Software Transactional Memory

Nir Shavit*

MIT and
Tel-Aviv University

Dan Touitou

Tel-Aviv University

Abstract

As we learn from the literature, flexibility in choosing synchronization operations greatly simplifies the task of designing highly concurrent programs. Unfortunately, existing hardware is inflexible and is at best on the level of a Load_Linked/Store_Conditional operation on a single word. Building on the hardware based transactional synchronization methodology of Herlihy and Moss, we offer *software transactional memory* (STM), a novel software method for supporting flexible transactional programming of synchronization operations. STM is non-blocking, and can be implemented on existing machines using only a Load_Linked/Store_Conditional operation. We use STM to provide a general highly concurrent method for translating sequential object implementations to lock-free ones based on implementing a k-word compare&swap STM-transaction. Empirical evidence collected on simulated multiprocessor architectures shows that the our method always outperforms all the lock-free translation methods in the style of Barnes, and outperforms Herlihy's translation method for sufficiently large numbers of processors. The key to the efficiency of our software-transactional approach is that unlike Barnes style methods, it is not based on a costly "recursive helping" policy.

1 Introduction

A major obstacle on the way to making multiprocessor machines widely acceptable is the difficulty of programmers in designing highly concurrent programs and data structures. Given the growing realization that unpredictable delay is an increasingly serious problem in modern multiprocessor architectures, we argue that conventional techniques for implementing concurrent objects by means of critical sections are unsuitable, since they limit parallelism, increase contention for memory and interconnect, and make the system vulnerable to timing anomalies and processor failures. The key to highly concurrent programming is to decrease the number and size of critical sections a multiprocessor program uses (possibly eliminating critical sections altogether) by constructing classes of implementations that are *non-blocking*[7, 15, 14]. As we learn ¿from the literature, flexibility in choosing the synchronization operations greatly simplifies the task of designing non-blocking concurrent programs. Examples are the non-blocking data-structures of Massalin and Pu [22] which use a Compare&Swap on two words, Anderson's [2] parallel path compression on lists which uses a special Splice operation, the counting networks of [5] which use combination of Fetch&Complement and Fetch&Inc, Israeli and Rappoport's Heap [18] which can be implemented using a three-word Compare&Swap , and many more. Unfortunately, most of the current or soon to be developed architectures support operations on the level of a Load_Linked/Store_Conditional operation for a single word, making most of these highly concurrent algorithms impractical in the near future.

Bershad [7] suggested to overcome the problem of providing efficient programming primitives on existing machines by employing operating system support. Herlihy and Moss [16] have proposed an ingenious hardware solution: transactional memory. By adding a specialized associative cache and making several minor changes to the cache consistency protocols, they are able to support a flexible transactional language for writing synchronization operations. Any synchronization operation can be written as a transaction and executed using an optimistic algorithm built into the consistency protocol. Unfortunately though, this solution is *blocking*.

This paper proposes to adopt the transactional approach, but not its hardware based implementation. We introduce *software transactional memory* (STM), a novel design that supports flexible transactional programming of synchronization operations in software. Though we cannot aim for the same overall performance, our software transactional memory has clear advantages in terms of applicability to todays machines, portability among machines, and resiliency in the face of timing anomalies and processor failures.

We focus on implementations of a software transactional memory that support static transactions, that is, transactions which access a pre-determined sequence of locations. This class includes most of the known and proposed synchronization primitives in the literature.

1.1 STM in a nutshell

In a non-faulty environment, the way to ensure the atomicity of the operations is usually based on locking or acquiring exclusively ownerships on the memory locations accessed by an operation *Op*. If a transaction cannot capture an ownerships

[1]Contact Author: E-mail: shanir@theory.lcs.mit.edu

it fails, and releases the ownerships already acquired. Otherwise, it succeeds in executing *Op* and frees the ownerships acquired. To guarantee liveness, one must first eliminate deadlocks, which for static transactions is done by acquiring the ownerships needed in some increasing order. In order to continue ensuring liveness in a faulty environment, we must make certain that every transaction completes even if the process which executes it has been delayed, swapped out, or crashed. This is achieved by a "helping" methodology, forcing other transactions which are trying to capture the same location to help the owner of this location to complete its own transaction. The key feature in the transactional approach is that in order to free a location one need only help its single owner transaction. Moreover, one can effectively avoid the overhead of coordination among several transactions attempting to help release a location by employing a "reactive" helping policy which we call *non-redundant-helping*.

1.2 Sequential to Lock-free Translation

One can use STM to provide a general highly concurrent method for translating sequential object implementations into non-blocking ones based on the caching approach of [6, 26]. The approach is straightforward: use transactional memory to to implement any collection of changes to a shared object, performing them as an atomic k-word Compare&Swap transaction (see Figure 2) on the desired locations. The non-blocking STM implementation guarantees that some transaction will always succeed.

Herlihy, in [15] (referred to in the sequel as *Herlihy's method*), was the first to offer a general transformation of sequential objects into non-blocking concurrent ones. According to his methodology, updating a data structure is done by first copying it into a new allocated block of memory, making the changes on the new version and tentatively switching the pointer to the new data structure, all that with the help of Load_Linked/Store_Conditional atomic operations. Unfortunately, *Herlihy's method* does not provide a suitable solution for large data structures and like the standard approach of locking the whole object, does not support concurrent updating. Alemany and Felten [4] and LaMarca [20] suggested to improve the efficiency of this general method at the price of loosing portability, by using operating system support making a set of strong assumptions on system behavior.

To overcome the limitations of *Herlihy's method*, Barnes, in [6], introduced his *caching* method, that avoids copying the whole object and allows concurrent disjoint updating. A similar approach was independently proposed by Turek, Shasha, and Prakash [26]. According to Barnes, a process first "simulates" the execution of the updating in its private memory, i.e reading a location for the first time is done from the shared memory but writing is done into the private memory. Then, the process uses an non-blocking k-word *Read-Modify-Write* atomic operation which checks if the values contained in the memory are equivalent to the the value read in the cache update. If this is the case, the operation stores the new values in the memory. Otherwise, the process restarts from the beginning. Barnes suggested to implement the k-word Read-Modify-write by locking in ascending order of their key, the locations involved in the update executing the operation and releasing the locks. The key to achieving the non-blocking resilient behavior in the caching approach of [6, 26] is the *cooperative method*: whenever a process needs a location already locked by another process it helps the locking process to complete its own operation, and this is done recursively along the dependency chain. Though Barnes and Turek, Shasha, and Prakash are vague on specific implementation details, a recent paper by Israeli and Rappoport [19] gives, using the cooperative method, a clean and streamlined implementation of a non-blocking k-word Compare&Swap using Load_Linked/Store_Conditional . However, as our empirical results suggest, both the general method and its specific implementation have two major drawbacks which are overcome by our STM based translation method:

- The *cooperative method* has a recursive structure of "helping" which frequently causes processes to help other processes which access a disjoint part of the data structure.
- Unlike STM's transactional k-word Compare&Swap operations which mostly fail on the transaction level and are thus not "helped," a high percentage of cooperative k-word Compare&Swap operations fail but generate contention since they are nevertheless helped by other processes.

Take for example a process P which executes a 2-word Compare&Swap on locations a and b. Assume that some other process Q already owns b. According to the cooperative method, P first helps Q complete its operation and only then acquires b and continues on its own operation. However, in many cases P's Compare&Swap will not change the memory since Q changed b after P already read it, and P will have to retry. All the processes waiting for location a will have to first help P, then Q, and again P, when in any case P's operation will likely fail. Moreover, after P has acquired b, all the processes requesting b will also redundantly help to P.

On the other hand, if P executes the 2-word Compare&Swap as an STM transaction, P will fail to acquire b, help Q, release a and restart. The processes waiting for a will have to help only P. The processes waiting for b will not have to help P. Finally, if Q hasn't changed b, P will most likely find the value of b in its own cache.

1.3 Our Empirical Results

To make sequential-to-non-blocking translation methods acceptable, one needs to reduce the performance overhead one has to pay when the system is stable (non-faulty). We present (see Section 5) the first experimental comparison of the performance under stable conditions of the translation techniques cited above. We use the well accepted Proteus Parallel Hardware Simulator [8, 9].

We found that on a simulated Alewife [1] cache-coherent distributed shared-memory machine, as the potential for concurrency in accessing the object grows, the STM non-blocking translation method outperforms both *Herlihy's method* and the *cooperative* method. Unfortunately, our experiments show that in general STM and other non-blocking

```
Dequeue()
    BeginTransaction
        DeletedItem = Read_transactional(Head)
        if DeletedItem = Null
            ReturnedValue = Empty
        else
            Write-transactional(Head,DeletedItem→Next)
        if DeletedItem→Next = Null
            Write-transactional(Tail,Null)
        ReturnedValue = DeletedItem→Value
    EndTransaction
end Dequeue
```

Figure 1: A Non Static Transaction

```
k_word_C&S(Size,DataSet[],Old[],New[])
    BeginTransaction
        for i=1 to Size do
            if Read_transactional (DataSet[i]]) ≠ Old[i]
                ReturnedValue = C&S-Failure
                ExitTransaction
        for i=1 to Size do
            Write_transactional (DataSet[i],New[i])
        ReturnedValue = C&S-Success
    EndTransaction
end k_word_C&S
```

Figure 2: A Static Transaction

techniques are inferior to standard *non-resilient* lock-based methods such as queue-locks [23]. Results for a shared bus architecture were similar in flavor.

In summary, STM offers a novel software package of flexible coordination-operation for the design of highly concurrent shared objects, which ensures resiliency in faulty runs and improved performance in non-faulty ones. The following section introduces STM. In Section 3 we describe our implementation and and provide a sketch of the correctness proof. Finally, in Section 5 we present our empirical performance evaluation.

2 Transactional Memory

We begin by presenting *software transactional memory*, a variant of the transactional memory of [16]. A transaction is a finite sequence of local and shared memory machine instructions:

Read_transactional – reads the value of a shared location into a local register.

Write_transactional – stores the contents of a local register into a shared location.

The *data set* of a transaction is the set of shared locations accessed by the Read_transactional and Write_transactional instructions. Any transaction may either fail, or complete successfully, in which case its changes are visible atomically to other processes. For example, dequeuing a value ¿from the head of a doubly linked list as in Figure 1 may be performed as a transaction. If the transaction terminates successfully it returns the dequeued item or an *Empty* value.

A k-word Compare&Swap transaction as in Figure 2 is a transaction which gets as parameters the data set, its size and two vectors *Old* and *New* of the data set's size. A successful k-word Compare&Swap transaction checks whether the values stored in the memory are equivalent to old. In that case, the transaction stores the New values into the memory and returns a *C&S-Success* value, otherwise it returns *C&S-Failure*.

A *software transactional memory* (STM), is a shared object which behaves like a memory that supports multiple changes to its addresses by means of transactions. A *transaction* is a thread of control that applies a finite sequence of primitive operations to memory. Any implementation of software transactional memory should satisfy the following standard properties [13]:

Atomicity: transactions appear to execute sequentially, i.e., without interleaving.

Serializability: The sequential order among transactions is consistent with their real-time order.

A *static* transaction is a special form of transaction in which the data set is known in advance, and can thus be thought of as a procedure which gets as parameters (1) the data set (2) the inputs of the transaction (3) a deterministic function which based on the inputs and the the data set, returns the new values which should be stored data set and the output of the transaction. This paper we will focus on implementations of a transactional memory that supports static transactions, a class that includes most of the known and proposed synchronization operations in the literature. The k-word Compare&Swap transaction in Figure 2 is an example of a static transaction, while the Dequeue procedure in Figure 1 is not.

An STM implementation is *wait-free* if any process which repeatedly executes the transaction terminates successfully after a finite number of attempts. It is *non-blocking* if the repeated execution of some transaction by a process implies that some process (not necessarily the same one and with a possibly different transaction) will terminate successfully after a finite number of attempts in the whole system. An STM implementation is *swap tolerant*, if it is non-blocking under the assumption that a process cannot be swapped out infinitely many times. The hardware implemented transactions of [16] could in theory repeatedly fail forever, if processes try to write two locations in different order (as when updating a doubly linked list). However, if used only for static transactions, their implementation can be made swap-tolerant (but not non-blocking, since a single process which is repeatedly swapped during the execution of a transaction will never terminates successfully).

3 Our Implementation of Static-STM

We implement a non-blocking static TM of size M using *Memory*$[M]$, a vector which contains the data stored in the transactional memory, *Ownerships*$[M]$, a vector which determines for any cell in *Memory*$[M]$, which transaction owns it. Each process keeps in the shared memory a record with

```
StartTransaction(input,DataSet)
    Initialize($Tran_j$,input,DataSet)
    $Tran_j \rightarrow$Stable = True
    Transaction($Tran_j$,$Tran_j \rightarrow$ version,True)
    $Tran_j \rightarrow$ Stable = False
    $Tran_j \rightarrow$ Version++
    if $Tran_j \rightarrow$Status = Success then
        return (Success,CalcOutput($Tran_j \rightarrow$ OldValues,input))
    else
        return Failure
```

Figure 3: StartTransaction

the following fields: *Size* which contains the size of the data set. *Add*[] – a vector which contains the data set addresses in increasing order. *Input* – the input of the transaction. *Oldvalues*[] a consensus vector which cells are initialized to *Null* at the beginning of every transaction. In case of a successful transaction this vector contains the former values stored in the involved locations. The output of the transaction is calculated from this vector and the input. The other fields are used in order to synchronize between the owner of the record and the processes which may eventually help its transactions: *Version*– an integer, initially 0, which determines the instance number of the transaction. This field is incremented every time the process terminates a transaction For every process P_j, $Tran_j$ determines the address of its record.

A process P_j initiates the execution of a transaction by calling the *Transaction* routine of Figure 3. Transaction first initializes the process's record then declares the record as *stable*, ensuring that any helping processors will read a consistent description of the transaction. After executing the transaction the process checks if the transaction has succeeded, and if so calculates the output from the input and the *OldValues* vector.

The procedure *Transaction* (Figure 4), gets as parameters *tran*, the record's address of the transaction executed, and a boolean value *IsInitiator*, indicating whether Transaction was called by the initiating process or by a helping process. The parameter *version* contains the instance number of the record executed[1] This parameter is not used when the routine is called by the initiating process since the version field will never change during the call. Transaction, first tries to acquire ownership on the data set's locations by calling *AquireOwnership*. If it fails to do so then upon returning from AquireOwnership, the status field will be set to *(Failure,failadd)*. If the status field doesn't have a value yet, the process sets it to *(Success,0)*. In case of success the process writes the old values into the transaction's record, calculates the new values to be stored, writes them to the memory and releases the ownerships. Otherwise, the status field contains the location that caused the failure. The process first releases the ownerships that it already owns and, in the case that it is not a helping process, it helps the transaction which owns the failing location. Helping is performed only if the record is in a *stable* state.

[1]The use of this unbounded field can be avoided if an additional *Validate* operation is available [18, 19].

```
Transaction(tran,version,IsInitiator)
    AcquireOwnerships(tran,version)
    (status,failadd) = LL(tran→status)
    if status = Null then
        if (version $\neq$ tran→version) then return
        SC(tran→status,(Success,0))
    (status,failadd) = LL(tran→status)
    if status = Success then
        AgreeOldValues(tran,version)
        NewValues = CalcNewValues(tran→OldValues,tran→input)
        UpdateMemory(tran,version,NewValues)
        ReleaseOwnerships(tran,version)
    else
        ReleaseOwnerships(tran,version)
        if IsInitiator then
            failtran= Ownerships[failadd]
            if failtran = Nobody then
                return
            else
                failversion = failtran→version
                if failtran→stable
                    Transaction(failtran,failversion,False)
```

Figure 4: Transaction

```
AcquireOwnerships(tran,version)
    transize = tran→size
    for i = 1 to size do
        while true do
            location = tran→add[i]
            if LL(tran→status) $\neq$ Null then return
            owner = LL (Ownerships[tran→Add[i]])
            if tran→version $\neq$ version return
            if owner = tran then exit while loop
            if owner = Nobody then
                if SC(tran→status, (Null , 0) ) then
                    if SC(Ownerships[location],tran) then
                        exit while loop
            else
                if SC(tran→status, (Failure,i) ) then
                    return

ReleaseOwnerships(tran,version)
    size = tran→size
    for i = 1 to size do
        location= tran→Add[i]
        if LL(Ownerships[location]) = tran then
            if tran→version $\neq$ version then return
            SC(Ownerships[location],Nobody)

AgreeOldValues(tran,version)
    size = tran→size
    for i = 1 to size do
        location= tran→Add[i]
        if LL(tran→OldValues[location]) $\neq$ Null then
            if tran→version $\neq$ version then return
            SC(tran→OldValues[location],Memory[location])

UpdateMemory(tran,version,newvalues)
    size = tran→size
    for i = 1 to size do
        location= tran→Add[i]
        oldvalue= LL(Memory[location])
        if tran→AllWritten then return
        if version $\neq$ tran→version then return
        if oldvalue$\neq$ newvalues[i] then
            SC(Memory[location],newvalues[i])
    if (not LL(tran→AllWritten)) then
        if version $\neq$ tran→version then return
        SC(tran→AllWritten,True)
```

Figure 5: Ownerships and Memory access

Since AcquireOwnerships of Figure 5 may be called either by the initiator or by the helping processes we must ensure that (1) all processes will try to acquire ownership on the same locations (this is done by checking the version between the Load_Linked and the Store_Conditional instructions) (2) from the moment that the status of the transaction becomes fixed, no additional ownerships are allowed for that transaction. The second property is essential for proving not only atomicity but also the non-blocking property. Any process which reads a free location will have before acquiring ownership on it, to confirm that the transaction status is still undecided. This is done by writing (with Store_Conditional) *(Null,0)* in the status field. This prevents any process which read the location in the past as owned by a different transaction, to set the status to *Failure*.

When writing the new values to the *UpdateMemory* as in Figure 5, the processes synchronize in order to prevent a slow process from updating the memory after the ownerships have been released. To do so every process sets the *AllWritten* field to be True, after updating the memory and before releasing the ownerships.

4 Correctness Proof Outline

Formally, following [21], the specification of a static transactional memory for n processes that supports k different static transactions can be described as an automaton with k types of input actions:
$Tran_j\,Request_i(DataSet)$ and k types of output actions:
$Tran_j\,Return_i(FinalStatus, Output)$ where $j \in 1 \ldots k$ and $i \in 1 \ldots n$.

In our implementation, any transaction T is related to a transaction record *tran*, and an instance number (the content of the version field). Therefore, we define the process which started the execution of T (which owns the record tran) as the *initiator* of T. All the processes which execute routine Transaction with parameters (tran,version,False), are defined to be the *helping* processes of T. The initiator and the helping processes are the *executing* processes of T.

Lemma 4.1 *The implementation is atomic and serializable.*

Sketch of proof: The proof of this lemma is based on the following invariants:

1. All the executing processes of a transaction T read the same data set vector which was stored by T's initiator. Any executing process of T which read a different data set will not be able to update any of the shared data structures.

2. All the executing processes of a transaction T will never acquire ownership after the the status of T has been set. All the ownerships owned by T will be released before the version field of T's record is incremented by T's initiator.

3. All the executing processes of a successful transaction T will update the memory before T's AllWritten field is set to True.

In order to prove the non-blocking property we first define that the *failing process* of a failing transaction T, is the executing process which wrote Failure to T's status. The *failing location* of T is the location that the failing process failed to acquire ownership on it.

Claim 4.2 *A failing transaction, T will never owns its failing location or a higher location that its failing location.*

Proof: (*Sketch*) Assume the contrary. Let P be an executing process which acquired an ownership on a higher location than the failing location. By the first invariant in Lemma 4.1, P acquired that ownership before the status was set to Failure. Therefore P has confirmed that the transaction's status is undefined before the failing process saw the failing location owned by another transaction. Now, if P has acquired ownership on a higher location then the failing process should have seen that the failing location belongs to T. Therefore P has acquired the failing location itself. But, in that case, since P saw the location free before the failing process saw it occupied and since the failing process saw the location occupied before P has executed the Store_Conditional instruction on the ownership, therefore the Store_Conditional instruction should has failed. ■

Lemma 4.3 *The implementation is non-blocking.*

Proof: (*Sketch*) Assume by way of contradiction that there is an infinite schedule in which no transaction terminates successfully. Assume that the number of transaction failures is finite. This happens only if from some point on, in the computation, all the processes are "stuck" in the AcquireOwnerships routine. In this case there are several processes which try to acquire ownership of the same location for the same transaction. This may happen only if the location is aquired and released infinitely often. Since every location is released only when the transaction is completed, it follows that there must be an infinite number of failing transactions. This in turn implies that there is at least one location on which processes fail infinitely often, and consider A, the highest such location. Since the initiator of the transaction tries to help the transaction which has failed him before retrying, it follows that there are infinitely many transaction which have acquired ownership on A but have failed. By Claim 4.2 those transactions have failed on addresses higher than A – a contradiction to the fact that A is the highest. ■

To avoid major overheads when no failures occur, any algorithm based on the helping paradigm must avoid as much as possible "redundant helping." In the STM implementation given above, redundant helping occurs when a failing transaction "helps" another non-faulty process. Such helping will only increase contention and consequently, will cause the helped process to release the ownerships later then it would have released if not helped. In our algorithm, a process increases or decreases the interval between helps as a function of the "redundant helps" it discovered.

5 An Empirical Evaluation of Translation Methods

5.1 Methodology

We compared the performance of STM and other software methods on 64 processor bus and network architectures using the Proteus simulator developed by Brewer, Dellarocas, Colbrook and Weihl [8]. Our network architecture was that of the Alewife cache-coherent distributed-memory machine currently under development at MIT [1]. Each processor had a cache with 2048 lines of 8 bytes and a memory access without contention cost 4 cycles in both architectures. The cost of switching or wiring in the Alewife architecture was 1 cycle/packet.

The current version of Proteus does not support Load_Linked/Store_Conditional instructions. Instead we used a slightly modified version that supports a 64-bit Compare&Swap operation where 32 bits serve as a time stamp. [2] On existing machines the 64 bits Compare&Swap may be implemented by using the a 64 bits Load_Linked/Store_Conditional as on the Alpha or using Bershad's lock-free methodology[3] [7].

We used four synthetic benchmarks for evaluating various methods for implementing shared data structures. The methods vary in the size of the data structure and the amount of parallelism.

Counting Each of n processes increments a shared counter $10000/n$ times. In this benchmark updates are short, change the whole object state, and have no built in parallelism.

Resource Allocation A resource allocation scenario [10]: a few processes share a set of resources and from time to time a process tries to atomically acquire a subset of size s of those resources. This is the typical behavior of a well designed distributed data structure. For lack of space we show only the benchmark which has n processes atomically increment $5000/n$ times with $s = 2$ locations chosen uniformly at random from a vector of length 60. The benchmark captures the behavior of highly concurrent queue and counter implementations as in [24, 25].

Priority Queue A shared priority queue on a heap of size n. We used a variant of a sequential heap implementation [11]. In this benchmark each of the n processes consequently enqueues a random value in a heap and dequeues the greatest value from it $5000/n$ times. The heap is initially empty and its maximal size is n. This is probably the most trying benchmark since there is no potential for concurrency and the size of the data structure increases with n.

Doubly Linked Queue An implementation of a queue as a doubly linked list in an array. The first two cells of the array contain the *head* and the *tail* of the list. Every item in the list is a couple of cells in the array, which represent the index of the previous and next element respectively. Each process enqueues a new item by updating *tail* to contain the new item's index and dequeues an item by updating the *head* to contain the index of the next item in the list. Each process executes $5000/n$ couples of enqueue/dequeue operations on a queue of initial size n. This benchmark supports limited parallelism since when the queue is not empty, enqueues/dequeues update the tail/head of the queue without interfering each other. For a high number of processes, the size of the updated locations in each enqueue/dequeue is relatively small compared to the object size.

We implemented the k-word Compare&Swap transaction (given in Figure 2) as specialization of the general STM scheme given above. The simplification is that processes do not have to agree on the value stored in the data set before the transaction started, only on a boolean value which says if the value is equal to *old*[] or not.

We used the above benchmarks to compare STM to the two nonblocking software translation methods described earlier and a blocking *MCS queue-lock* [23] based solution (the data structure is accessed in a mutually exclusive manner). The non-blocking methods include *Herlihy's Method* and Israeli and Rappoport's k-word Compare&Swap based implementation of the *cooperative method*. All the non-blocking methods use exponential backoff [3] to reduce contention.

5.2 Results

The data to be presented leads us to conclude that there are three factors differentiating among the performance of the four methods:

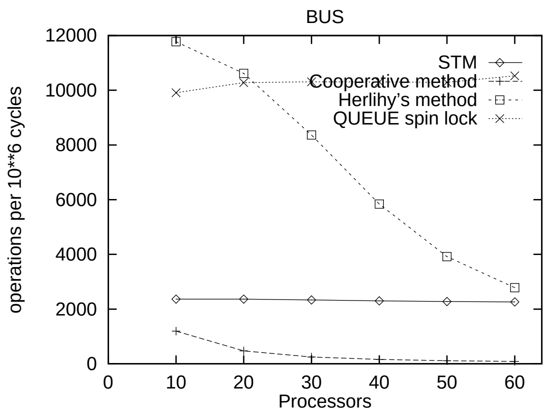
1. *Potential parallelism*: Both locking and Herlihy's method do not exploit potential parallelism and only one process at a time is allowed to update the data structure. The software-transactional and the cooperative methods allow concurrent processes to access disjoint parts of the data structure.

2. *The price of a failing update*: In Herlihy's lock-free method, the number of memory accesses of a failing update in is at least the size of the object (reading the object and copying it to the private copy, and reading and writing to the pointer). Fortunately, the nature of the cache coherence protocols is such that almost all accesses performed when the process updates its private copy are local. In both caching methods, the price of a failure is a least the number locations accessed during the cached execution.

3. *The amount of helping by other processes*: Helping exists only in the software-transactional and the cooperative methods. In the cooperative implementation,

[2]Naturally this operation is less efficient than the theoretical Load_Linked/Store_Conditional proposed in [6, 15, 18] (which we could have built directly into Proteus), since a failing Compare&Swap will cost a memory access while a failing Store_Conditional wont. However, we believe the 64-bit Compare&Swap is closer to the real world then the theoretical Load_Linked/Store_Conditional since existing implementations of Load_Linked/Store_Conditional as on Alpha [12] or PowerPC [17] do not allow access to the shared memory between the Load_Linked and the Store_Conditional operations.

[3]The non-blocking property will be achieved only if the number of spurious failures is finite.



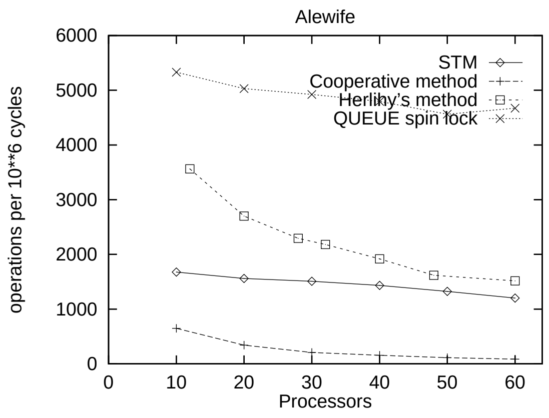


Figure 6: Counting Benchmark

k-word Compare&Swap , including failing ones, are helped not only by the k-word Compare&Swap operations that access the same locations concurrently, but also by all the operations that are in turn helping them and so on... In the STM method, an k-word Compare&Swap is helped only by operations that need the same locations. Moreover, and this is a crucial performance factor, in STM most of the unsuccessful updates terminate as *failing* transactions, not as failing k-word Compare&Swap , and when a transaction fails on the first location, it is not helped.

The results for the counting benchmark are given in Figure 6. The horizontal axis shows the number of processors and the vertical axis shows the throughput achieved. This benchmark is cruel to the caching based methods, since the amount of updated memory is equivalent to the size of the object and there is no potential for parallelism . On the bus architecture, locking and Herlihy's method give significantly higher throughput than the caching methods.

In the resource allocation benchmark Figure 7, as the number of processors increases, local work can be performed concurrently, and thus the performance of the STM improves. On the bus, beyond a certain number of processors, the potential for parallelism declines, causing a growing number of k-word Compare&Swap conflicts, and the throughput degrades.

A priority queue is a data structure that does not allow concurrency, and as the number of processors increases, the number of locations accessed increases too. Still, the number of accessed locations is smaller than the size of the object. Therefore, the STM performs better than Herlihy's method in most concurrency level.

Figure 9 contains the doubly linked queue results. There is more concurrency in accessing the object than in the counter benchmark, though it is limited: at most two processes may concurrently update the queue. Herlihy's method performs poorly because the penalty paid for a failed update grows linearly with queue size: usually twice the number of the processes. In the STM method, the low granularity of the two-word Compare&Swap transactions implies that the price of a failure remains constant in all concurrency levels, though local work is still higher than the `Test-and-Test-and-Set`.

5.3 A comparison of non-blocking methods only

Every theoretical method can be improved in many ways when implemented in practice. In order to get a fair comparison between the non-blocking methods one should use them in their *purest* form. Therefore, we compare the performance of all the non-blocking methods without backoff (in all the methods) and without the non-redundant-helping policy (in STM). We also compare the cooperative k-word Compare&Swap with STM for a specific implementation which explicitly needs such a software supported operation. We chose Israeli and Rappoport's algorithm for a concurrent priority queue [18], since it is based on recursive helping. Therefore, whenever a process during the execution of a k-word Compare&Swap helps another remote disjoint process, it should give an advantage to Israeli and Rappoport method. Our implementation is slightly different since it uses a 3-word Compare&Swap operation instead of a 2-word Store-Conditional operation [4].

We ran the same benchmark as for the regular priority queue. The results of the concurrent priority queue benchmark are given in Figure 10. In spite of the advantage that the inherent structure of the algorithm should give to Israeli and Rappoport method, STM provides the highest throughput. As in the counter and the sequential priority queue benchmarks, the reason for this is the high number of failing k-word Compare&Swap operations in Israeli and Rappoport method: up to 2.5 times the number of successful k-word Compare&Swap .

We summarize the highlights of the other pure benchmarks in Table 1, where entries are the throughput ratio of $\frac{STM}{Other Method}$. As can be seen, STM outperforms the cooperative method in all benchmarks and outperforms Herlihy's in all except for the counter benchmark.

[4] In fact, using 3-word Compare&Swap simplifies the implementation since it avoids *freezing* [18] nodes

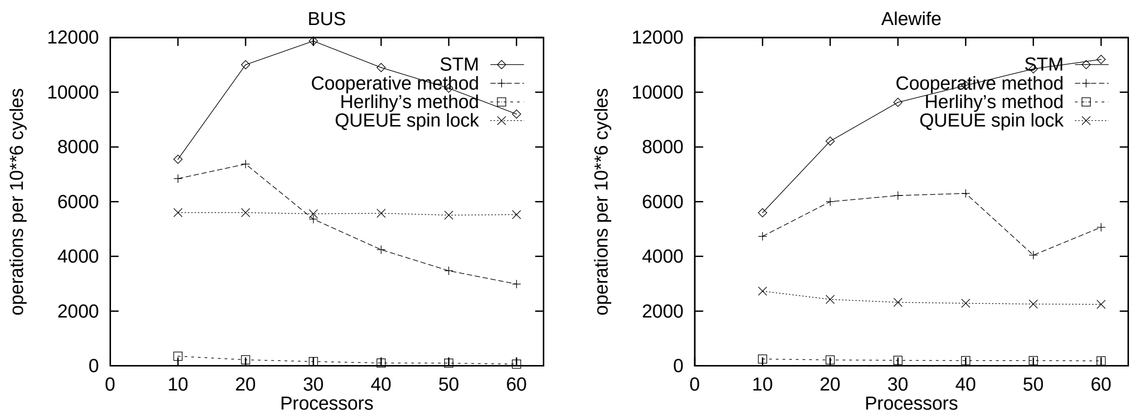


Figure 7: Resource Allocation Benchmark

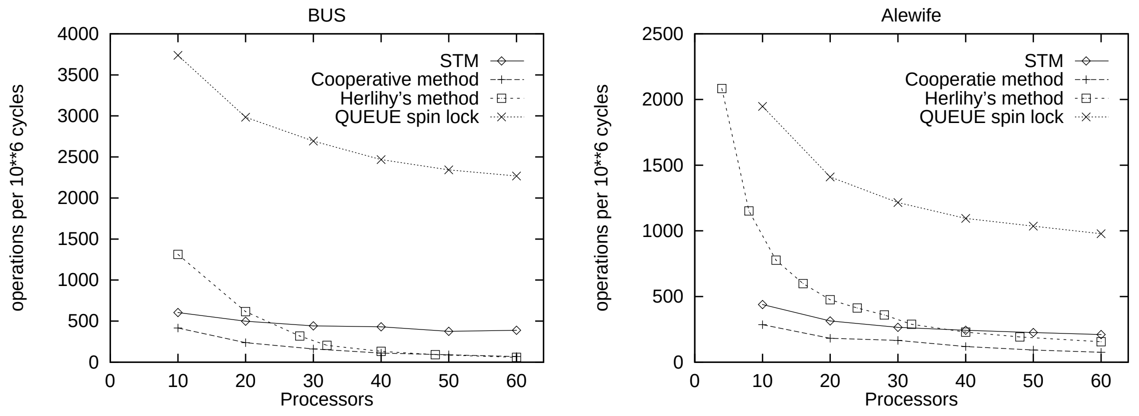


Figure 8: Priority Queue Benchmark

6 Acknowledgments

We wish to thank Greg Barnes and Maurice Herlihy for their many helpful comments.

References

[1] A. Agarwal et al. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors.* Kluwer Academic Publishers, 1991. An extended version of this paper has been submitted for publication, and appears as MIT/LCS Memo TM-454, 1991.

[2] R. J. Anderson. Primitives for Asynchronous List Compression. *Proceeding of the 4th ACM Symposium on Parallel Algorithms and Architectures,* pages 199-208, 1992.

[3] T.E. Anderson. The performance of spin lock alternatives for shared memory multiprocessors. In *IEEE Transaction on Parallel and Distributed Systems,* 1(1):6-16, January 1990.

[4] J. Alemany, E.W. Felten Performance Issues in Non-Blocking Synchronization on Shared-Memory Multiprocessors. In *Proceedings of 11th ACM Symposium on Principles of Distributed Computation, Pages 125-134* August 1992.

[5] J. Aspnes, M.P. Herlihy, and N. Shavit. Counting Networks. *Journal of the ACM*, Vol. 41, No. 5 (September 1994), pp. 1020-1048.

[6] G. Barnes A Method for Implementing Lock-Free Shared Data Structures In *Proceedings of the 5th ACM Symposium on Parallel Algorithms and Architectures* 1993.

[7] B.N Bershad. Practical consideration for lock-free concurrent objects. Technical Report, CMU-CS-91-183, Carnegie Mellon University. September 1991.

[8] E.A. Brewer C.N. Dellarocas, A. Colbrook, and W. E. Weihl. Proteus: A High-Performance Parallel-

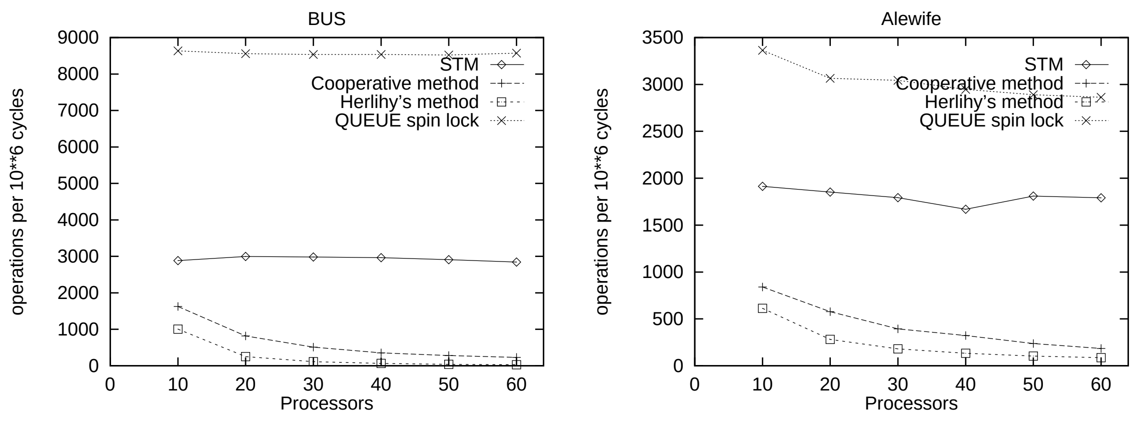


Figure 9: Doubly Linked Queue Benchmark

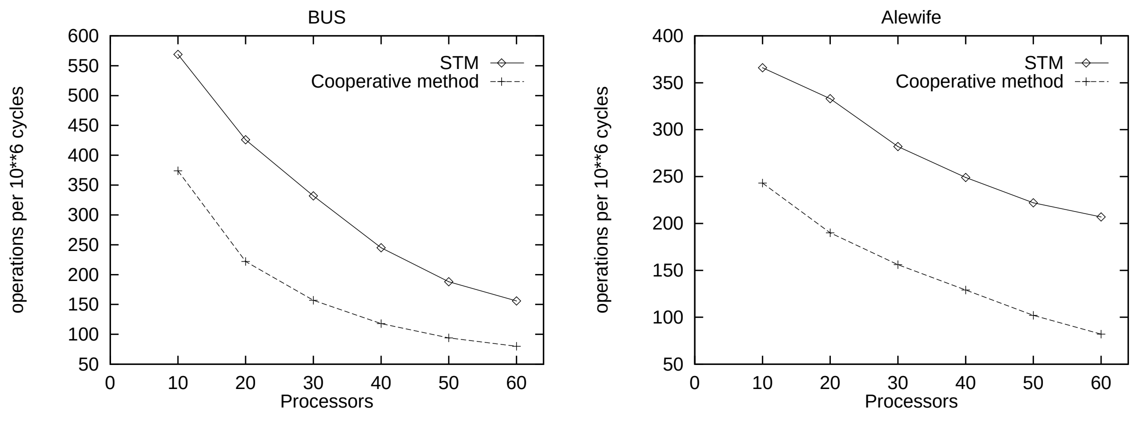


Figure 10: Non-blocking implementations of Israeli & Rappoport's Priority Queue

Architecture Simulator. MIT/LCS/TR-516. September 1989.

[9] E.A. Brewer C.N. Dellarocas. Proteus. User Documentation.

[10] K. Chandy and J. Misra. The Drinking Philosophers Problem. In*ACM Transaction on Programming Languages and Systems,* 6(4):632-646, October 1984.

[11] T.H. Cormen, C.E. Leiserson and R.L. Rivest. Introduction to algorithms. MIT Press.

[12] DEC. Alpha system reference manual.

[13] M. Herlihy and J.M. Wing. Linearizability: A correctness condition for concurrent objects. In *ACM Transaction on Programming Languages and Systems,* 12(3), pages 463-492, July 1990.

[14] M. Herlihy. Wait-Free Synchronization. In *ACM Transaction on Programming Languages and Systems,* 13(1), pages 124-149, January 1991.

[15] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(9): 745–770, November 1993.

[16] M. Herlihy and J.E B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *20th Annual Symposium on Computer Architecture, pages 289-300*, May 1993.

[17] IBM. Power PC. Reference manual.

[18] A. Israeli and L. Rappoport. Efficient Wait Free Implementation of a Concurrent Priority Queue. In *WDAG 1993. Lecture Notes in Computer Science 725, Springer Verlag, pages 1-17.*

[19] A. Israeli and L. Rappoport. Disjoint-Access-Parallel Implementations of Strong Shared Memory *Proc. of the 13th ACM Symposium on Principles of Distributed Computing pages 151-160.*

[20] A. LaMarca. A Performance Evaluation of Lock-Free Synchronization Protocols. *Proc. of the 13th ACM Sym-*

		10 processors		60 processors	
Throughput ratio of	*STM/other*	Herlihy's method	Cooperative method	Herlihy's method	Cooperative method
Counter	Bus	0.34	1.98	0.74	8.44
	Alewife	0.30	1.92	0.45	7.6
Doubly linked queue	Bus	6.07	1.44	58.9	3.36
	Alewife	2.44	1.75	12.9	7.28
Resource Allocation	Bus	22.5	1.09	85.61	1.69
	Alewife	24.14	1.12	59.8	2.35
Priority queue	BUS	0.42	1.26	2.8	2.16
	Alewife	0.41	1.27	1.1	2.24

Table 1: Pure implementation throughput ratio: STM / other methods

posium on Principles of Distributed Computing, pages 130-140.

[21] N. Lynch and M. Tuttle. Hierachical Correctness Proofs for Distributed Algorithm. In *Proceedings of 6th ACM Symposium on Principles of Distributed Computation, Pages 137-151* August 1987.

[22] H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91. Columbia University. Mars 1991.

[23] J.M. Mellor-Crummey and M.L. Scott Synchronization without Contention. In *Proceedings of the 4th International Conference on Architecture Support for Programming Languages and Operating Systems,* April 1991.

[24] L. Rudolph, M. Slivkin, and E. Upfal. A Simple Load Balancing Scheme for Task Allocation in Parallel Machines. In *Proceedings of the 3rd ACM Symposium on Parallel Algorithms and Architectures*, pages 237–245, July 1991.

[25] N. Shavit and A. Zemach. Diffracting Trees. In *Proceedings of the Annual Symposium on Parallel Algorithms and Architectures (SPAA)*, June 1994.

[26] J. Turek D. Shasha and S. Prakash. Locking without blocking: Making Lock Based Concurrent Data Structure Algorithms Non-blocking. In *Proceedings of the 1992 Principle of Database Systems pages 212-222.*

[27] D. Touitou. Lock-Free Programming: A Thesis Proposal. Tel Aviv University April 1993.