double-Ended Queues as an Example.* In Proceedings of the 23^{rd} International Conference on Distributed Computing Systems, May 2003.

[Herlihy 03b] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. *Software Transactional Memory for Dynamic-Sized Data Structures.* In Proceedings of the 22^{nd} Annual ACM Symposium on Principles of Distributed Computing, July 2003.

[Jensen 87] E. H. Jensen, G. W. Hagensen, and J. M. Broughton. *A New Approach to Exclusive Data Access in Shared Memory Multiprocessors.* Technical Report UCRL-97663, Lawrence Livermore National Laboratory, November 1987.

[Scherer 04] W. N. Scherer III, and M. L. Scott. *Contention Management in Dynamic Software Transactional Memory.* In Proceedings of the Workshop on Concurrency and Synchronization in Java Programs, St. John's, Newfoundland, Canada, July 2004.

[Shavit 95] N. Shavit, and D. Touitou. *Software Transactional Memory.* In Proceedings of the 14^{th} Annual ACM Symposium on Principles of Distributed Computing, August 1995.

[Stone 93] J. M. Stone, H. S. Stone, P. Heidelberger, and J. Turek. *Multiple Reservations and the Oklahoma Update.* IEEE Parallel and Distributed Technology, 1(4):58–71, November 1993.