Software Transactional Memory

Nir Shavit*
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 non-blocking 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 non-blocking 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.

*Contact Author: E-mail: shanir@theory.lcs.mit.edu

†A preliminary version of this paper appeared in the 14th ACM Symposium on the Principles of Distributed Computing, Ottowa, Ontario, Canada, 1995

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, 16, 15]. 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 [24] 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 [20] 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 [17] 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 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-Non-Blocking 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, 28]. 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 [16] (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 [22] 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 [28]. 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, after executing the operation needed, releasing the locks. The key to achieving the non-blocking resilient behavior in the caching approach of [6, 28] 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 [21] 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 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 techniques are inferior to standard *non-resilient* lock-based methods such as queue-locks [25]. 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 [17]. 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*.

```
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

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. The basic correctness requirement for a STM implementation is *linearizability* [14]: every concurrent history is "equivalent" to some legal sequential history which which is consistent with the real-time order induced by the concurrent history.

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 an atomic procedure which gets as parameters the data set and a deterministic transition function which determines the new values to be stored in the data set. This procedure updates the memory and returns the previous value stored. 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 [17] 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).

2.1 The System Model

Our computation model follows Herlihy and Wing [14] and can also be cast in terms of the I/O automata model of Lynch and Tuttle [23]. A concurrent *system* consists of a collection of *processes*. Processes communicate through shared data structures called *objects*. Each object has a set of primitive *operations* that provide the only means to manipulate that object. Each process is a sequential thread of control [14] which applies a sequence of operations to objects by issuing an invocation and receiving the associated response. A *history* is a sequence of invocations and responses of some system execution. Each history induces a "real-time" order of operations ($\rightarrow$) where an operation A precedes another operation B if A's response occurs before B's invocation. Two operations are *concurrent* if they are unrelated by the real-time order. A sequential history is a history in which each invocation is followed immediately by its corresponding response. The *sequential specification* of an object is the set of *legal* sequential histories associated with it. The basic correctness requirement for a concurrent implementation is *linearizability* [14]: every concurrent history is "equivalent" to some legal sequential history which which is consistent with the partial real-time order induced by the concurrent history. In a linearizable implementation, operations appear to take effect atomically at some point between their invocation and response. In our model, every shared memory location L of a multiprocessor machine's memory is formally modeled as an object which provides every processor $i = 1 \ldots n$ four types of possible operations, with the

following sequential specification:

$Write^i(L, v)$ writes the value v to location L.

$Read^i(L)_i$ reads location L and returns its value v.

$Load\_Linked^i(L)$ reads location L and returns its value v. Marks location L as "read by i."

$Store\_Conditional^i(L, v)$ if location L is marked as "read by i," the operation writes the value v to L, erases all existing marks by other processors on L and returns a *success* status. Otherwise returns a *failure* status.

The a more detailed formal specification of these operation can be found in [15, 16].

2.2 A Sequential Specification of STM

The following is the sequential specification of STM. Let $L \subseteq N$ be a set of locations. A *memory state* is a function $s : L \mapsto V$ which returns for each location of L a value from some set V. Let S be the set of all possible memory states. A *transition* function $t : S \mapsto S$, is a *computable* function which gets as a parameter a state and returns a new state. Given a subset $ds \subseteq L$, we say that a transition function t is *ds dependent*, if the following conditions hold: (a) for every state s and every location l, if $l \notin ds$ then $s(l) = t(s)(l)$ (b) if s_1 and s_2 are two states s.t. for every $l \in ds$, $s_1(l) = s_2(l)$, then for every $l \in ds$ $t(s_1)(l) = t(s_2)(l)$.

Given a set L of locations, a *Static Transactional Memory* over L is a concurrent object which provides every process i with a *Tran_i(DataSet, f, r, status)* operation. Its has as input *DataSet* – a subset of L, and f – a transition function which is *DataSet* dependent. It returns a function $r : DataSet \mapsto V$ and a boolean value *status*. We omit the subscript of a $Tran$ operation when the id of the processor performing the operation is unimportant.

Let $h = o_1 o_2 o_3 \ldots$ be a finite or infinite sequential history where o_i is the ith operation executed. For every finite prefix $h^m = o_1 o_2 o_3 \ldots o_m$ of h, we define the *terminating* state of h^m, $TS(h^m)$ in the following inductive way: If $m = 0$ then $TS(h^m) = e$ where e is the function $e(l) = \emptyset$ for every $l \in L$. If $m > 0$ then assume w.l.o.g. that $o_m = Tran(DS, f, r, status)$ and let $h^{m-1} = o_1 o_2 o_3 \ldots o_{m-1}$. If $status = success$ then $TS(h^m) = f(TS(h^{m-1}))$ otherwise $TS(h^m) = TS(h^{m-1})$

We can now proceed to define the sequential specification of the static transactional memory. Given a function $f : A \mapsto B$ and $A' \subseteq A$, we define the *restriction* of f on A' (denoted $f \Uparrow A'$) to be the function $f' : A' \mapsto B$ s.t. $\forall a \in A' f'(a) = f(a)$. We require that a *correct* implementation of an STM object meet the following sequential specification:

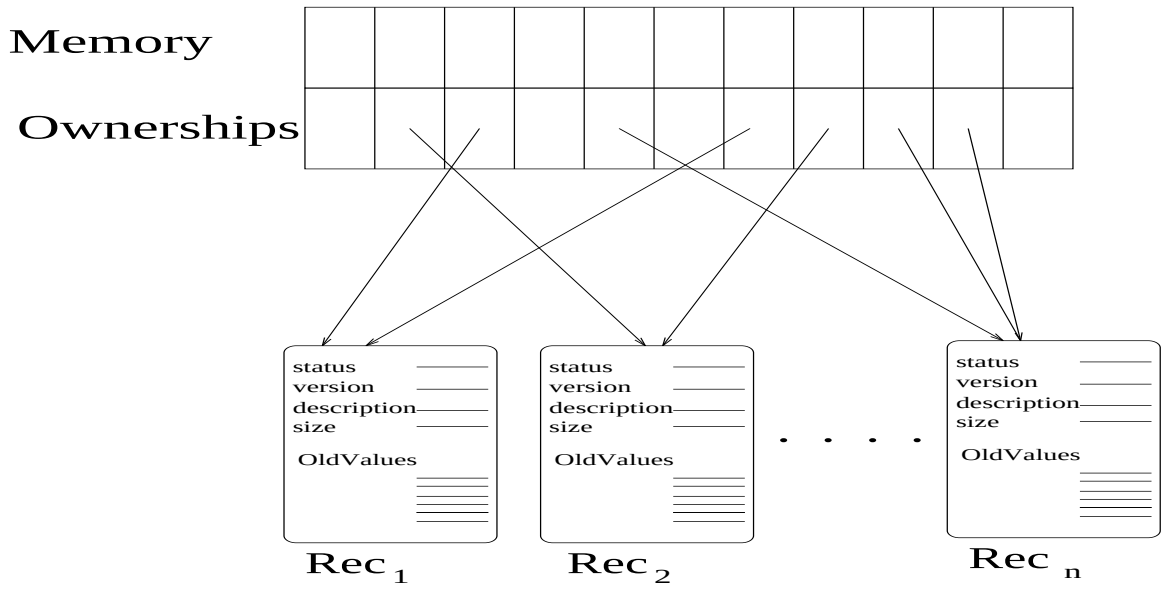

Figure 3: STM implementation: shared data structures

Definition 2.1 *The set of sequential histories, such that for each finite or infinite history* $h = o_1 o_2 o_3 \ldots$*, it is the case that for all* k*, if* $o_k = Tran(DataSet, f, r, status)$ *and* $status = success$ *then* $r = TS(o_1 o_2 o_3 \ldots o_{k-1}) \Uparrow DataSet$.

3 A Non-Blocking Implementation of STM

We implement a non-blocking static STM of size M using the following data structures (See Figure 3):

- $Memory[M]$, a vector which contains the data stored in the transactional memory.
- $Ownerships[M]$, a vector which determines for any cell in *Memory*, which transaction owns it.

Each process i keeps in the shared memory a record, pointed to by Rec_i, that will be used to store information on the current transaction it initiated. It has the following fields: *Size* which contains the size of the data set. *Add*[] – a vector which contains the data set addresses in increasing order. *OldValues*[] a vector whose cells are initialized to *Null* at the beginning of every transaction. In case of a successful transaction this vector will contain the former values stored in the involved

```
StartTransaction(DataSet)
    Initialize(Rec_i,DataSet)
    Rec_i ↑.stable = True
    Transaction(Rec_i,Rec_i ↑.version,True)
    Rec_i ↑.stable = False
    Rec_i ↑.version++
    if Rec_i↑.status = Success then
        return (Success,Rec_i ↑.OldValues)
    else
        return Failure
```

Figure 4: StartTransaction

locations. 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* is an integer, initially 0, which determines the instance number of the transaction. This field is incremented every time the process terminates a transaction.

A process i initiates the execution of a transaction by calling the *Transaction* routine of Figure 4. *Transaction* first initializes the process's record and then declares the record as *stable*, ensuring that any processors helping the transaction complete will read a consistent description of the transaction. After executing the transaction the process checks if the transaction has succeeded, and if so returns the content of the vector *OldValues*.

The procedure *Transaction* (Figure 5), gets as parameters *Rec*, 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 helped transaction's record is in a *stable* state.

[1]The use of this unbounded field can be avoided if an additional *Validate* operation is available [20, 21].

```
Transaction(rec,version,IsInitiator)
    AcquireOwnerships(rec,version)
    (status,failadd) = LL(rec↑.status)
    if status = Null then
        if (version ≠ rec↑.version) then return
        SC(rec↑.status,(Success,0))
    (status,failadd) = LL(rec↑.status)
    if status = Success then
        AgreeOldValues(rec,version)
        NewValues = CalcNewValues(rec↑.OldValues)
        UpdateMemory(rec,version,NewValues)
        ReleaseOwnerships(rec,version)
    else
        ReleaseOwnerships(rec,version)
        if IsInitiator then
            failtran= Ownerships[failadd]
            if failtran = Nobody then
                return
            else
                failversion = failtran↑.version
                if failtran↑.stable
                    Transaction(failtran,failversion,False)
```

Figure 5: Transaction

```
AcquireOwnerships(rec,version)
    transize = rec↑.size
    for j = 1 to size do
        while true do
            location = rec↑.add[j]
            if LL(rec↑.status) ≠ Null then return
            owner = LL (Ownerships[rec↑.Add[j]])
            if rec↑.version ≠ version return
            if owner = rec then exit while loop
            if owner = Nobody then
                if SC(rec↑.status, (Null , 0) ) then
                    if SC(Ownerships[location],rec) then
                        exit while loop
            else
                if SC(rec↑.status, (Failure,j) ) then
                    return

ReleaseOwnerships(rec,version)
    size = rec↑.size
    for j = 1 to size do
        location= rec↑.Add[j]
        if LL(Ownerships[location]) = rec then
            if rec↑.version ≠ version then return
            SC(Ownerships[location],Nobody)

AgreeOldValues(rec,version)
    size = rec↑.size
    for j = 1 to size do
        location= rec↑.Add[j]
        if LL(rec↑.OldValues[location]) ≠ Null then
            if rec↑.version ≠ version then return
            SC(rec↑.OldValues[location],Memory[location])

UpdateMemory(rec,version,newvalues)
    size = rec↑.size
    for j = 1 to size do
        location= rec↑.Add[j]
        oldvalue= LL(Memory[location])
        if rec↑.AllWritten then return
        if version ≠ rec↑.version then return
        if oldvalue≠ newvalues[j] then
            SC(Memory[location],newvalues[j])
    if (not LL(rec↑.AllWritten)) then
        if version ≠ rec↑.version then return
        SC(rec↑.AllWritten,True)
```

Figure 6: *Ownerships* and *Memory* access

Since *AcquireOwnerships* of Figure 6 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 while it was owned by a different transaction, to set the status to *Failure*.

When writing the new values to the *UpdateMemory* as in Figure 6, 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

Given a run (we freely interchange between *run* and *history*) of the STM implementation, the nth transaction execution of process i is marked as $T(i,n)$. The transaction record for $T(i,n)$ is denoted as R_i, and by definition only process i updates R_i. It is thus clear that the number n in $T(i,n)$ is equal to the content of $R_i\uparrow.version$ during $T(i,n)$'s execution. The *executing* processes of $T(i,n)$ consist of process i, called the *initiator*, and the *helping* processes, those executing *Transaction* with parameters (R_i,n,False).

The following are the definitions of the register operations, where the superscript of an operation marks the id of the process which executed it, and the subscript marks the transaction instance that the process executes. Sometimes, when subscript and superscript are not needed we will omit them.

W_T^i(*variable,value*) Process i performs a *Write* operation on *variable* with *value* while executing transaction T.

R_T^i(*variable,value*) Process i performs a *Read* operation on *variable* which returns *value* while executing transaction T.

LL_T^i(*variable,value*) Process i deforms is a *Load_Linked* operation on *variable* which returns *value* while executing T.

SC_T^i(*variable,value*) Process i performs a *successful Store_Conditional* operation on *variable* with *value* while executing T.

$R^i_T(\Phi(variable))$ is a short form for $R^i_T(variable, value) \wedge \Phi(value)$ for some predicate Φ.

Clearly, any implementation of transactional memory which is based on an ownerships policy only, without helping, will satisfy the linearizability requirement: if a single process is able to lock all the needed memory locations it will be able to update the memory atomically. Consequently, in order to prove the linerizability of our implementation, we will have mainly to show that the fact that many processes may execute the same transaction will behave as if they were a single process running alone. In the following proof we will first show that all the executing processes of a transaction perform the same transaction that the initiator intended. Then, we will prove that all the executing processes agree on the final status of the transaction. Finally, we will demonstrate that the executing process of a successful transaction will update the memory correctly.

The non-blocking property of the implementation will be established by showing first that no executing process will ever be able to acquire an ownership after the transaction has failed, and then showing that since locations are acquired in increasing order, some transaction will eventually succeed.

4.1 Linearizability

We first show, that although process i uses the same record for all its transactions and may eventually change it while some executing process reads its content, all the executing processes of a transaction read a consistent description of what is is supposed to do.

Claim 4.1 *Given an execution r of the STM implementation, the helping processes of a transaction $T(i,n)$ in r read the same data set vector which was stored by i. Any executing process of $T(i,n)$ which read a different data set will not update any of the the shared data structures.*

Proof: Assume by way of contradiction that there is a helping process j of $T(i,n)$ which read a different description of the transaction. That means that for some location a from $T(i,n)$'s description, $R^j_{T(i,n)}(a,x)$ and $W^i_{T(i,n)}(a,y)$ but $x \neq y$. By the algorithm, only process i updates the description fields in Rec_i and it does it only once per transaction. Assume first that $W^i_{T(i,n)}(a,y) \rightarrow R^j_{T(i,n)}(a,x)$. Since $x \neq y$, there is some write operation $W^i_{T(i,n')}(a,x)$ s.t:

$$W^i_{T(i,n)}(a,y) \rightarrow W^i_{T(i,n')}(a,x) \rightarrow R^j_{T(i,n)}(a,x)$$

where $n' > n$. Since

$$W^i_{T(i,n)}(Rec_i\uparrow.version, n+1) \rightarrow W^i_{T(i,n')}(a,x) \rightarrow R^j_{T(i,n)}(a,x)$$

and all the helping processes of $T(n,i)$ compare between n and $Rec_i\uparrow.version$ before executing a SC operation, j will not, ¿from this point on, update any shared data structure. Assume that $R^j_{T(i,n)}(a,x) \rightarrow W^i_{T(i,n)}(a,y)$. By the algorithm (lines 19,20 in the Transaction procedure),

$$R^j_{T(i,n)}(Rec_i\uparrow.version, n) \rightarrow R^j_{T(i,n)}(Rec_i\uparrow.stable, true) \rightarrow R^j_{T(i,n)}(a,x),$$

and in that case

$$W^i_{T(i,n-1)}(Rec_i\uparrow.version, n) \rightarrow W^i_{T(i,n)}(Rec_i\uparrow.stable, true) \rightarrow W^j_{T(i,n)}(a,y)$$

which is a contradiction to the description of the *StartTransaction* procedure. ■

Next we show that all the executing processes of a transaction agree on its terminating status.

Claim 4.2 *Assume that i and j are two executing processes of some transaction $T(i,n)$. If i and j read different values of the terminating status (line 6 in Transaction procedure), at least one of them will henceforth not update the shared data structures.*

Proof: Assume by way of contradiction that

$$R^k_{T(i,n)}(Rec_i\uparrow.status, Failure) \text{ and } R^j_{T(i,n)}(Rec_i\uparrow.status, Success),$$

and assume w.l.o.g. that

$$R^k_{T(i,n)}(Rec_i\uparrow.status, Failure) \rightarrow R^j_{T(i,n)}(Rec_i\uparrow.status, Success).$$

In that case, there is some process z such that

$$R^k_{T(i,n)}(Rec_i\uparrow.status, Failure) \rightarrow LL^k(Rec_i\uparrow.status, Null) \rightarrow$$
$$SC^k(Rec_i\uparrow.status, Success) \rightarrow R^j_{T(i,n)}(Rec_i\uparrow.status, Success).$$

Since i is the only process which initializes $Rec_i\uparrow.status$, it follows that

$$R^k_{T(i,n)}(Rec_i\uparrow.status, Failure) \rightarrow W^z(Rec_i\uparrow.status, Null) \rightarrow$$
$$SC^z(Rec_i\uparrow.status, Success) \rightarrow R^j_{T(i,n)}(Rec_i\uparrow.status, Success).$$

By the algorithm

$$R^k_{T(i,n)}(Rec_i\uparrow.version, n) \rightarrow R^k_{T(i,n)}(Rec_i\uparrow.stable, true) \rightarrow R^k_{T(i,n)}(Rec_i\uparrow.status, Failure)$$

and

$$W^i(Rec_i\uparrow.version, n+1) \rightarrow W^i(Rec_i\uparrow.status, Null)$$

we may therefore conclude that process j will not update the shared data structures anymore after executing $R^j_{T(i,n)}(Rec_i\uparrow.status, Success)$. ■

Thanks to Claim 4.2, we can now define a transaction as *successful* if its terminating status is *Success* and *failing* otherwise. From the algorithm and Claim 4.2 it is clear that executing processes of failing transactions will never change the *Memory* data structure.

Claim 4.3 *Every successful transaction has:*

a *only one executing process which writes Success as the terminating status of the transaction*

b *only one executing process who sets the AllWritten field to true.*

Proof: Assume that during a successful transaction $T(i,n)$, one of those fields, f was updated by two executing processes k and j. Both have executed

$$R_{T(i,n)}(Rec_i\uparrow.version, n) \rightarrow LL_{T(i,n)}(f, Null) \rightarrow R_{T(i,n)}(Rec_i\uparrow.version, n)$$
$$\rightarrow W(Rec_i\uparrow.stable, True) \rightarrow LL(f, Null) \rightarrow SC_{T(i,n)}(f, v).$$

Assume w.l.o.g. that $SC^k_{T(i,n)}(f,v) \rightarrow SC^j_{T(i,n)}(f,v)$. By the specification of the *Load_Linked/Store_Conditional* operation,

$$LL^k_{T(i,n)}(f, Null) \rightarrow (SC^k_{T(i,n)}(f,v) \rightarrow LL^j_{T(i,n)}(f, Null) \rightarrow SC^j_{T(i,n)}(f,v)$$

But since only process i writes *Null* into field f, it follows that

$$W^i(Rec_i\uparrow.stable, False) \rightarrow W^i(Rec_i\uparrow.version, n) \rightarrow W^i_{T(i,n)}(f, Null).$$

Process j thus read $Rec_i\uparrow.stable$ as $false$ and therefore should not have helped $T(i,n)$. ■

For any successful transaction $T(i,n)$ let $SU(n,i)$ be the SC operation which has set $T(i,n)$'s status to Success and let $AW(n,i)$ be the SC operation which has set the *AllWritten* field to True. By the above claims those operations are well defined. The following lemma shows that successful transactions access the memory atomically.

Lemma 4.4 *For every n and every process i, if $T(i,n)$ is a successful transaction, then:*

a *between $SU(n,i)$ and $AW(n,i)$ all the entries in the Ownerships vector from $T(i,n)$'s data set contain Rec_i, and*

b *at $W^i_{T(i,n)}(Rec_i\uparrow.version, n+1)$ no entry contains Rec_i.*

Proof: The proof is by joint induction on n. Assume that the properties hold for $n' < n$ and let us prove them for n.

To prove **a**, consider j, the process which executed $SU(n,i)$. By the algorithm, j has performed

$$R^j(Rec_i\uparrow.version, n) \rightarrow \phi^j_{Tran(i,n)}(Ownerships[x_1], Rec_i) \rightarrow \ldots$$
$$\rightarrow \phi^j_{Tran(i,n)}(Ownerships[x_l], Rec_i) \rightarrow SU(n,i)$$

where ϕ is either a SC or R operation, and $x_1 \ldots x_l$ are $Tran(i,n)$'s data set locations. Assume that for some location x_r at $SU(n,i)$, $Ownerships[x_j]$ differ from those of Rec_i. By the algorithm this may happen only if

$$\phi^j_{Tran(i,n)}(Ownerships[x_r], Rec_i) \rightarrow SC^k(Ownerships[x_r], Null).$$

Therefore, there is some process k executing *release_ownerships* during $Tran(i,n')$ for $n' < n$. More precisely, the following sequence of operations has occured:

$$LL^k_{Tran(i,n')}(Ownerships[x_r], Rec_i) \rightarrow R^k_{Tran(i,n')}(Rec_i\uparrow.version, n') \rightarrow W^i(Rec_i\uparrow.version, n)$$
$$\rightarrow R^j(Rec_i\uparrow.version, n) \rightarrow SC^k_{Tran(i,n')}(Ownerships[x_r], Null).$$

By the induction hypothesis on property **b**, at $W^i(Rec_i\uparrow.version, n)$, $Ownerships[x_r]$ differs from Rec_i and therefore the $SC^k_{Tran(i,n')}(Ownerships[x_r], Null)$ should have failed. A contradiction.

To prove **b**, note that from the algorithm it follows that process i has executed

$$\phi^i_{Tran(i,n)}(Rec_i\uparrow.status, Success) \rightarrow \phi^i_{Tran(i,n)}(Ownerships[x_1], Null) \rightarrow \ldots$$
$$\rightarrow \phi^j_{Tran(i,n)}(Ownerships[x_l], Null) \rightarrow W^i_{T(i,n)}(Rec_i\uparrow.version, n+1).$$

Assume by way of contradiction that at $W^i_{T(i,n)}(Rec_i\uparrow.version, n+1)$ for some location x_r where $Ownerships[x_r] = Rec_i$. By the induction hypothesis on property **b**, x_r belongs to $Tran_i$'s data set. Let k be the processor that has written Rec_i on $Ownerships[x_r]$. By the algorithm, k performed

$$LL^k(Rec_i\uparrow.status, Null) \rightarrow LL^k(Ownerships[x_r], Null) \rightarrow SC^k(Rec_i\uparrow.status, Null) \rightarrow$$
$$\phi^i_{Tran(i,n)}(Rec_i\uparrow.status, Success) \rightarrow SC^k(Ownerships[x_r], Rec_i).$$

By property a, at the point of exeuting $SC_{Tran(i,n)}(Rec_i\uparrow.status, Success)$, $Ownerships[x_j] = Rec_i$ and therefore $SC^k(Ownerships[x_r], Null)$ should have failed, a contradiction. ■

The following corollary will be useful when proving the non blocking property of the implementation. The proof is similar to the proof of part **b** in Lemma 4.4

Corollary 4.5 *Let $T(i,n)$ be a failing transaction then at the point of executing $W^i_{T(i,n)}(Rec_i\uparrow .version, n+1)$, no entry contains Rec_i.*

We can now complete the proof of linearizability. We define the *execution state* of the implementation at any point of the execution to be the function F s.t. $F(x) = Memory[x]$ for every $x \in L$.

Lemma 4.6 *Let $T(i,n)$ be a successful transaction, and let $F1$ and $F2$ be the execution states of $SU(i,n)$ and $AW(i,n)$ respectively. The following properties hold:*

a *At $AW(i,n)$, $Rec_i\uparrow.old\_values = F1 \Uparrow DataSet_{(i,n)}$.*

b *If $F1$ and $F2$ are the execution states of $SU(i,n)$ and $AW(i,n)$ respectively then $F2 \Uparrow DataSet_{(i,n)} = f_{(i,n)}(F1) \Uparrow DataSet_{(i,n)}$, where $f_{(i,n)}$ is the transition function of $T(i,n)$.*

c *After $AW(i,n)$ no process executing $T_{(i,n)}$ will update Memory.*

Proof: The proof is by joint induction on the length of the execution.

To prove **a**, let $F1$ be the execution state at $SU(i,n)$. Assume by way of contradiction that for some location $x \in DataSet_{i,n}$, $Rec_i\uparrow.old\_values[x] \neq F1(x)$. That means that $Memory[x]$ was changed between $SU(i,n)$ and the point in the execution in which $Rec_i\uparrow.old\_values[x]$ was set. Since, by the algorithm, all the executing processes of $T(n,i)$ update $Rec_i\uparrow.old\_values$ before updating the *Memory*, $Memory[x]$ was altered by an executing process of some other successful transaction $T(i',n')$. By Lemma 4.4, $AW(i',n') \uparrow SU(i,n)$ and therefore by the induction hypothesis on property **c**, we have a contradiction.

To prove **b**, assume by way of contradiction that at $AW(i,n)$ there is some location $x_r \in DataSet_{i,n}$ s.t $Memory[x_r] \neq f_{(i,n)}(F1)(x_r)$. Let j be the process which has executed $AW(i,n)$. By the algorithm, as a part of the *UpdateMemory* procedure, j performed either

$$R^j_{T(i,n)}(Memory[x_r], f_{(i,n)}(F1)(x_r)) \rightarrow AW(i,n)$$

or

$$R^j_{T(i,n)}(Memory[x_r] \neq f_{(i,n)}(F1)(x_r)) \rightarrow SC^j_{T(i,n)}(Memory[x_r] f_{(i,n)}(F1)(x_r)) \rightarrow AW(i,n).$$

Therefore, there is some process k which performed SC^k on $Memory[x_r]$ with a value different then $f_{(i,n)}(F1)(x_r)$ after $R^j(x_r,*)$ and before $AW(i,n)$. Assume w.l.o.g. that k is executing the transaction $Tran(i',n')$. If $i = i'$ then clearly $n' < n$ and by the induction hypothesis on property **c**, k's writing should have failed. Therefore $i' \neq i$. If $AW(i',n') \rightarrow SC^k$ then $AW(i',n') \rightarrow AW(i,n)$

and using the induction hypothesis on property **c** we again have a contradiction. Therefore $SC^k \rightarrow AW(i', n')$. In that case we have a contradiction to Lemma 4.4 since at SC^k, $Ownerships[x_r]$ is supposed to contain both Rec_i and $Rec_{i'}$.

To prove **c**, assume by way of contradiction that some executing process j of $T(i,n)$ updated a location x_r in memory after $AW(i,n)$. Process j performed the following sequence of operations:

$$LL^j_{Tran(i,n)}(Memory[x_r], val) \rightarrow R^j_{Tran(i,j)}(Rec_i\uparrow.AllWritten, False) \rightarrow$$
$$R^j_{Tran(i,j)}(Rec_i\uparrow.version, n) \rightarrow SC^j_{Tran(i,n)}(Memory[x_r], f_{(i,n)}(F1)(x_r))$$

where

$$val \neq f_{(i,n)}(F1)(x_r) \text{ and therefore } LL^j_{Tran(i,n)}(Memory[x_r], val) \rightarrow AW(i,n).$$

By property **a**, at $AW(i,n)$, $Memory[x_r]$ contains $f_{(i,n)}(F1)(x_r)$ and therefore $SC^j_{Tran(i,n)}(Memory[x_r], f_{(i,n)}(F1)(x_r))$ should have failed. ■

In order to prove that the implementation is linearizable, let us first consider executions of the STM implementation which contain successful transactions only. Let HS, be one of those executions and let $AW1 \rightarrow AW2 \rightarrow AW3 \ldots$ be the sequential subsequence of all the AW events that occured during HS. Since an AW event occurs only once for every successful transaction, let H be the sequence

$$T_{AW1}(DataSet_1, f_1, ov_1, success)\ T_{AW2}(DataSet_2, f_2, ov_2, success)\ T_{AW3}(DataSet_1, f_1, ov_1, success) \ldots$$

of transaction executions induced by the AW events, where for every T_{AWn} the triple $(DataSet_n, f_n, ov_n, success)$ represents the content of the *DataSet*, F, *old_values*, and *status* fields respectively in T_{AW_n}'s records at AW_n. By Lemma 4.6, it is a simple exercise to show by induction that H is a legal sequential history according to Definition 2.1. Since failing transactions do not cause any change to *Memory*, we may conclude that:

Theorem 4.7 *The implementation is linearizable.*

4.2 Non-blocking

We denote the executing process which wrote *Failure* to $Rec_i\uparrow.status$ of a transaction $T(i,n)$ as its *failing process*. In order to prove the non-blocking property of the implementation, The *failing location* of $T(i,n)$ is the location that the failing process has failed to acquire.

Claim 4.8 *Given a failing transaction $T(i,n)$, all the executing process of $T(i,n)$ will never acquire a location which is higher or equal to the failing location of $T(i,n)$.*

Proof: Assume by way of contradiction that some executing process of $T(i,n)$ acquired a location, higher or equal to $T(i,n)$'s failing location. Since the process saw all the smaller locations captured for $T(i,n)$, let j be executing process of $T(i,n)$ that captured the failing location x_r of $T(i,n)$. By the algorithm, j performed the following sequence of operations:

$$LL^j_{T(i,n)}(Rec_i\uparrow.status, Null) \rightarrow LL^j_{T(i,n)}(Ownerships[x_r], Nobody) \rightarrow$$
$$SC^j_{T(i,n)}(Rec_i\uparrow.status, Null) \rightarrow SC^j_{T(i,n)}(Ownerships[x_r], Rec_i).$$

The failing process of $T(i,n)$, k has performed the following sequence of operations:

$$LL^k_{T(i,n)}(Rec_i\uparrow.status, Null) \rightarrow LL^k_{T(i,n)}(Ownerships[x_r], other)$$
$$\rightarrow SC^k_{T(i,n)}(Rec_i\uparrow.status, Failure).$$

where *other* is neither *Null* nor Rec_i. Assume that

$$SC^k_{T(i,n)}(Rec_i\uparrow.status, Failure) \rightarrow SC^j_{T(i,n)}(Ownerships[x_r], Rec_i).$$

In that case

$$SC^j_{T(i,n)}(Rec_i\uparrow.status, Null) \rightarrow LL^k_{T(i,n)}(Rec_i\uparrow.status, Null) \rightarrow$$
$$LL^k_{T(i,n)}(Ownerships[x_r], other) \rightarrow SC^k_{T(i,n)}(Rec_i\uparrow.status, Failure)$$

and consequently k must have seen $Ownerships[x_r]$ already owned by i or the $SC^j_{T(i,n)}(Ownerships[x_r], Rec_i)$ should have failed. Therefore $SC^j_{T(i,n)}(Ownerships[x_r], Rec_i) \rightarrow SC^k_{T(i,n)}(Rec_i\uparrow.status, Failure)$. Now, if $SC^j_{T(i,n)}(Ownerships[x_r], Rec_i) \rightarrow LL^k_{T(i,n)}(Ownerships[x_r], other)$ we have a contradiction since a process executing $T(i,n)$ never releases an ownerships before the status was set and processes executing $T(i,n'), n' < n$ will see that the $Rec_i\uparrow.version$ has changed. For that reason,

$$LL^k_{T(i,n)}(Ownerships[x_r], other) \rightarrow SC^k_{T(i,n)}(Rec_i\uparrow.status, Failure)$$
$$\rightarrow SC^j_{T(i,n)}(Ownerships[x_r], Rec_i).$$

In that case

$$LL^k_{T(i,n)}(Rec_i\uparrow.status, Null) \rightarrow LL^k_{T(i,n)}(Ownerships[x_r], other) \rightarrow$$
$$LL^j_{T(i,n)}(Ownerships[x_r], Nobody) \rightarrow SC^j_{T(i,n)}(Rec_i\uparrow.status, Null)$$
$$\rightarrow SC^j_{T(i,n)}(Ownerships[x_r], Rec_i)$$

and $SC^k_{T(i,n)}(Rec_i\uparrow.status, Failure)$ must have failed- a contradiction. ■

Theorem 4.9 *The implementation is non-blocking.*

Proof: Assume by way of contradiction that there is an infinite schedule in which no transaction terminates successfully. Assume that the number of failing transaction 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 get ownership on the same location for the same transaction. But in that case at least one process will succeed or will fail the transaction, a contradiction. It must thus be the case that the number of failing transactions is infinite. In that case, there is at least one location which is a failing address infinitely often. Consider A, the highest of those addresses. Since the initiator of the transaction tries to help the transaction which has failed him before retrying, and since by Corollary 4.5 all acquired locations are released before helping, it follows that there are infinitely many transactions which have acquired ownership on A but have failed. By Claim 4.8 those transactions have failed on addresses higher than A, a contradiction to the fact that A is the highest failed location. ■

To avoid major overheads when no *Failure*s 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]. Proteus simulates parallel code by multiplexing several parallel threads on a single CPU. Each thread runs on its own virtual CPU with accompanying local memory, cache and communications hardware, keeping track of how much time is spent using each component. In order to facilitate fast simulations, Proteus does not do complete hardware simulations. Instead, operations which are local (do not interact with the parallel environment) are run uninterrupted on the simulating machine's CPU and memory. The amount of time used for local calculations is added to the time spent performing (simulated) globally visible operations to derive each thread's notion of the current time. Proteus makes sure a thread can only see global events within the scope of its local time.

In the simulated bus architecture processors communicate with shared memory modules through a common bus. Uniform shared-memory access is assumed, that is, access of any memory module from any processor takes the same amount of time which is 4 cycles (ignoring delays due to bus contention). Each processor has a cache with 2048 lines of 8 bytes and the cache coherence is maintained using Goodman's [18] "snoopy" cache-coherence protocol.

The simulated network architecture is similar to that of the Alewife cache-coherent distributed-memory machine currently under development at MIT [1]. Each node of the machines Torus shaped communication grid consists of a processor, cache memory, a router, and a portion of the globally-addressable memory. The cost of switching or wiring in the Alewife architecture was 1 cycle/packet. As for the bus architecture, each processor has a cache with 2048 lines of 8 bytes. The cache coherence is provided using a using a version of Chaiken's [12] directory-based cache-coherence protocol.

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. Naturally this operation is less efficient than the theoretical *Load_Linked/Store_Conditional* proposed in [6, 16, 20] (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 [13] or PowerPC [19] do not allow access to the shared memory between the *Load_Linked* and the *Store_Conditional* operations. 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[2] [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 $s = 2, 4, 6$ 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 [26, 27].

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

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* [25] 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 non-blocking method, the number of memory accesses of a failing update in is at least the size of the object (reading the object and copying

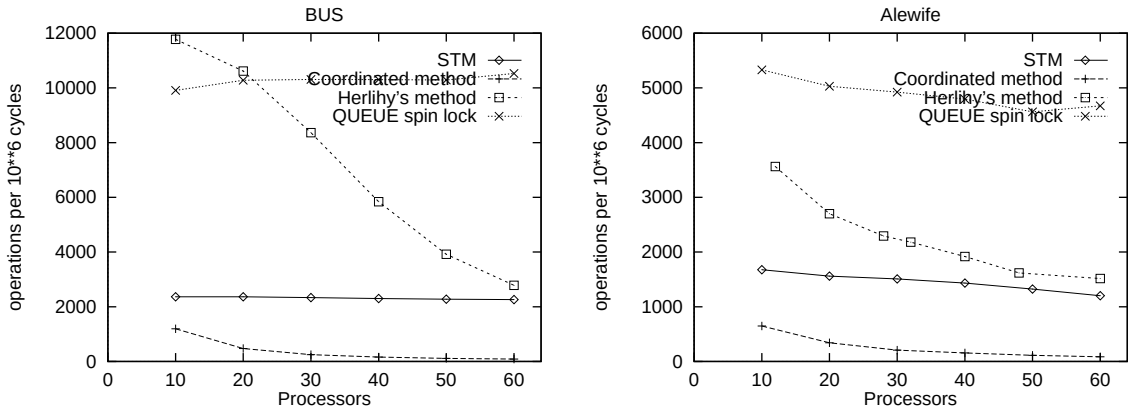

Figure 7: Counting Benchmark

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, 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 7. 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.

5.3 Resource Allocation Benchmark

The results of the resource allocation benchmark are shown in Figure 8. We measured the potential for parallelism as a percentage of the atomic s-word-increments that succeeded on first attempt. When $s = 2$ this percentage varies between 73-75% at 10 processors down to 33-34% at 60 processors. For $s = 4$ the potential for parallelism is 40-44% at 4 processors down to 16% at 60 processors, and when $s = 6$ it varies between 24-29% at 10 processors to 9-10% at 60 processors. In general, as the number of processors increases, local work can be performed concurrently, and thus the performance of the STM improves. 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. This is the reason for the relatively low throughput of the STM method for small numbers of processors and the concave form of STM graphs. As one can see, when $s = 2$ on the bus or when $s = 2, 4$ on Alewife architecture, the STM method outperforms even the queue-lock method.

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 10 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*.

5.4 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 [20], 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

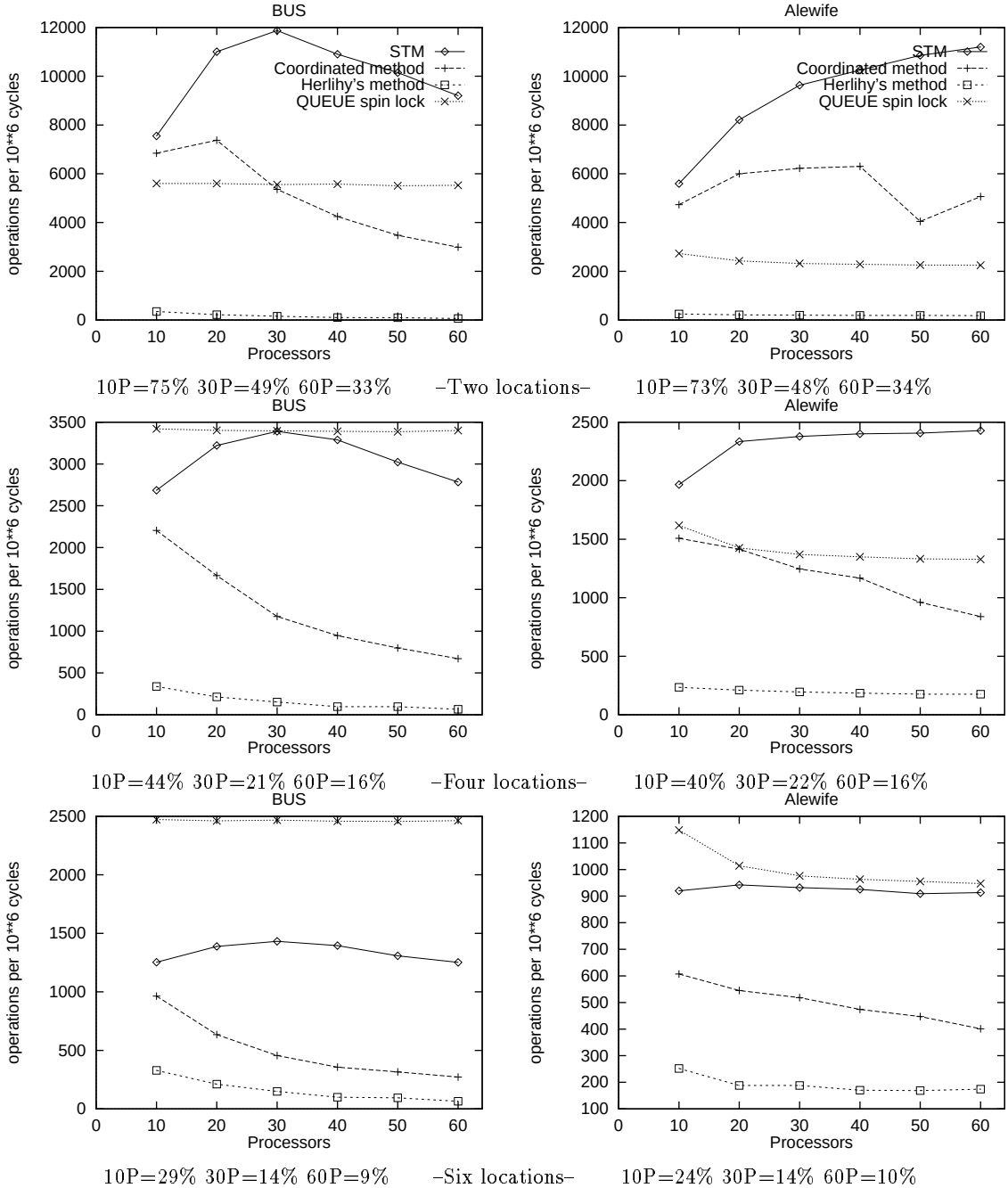

Figure 8: Resource Allocation Benchmark

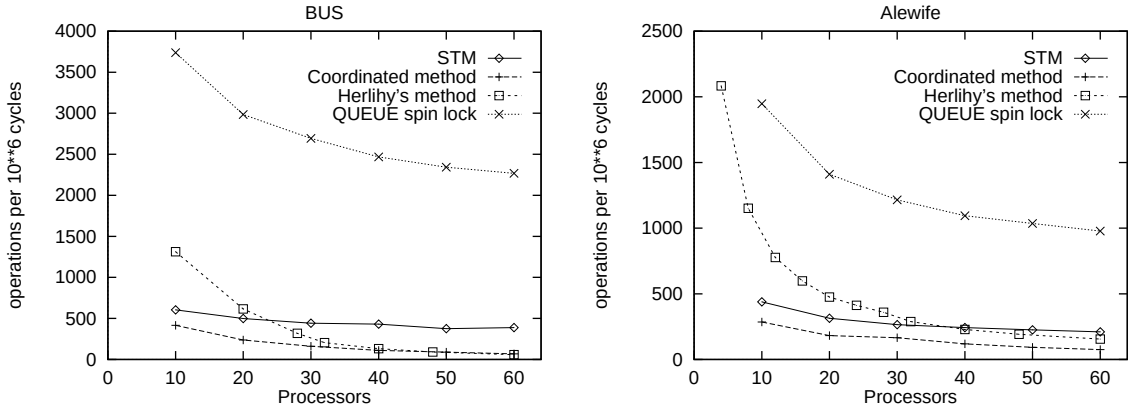

Figure 9: Priority Queue Benchmark

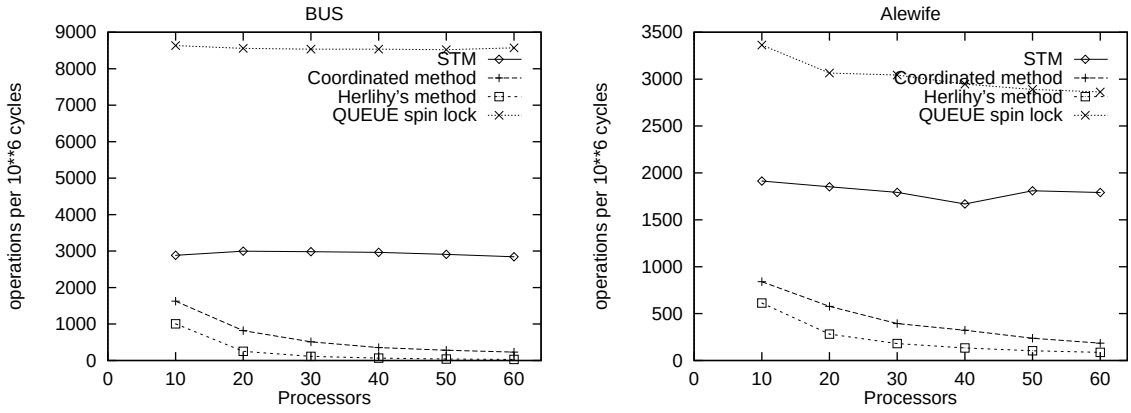

Figure 10: Doubly Linked Queue Benchmark

a 3-word Compare&Swap operation instead of a 2-word Store-Conditional operation [3].

We ran the same benchmark as for the regular priority queue. The results of the concurrent priority queue benchmark are given in Figure 15. 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

[3] In fact, using 3-word Compare&Swap simplifies the implementation since it avoids *freezing* [20] nodes

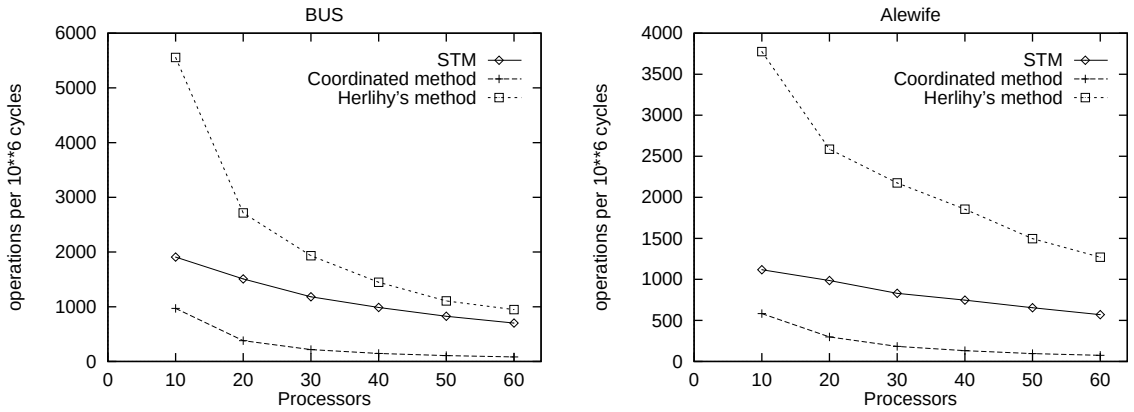

Figure 11: non-blocking comparison: Counting Benchmark

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 . Every theoretical method can be improved in many ways when implemented in practice.

In general, the result we got were that STM method outperforms Barnes method in all circumstances and, except from the counter benchmark STM outperforms Herlihy's method too. The results of the counter benchmark are shown in Figure 11. As in the previous tests Herlihy's method performs better than caching methods on both architectures. In Bus architecture Herlihy's method is from 2.91 times faster than STM on 10 processors, down to 1.35 time faster than STM on 60 processors. On Alewife architecture, Herlihy's method is from 3.38 times faster than STM on 10 processors, down to 2.23 times faster than STM on 60 processors. STM is from 1.97 faster than Israeli and Rappoport method on Bus architecture up to 8.44 time faster than Israeli and Rappoport method on 60 processors. On Alewife architecture, STM is from 1.92 times up to 7.6 time faster than Israeli and Rappoport method. This degradation in the performance of Israeli and Rappoport method is due to the high number of failing k-word Compare&Swap : up to 8.4 (!) times the number of successful k-word Compare&Swap while in STM the number of failing k-word Compare&Swap is at most 0.26 times the number of successful k-word Compare&Swap . Most of the transactions in STM terminate ad failing transactions and are not helped since they failed in acquiring the first (and last) location needed. In the priority queue benchmark, on a simulated bus architecture, Herlihy's method is from 2.36 times faster than STM, down to 2.8 times *slower* than STM. On the Alewife architecture, Herlihy's method has a throughput that is 2.41 times higher than STM throughput, down to 1.1 times *lower* than in the STM method. The results of the doubly linked queue are in Figure 13. On the Bus architecture, our STM method is up to 3.37 faster than

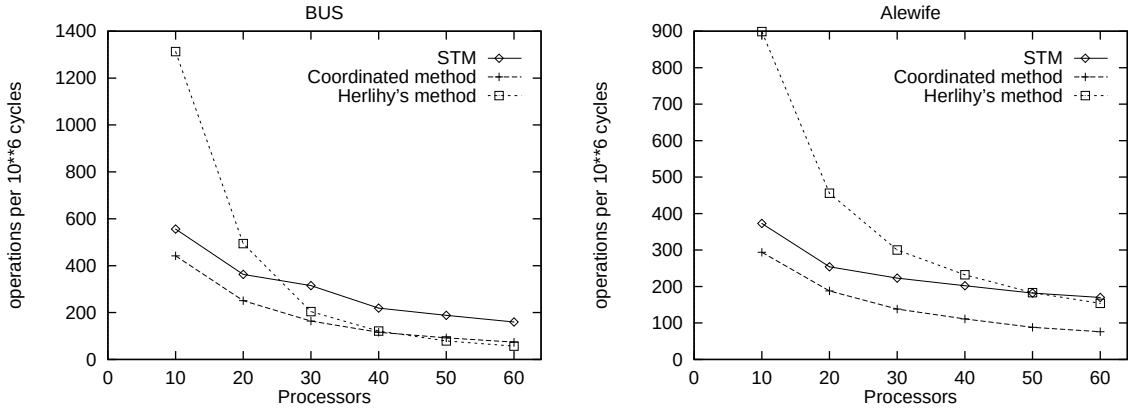

Figure 12: non-blocking comparison: Priority Queue Benchmark

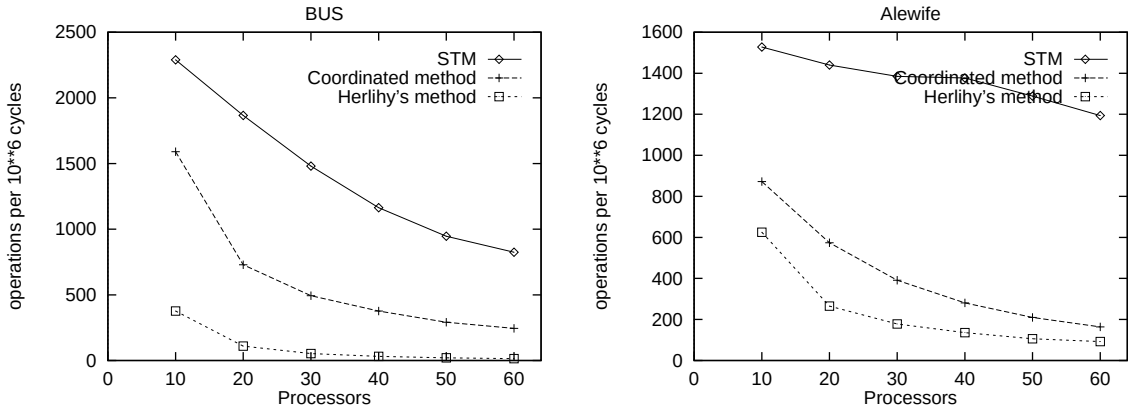

Figure 13: non-blocking comparison: Doubly Linked Queue Benchmark

Israeli and Rappoport method and up to 59 times faster than Herlihy's method. When simulating Alewife architecture, our method had a 12.9 times higher throughput than Herlihy's method and 7.28 higher throughput than Israeli and Rappoport method. In the resource allocation benchmark too (Figure 14) STM outperforms other methods: On Bus simulations, STM is between 1.1-1.27 times faster than Israeli and Rappoport method on 10 processors, and 1.6 times faster on 60 processors. On Alewife simulations, STM is 1.09-1.27 times faster than Israeli and Rappoport method on 10 processors up to 1.61-1.68 time faster on 60 processors. Note, that in this benchmark, the factor

that effect Israeli and Rappoport performance is not the number of failing k-word Compare&Swaps, which is relatively low, but more the remote redundant help that processors execute.

The results of the concurrent priority queue benchmark are given in Figure 15. Though the advantage that the inherent structure of the algorithm should give to Isreali and Rappoport method, STM give here also 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&Swaps.

6 Conclusions

Our paper introduces a non-blocking software version of Herlihy and Moss' transactional memory approach. There are many possible directions in which it can be extended. One issue is to design better non-blocking translation engines, possibly by limiting STM's expressability to a smaller set of implementable transactions. Another interesting question is what performance guarantees one can get with a less robust STM software package, possibly programmed on the machines' message passing level. Finally, the ability to add an STM component to existing software based virtual shared memory systems, raises theoretical questions on the computational power of a programming abstraction based on having a variety of "operations" that can be applied to memory locations, vs. the traditional approach of thinking of synchronization operations as "objects."

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.

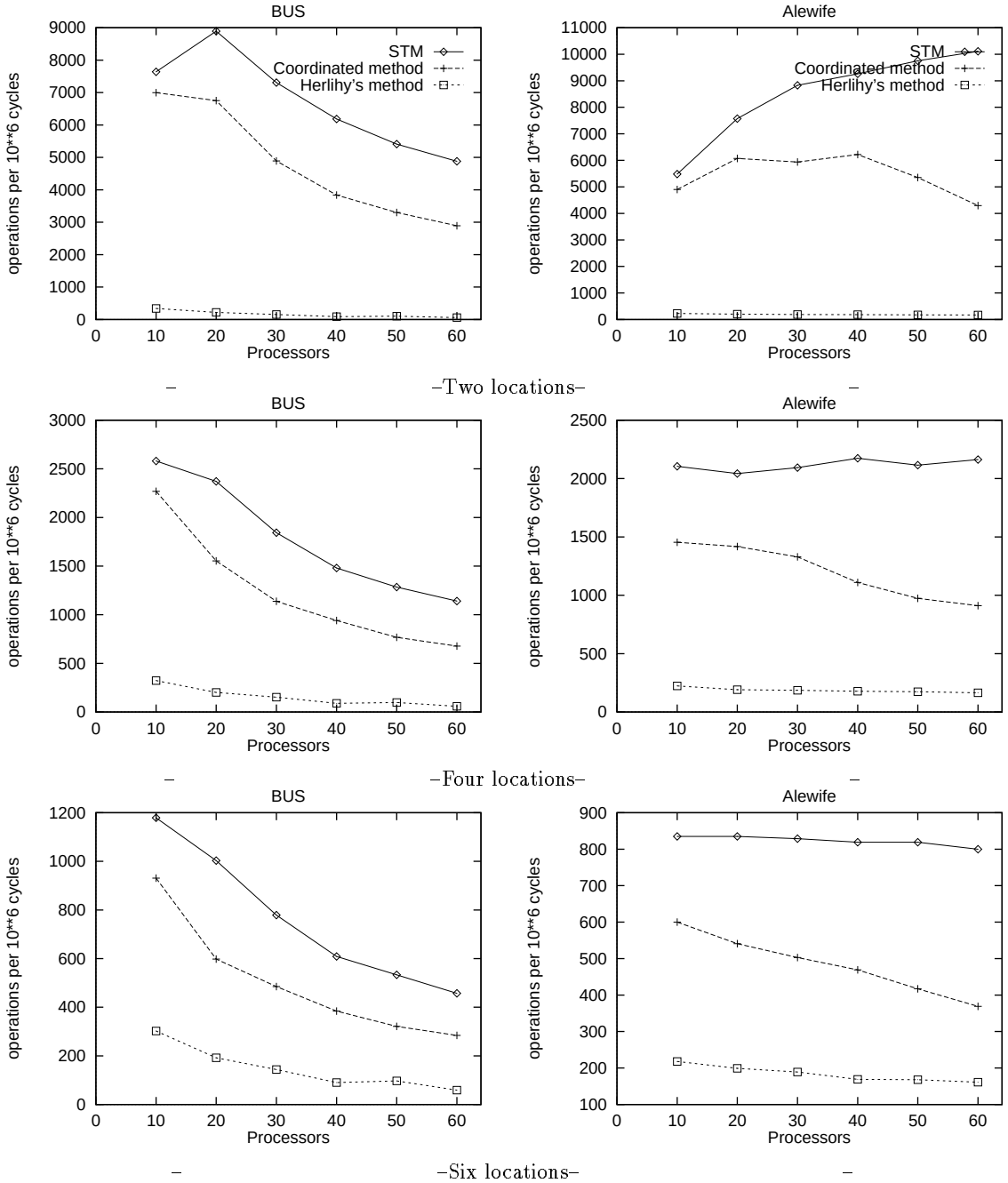

Figure 14: non-blocking comparison: Resource Allocation Benchmark

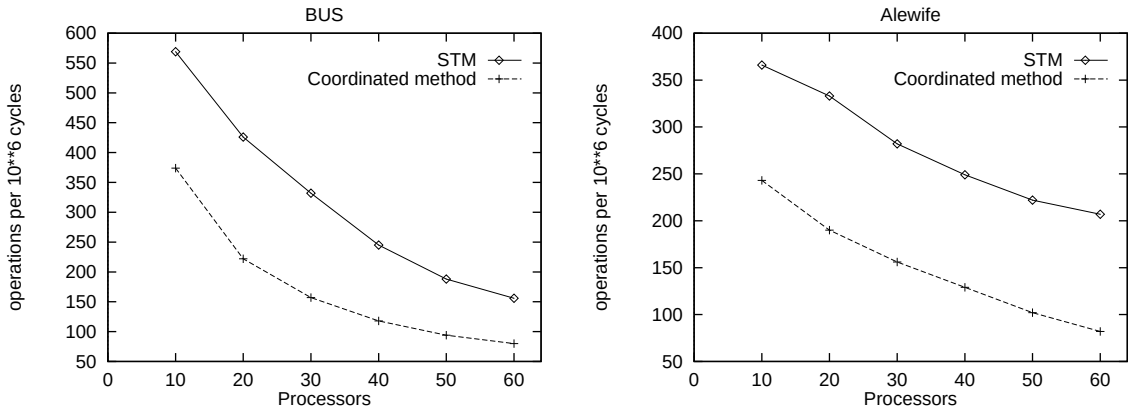

Figure 15: non-blocking comparison: Israeli & Rappoport Priority Queue

[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-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] David Chaiken. Cache Coherence Protocols for Large-Scale Multiprocessors. S.M. thesis, Massachusetts Institute of Technology, Laboratory for Computer Science Technical Report MIT/LCS/TR-489, September 1990.

[13] DEC. Alpha system reference manual.

[14] 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.

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

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

[17] 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.

[18] J. R. Goodman. Using cache-memory to reduce processor-memory traffic. In *Proceeding of the 10th International Symposium on Computer Architectures,* 13(1), pages 124-131, June 1983.

[19] IBM. Power PC. Reference manual.

[20] 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.*

[21] 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.*

[22] A. LaMarca. A Performance Evaluation of Lock-Free Synchronization Protocols. *Proc. of the 13th ACM Symposium on Principles of Distributed Computing, pages 130-140.*

[23] 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. Full version available as MIT Technical Report MIT/LCS/TR–387.

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

[25] 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.

[26] 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.

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

[28] 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.*

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