Efficient Nonblocking Software Transactional Memory

Virendra J. Marathe
Department of Computer Science,
University of Rochester
vmarathe@cs.rochester.edu

Mark Moir
Sun Microsystems Labs
mark.moir@sun.com

Technical Report #932
Department of Computer Science, University of Rochester

March 2008

Abstract

The current state of the art seems to favour blocking software transactional memory (STM) implementations over nonblocking ones, and a common belief is that nonblocking STMs fundamentally cannot be made to perform as well as blocking ones. But this belief is based on experience, intuition, and anecdote, not on rigorous analysis.

We believe there is still plenty of room for improvement in the performance of nonblocking STMs and that, regardless of performance, blocking is unacceptable in some contexts. It is therefore important to continue improving nonblocking STMs, both as a goal in its own right, as well as to inform research aimed at determining whether a fundamental gap exists between blocking and nonblocking STMs.

We describe a novel nonblocking copyback mechanism for a word-based software transactional memory (STM), which closely follows simple and efficient blocking mechanisms in the common case. Previous nonblocking copyback mechanisms impose significant overhead on the common case. Our performance experiments show that this approach yields significant performance improvement over the previous best nonblocking word-based STM. Our design approach can be applied to some other blocking STMs to achieve nonblocking counterparts that perform similarly in the common case.

Keywords: Software Transactional Memory, Nonblocking Progress, Performance, WSTM

1 Introduction

There has been a recent flurry of interest in transactional memory, and in techniques for impementing it in hardware, in software, or in a combination of the two. In this paper, we focus on software transactional memory (STM). Foundational research into transactional memory grew out of research into nonblocking concurrent data structures, which aim to overcome the many well-known software engineering, performance, and robustness problems associated with lock-based implementations. Thus, for a long time most transactional memory researchers considered it mandatory for transactional memory implementations to be nonblocking.

Recently, many researchers have developed *blocking* STMs, recognising that they are much easier to design and that the software engineering benefits of STM can be delivered even by a blocking STM. Nonetheless, hiding blocking from the application programmer does not eliminate all of its disadvantages. For example, a blocking implementation may cause a transaction to wait until a preempted thread is rescheduled, resulting in serious performance degradation in some cases. In other cases, blocking is more than just a performance problem: it is *unacceptable*. For one example, consider using an STM to coordinate access to a data structure that is shared between a thread and an interrupt handler. In this case, the thread must not block the interrupt handler, because the thread will not run again until the interrupt handler completes. The design of interrupt handlers is often significantly complicated by this restriction.

A common belief is emerging that blocking STMs are fundamentally faster than nonblocking ones [3, 4, 5, 24], based largely on an argument made by Ennals [5]. However, this is an informal argument based only on intuition, and not a formal lower bound proof. In particular, it misses the possibility of storing the logical values of transactional data in place in the common case, while displacing it to metadata located elsewhere when necessary to avoid blocking, which is the technique we use in the nonblocking STM we present here.

Research on improving nonblocking STMs may bear fruit by delivering performance improvements, or by informing efforts to prove a fundamental gap between blocking and nonblocking STMs, or both.

In this paper, we present a new nonblocking STM, which we designed by modifying a simple blocking STM to be nonblocking, while keeping its common-case behaviour (and thus performance) almost unchanged. Previous nonblocking STMs impose significant overhead on the common case in order to be nonblocking. We show that our design performs significantly better than the best previous comparable nonblocking STM, and comparably with and sometimes better than the simple blocking STM on which it is based.

1.1 Motivation and background

The industry is quickly moving towards multicore computing, rather than continuing to focus on improving single-core performance. Applications will thus increasingly need to exploit concurrency in order to benefit from advances in technology. In today's programming practice, sections of code are made to appear atomic using locks and condition variables. These blocking constructs have numerous well-known disadvantages related to software engineering, performance, and robustness, making concurrent programming a delicate art that is difficult even for experts. This situation is unworkable as the multicore paradigm increasingly compels everyday programmers to become concurrent programmers.

Transactional Memory (TM) allows programmers to express *what* should be executed atomically, leaving the system to determine *how* this atomicity should be achieved. Herlihy and Moss [14] proposed hardware transactional memory (HTM) and Shavit and Touitou [25] proposed software transactional memory (STM). While early proposals for both HTM and STM had some significant drawbacks, a recent flurry of activity in both HTM (e.g., [1, 7, 22, 23]) and STM (e.g., [4, 8, 9, 10, 13, 18, 24]) has yielded substantial progress towards practicality. Recently, *hybrid transactional memory* (HyTM) [2, 21] has been proposed, in which transactions can be attempted using HTM (if HTM is available) or using a compatible STM. While we do not focus on HyTM in this paper, all of the techniques we present are compatible with this approach.

Transactional memory largely eliminates the software engineering problems associated with locks. However, the extent to which it also eliminates problems related to performance, scalability, and robustness depends on the system support for transactions. Numerous research groups are working on STM implementations for a variety of execution contexts, and this work has shed light on a rich design space, and a number of issues and tradeoffs.

STMs can be characterised as either *word-based* or *object-based*. Word-based STMs designs are more constrained than object based ones, because they cannot depend on aspects of an existing object infrastructure, such as existing object metadata, type safety for pointers, etc. However, this means they can be used in programming languages such as C and C++, which are frequently used to implement the object infrastructure upon which object-based STMs (such as DSTM [13]) are built. Thus, word-based STMs have the potential to benefit a much wider range of applications than object-based ones do. In this paper we focus on word-based STMs, though our general approach is also applicable to the design of object-based ones.

A key issue for any STM design is the extent to which it avoids blocking. Generally, implementations providing strong nonblocking progress properties such as wait-freedom [11] entail significantly more overhead and are less practical than implementations providing weaker properties such as lock-freedom [11]. Recently, Herlihy et al. [12] identified an even weaker progress property, which they called *obstruction-freedom*. The weaker property admits simpler synchronization algorithms that are faster in the common uncontended case because the mechanisms that must be integrated directly into the algorithms in order to provide a stronger property such as lock-freedom or wait-freedom are unnecessary. Exploiting this advantage, Herlihy et al. [13] presented an obstruction-free *dynamic* form of STM (DSTM), achieving an STM system that is significantly more practical than previous ones. This work has lead a number of research groups to experiment with variatons on DSTM [6, 16, 17, 18, 20].

An STM implementation is *obstruction-free* if it guarantees that, if a transaction is repeatedly retried and eventually encounters no conflicts with other active transactions, then eventually the transaction commits successfully. Obstruction-freedom makes no progress guarantees in case of repeated interference between two or more transactions, and Herlihy et al. [13] observed that resulting “livelock” scenarios are not merely a theoretical concern: livelock occurs in practice if nothing is done to prevent it. Herlihy et al. also showed that, by augmenting an obstruction-free STM with a modular *contention manager*, such livelocks can be eliminated in practice. This is the approach we have taken in designing our nonblocking STM.

1.2 Overview of our design approach and STM

We present a new technique for implementing an obstruction-free STM that closely tracks the behaviour and performance of a simple blocking scheme until the contention manager decides that a transaction should not wait for another to complete. In a blocking implementation, there is no choice in such circumstances: the mechanism dictates the policy.

To demonstrate our idea, we have implemented two word-based STMs, one blocking and one nonblocking. The only other nonblocking word-based STM of which we are aware is Harris and Fraser's WSTM [8, 6]. We found their nonblocking copyback mechanism ingenious, and in fact, our work was inspired in part by theirs. However, we also felt that it imposed too much overhead on the common case in preparation for avoiding blocking. We later explain several sources of overhead in WSTM that we have eliminated, and present performance experiments verifying that this indeed translates to significantly improved performance.

In our blocking STM implementation, as a transaction executes it acquires exclusive ownership of locations it intends to modify, and copies new values back to modified locations upon successful commit, before releasing ownership. Because transactions maintain exclusive ownership of these locations throughout the copyback, no other transaction can access the locations until the copying is complete. Therefore the copying can be performed with simple store instructions, and no transaction will ever detect the difference between the *logical* value stored to a location by a recently-committed transaction, and the *physical* value that remains in the location because the committed transaction has not yet copied its new value to it. However, this simplicity comes at a price: a transaction that needs to access a location that is owned by a committed transaction has no choice but to wait.

Our blocking STM includes some novel optimizations. In modifying this STM to achieve our nonblocking STM, we were able to retain all of these optimizations, resulting in a nonblocking STM that largely eliminates a number of sources of overhead in WSTM.

Our nonblocking STM is structured similarly to the blocking one, and in the common case, a transaction behaves almost identically to the blocking STM. However, it overcomes the blocking nature of our first STM by allowing a transaction to determine the logical value of a memory location that is owned by a committing transaction, and even to "steal" ownership of the memory location from the committing transaction while preserving this logical value. This way, committing transactions do not prevent progress by others.

Committing transactions usually copy their values to the affected locations using ordinary stores with no additional synchronization, just as in the blocking STM. A stealing transaction cannot control when its victim will perform its store to a stolen location. Therefore, when a location is stolen, its logical value is subsequently determined by separate transactional metadata associated with the location, making the timing of the victim's stores to the affected memory locations irrelevant. The ownership data associated with the location is marked as "stolen" until it can be determined that the original owner is no longer copying values back to the affected memory locations, at which time normal operation can resume.

Managing stolen locations and their logical values accounts for most of the complexity in our nonblocking STM. However, until such time as we decide to steal a location, the STM closely mimics the simple blocking STM, and performs very similarly. Thus, we show that it is possible to have the *ability* to make progress despite a conflict with a delayed transaction, while incurring very little overhead until this option is exercised.

1.3 Roadmap

The remainder of this paper is structured as follows. We describe our simple blocking STM in Section 2 and the modifications we made to it to achieve the nonblocking STM in Section 3. Section 4 compares our nonblocking STM to the closest previous work, namely WSTM [8]. In Section 5, we present some performance experiments showing that the simple blocking STM substantially outperforms WSTM, and that our nonblocking STM performs comparably with the simple blocking one in most cases, and alwats outperforms WSTM significantly. We conclude in Section 6.

2 A Simple Blocking STM System

In this section, we describe a simple STM that uses well-known techniques to avoid blocking in most cases, but does not attempt to avoid blocking during commit, which is significantly more complicated. This STM is derived from the one used in the hybrid transactional memory prototype described in [2], but includes some differences and some novel optimisations, as described below. Our main contribution is in the next section, where we show how to extend the blocking STM to make it nonblocking; the nonblocking STM preserves these optimisations.

In a nutshell, our simple blocking STM is a *word-based* STM that employs *eager ownership acquisition* and buffers modifications in a *write set* to be copied back to the affected memory locations upon commit. In contrast to the STM in [2], we used *invisible read sharing* to allow a more direct comparison to Harris and Fraser's WSTM [6]. Different read sharing techniques perform better for different workloads, and we did not want this factor to complicate our comparison.

Our STMs all provide the following interface:

`stm_begin(my_txn)`	Begins transaction `my_txn`.
`stm_read(my_txn, addr)`	`my_txn` reads from memory location `addr`.
`stm_write(my_txn, addr, value)`	`my_txn` writes `value` to `addr`.
`stm_commit(my_txn)`	Commits `my_txn`.
`stm_abort(my_txn)`	Aborts `my_txn`.

Mapping higher-level language notation for atomic blocks onto this interface, for example using a compiler, is beyond the scope of our description here.

2.1 Overview of blocking STM

Each transaction has a *status*, which is `ACTIVE`, `ABORTED`, or `COMMITTED`. A transaction starts with an `ACTIVE` status, and it commits by atomically (for example using the compare-and-swap (CAS) instruction) changing its status from `ACTIVE` to `COMMITTED`; another transaction that wishes to abort the transaction can attempt to do so by using CAS to change its status from `ACTIVE` to `ABORTED`.

As a transaction executes, it acquires exclusive ownership of each location it accesses, and maintains a *write set* (set of address-value pairs) to record what values it intends to write to which memory locations.

Transactional read operations look up the write set to determine if the transaction has previously stored a value to the accessed location, returning the most recent such value if so. Otherwise, they determine the logical value of the location based on the definition given below, aborting the owning transaction if it is active to avoid the transaction committing and thus changing the value of the location read. The transaction also records information about the location being read in a *read set* for later validation. Read validation ensures that a transaction commits only if none of the values it has read have changed since they were read, and also ensures that user transaction code does not run having read inconsistent values from different locations, which could result in arbitrary behaviour even though the transaction is doomed to fail.

Correct behaviour of an STM is defined using the standard linearisability [15] condition. Formal details would only obscure our description here, but a brief intuitive discussion is necessary to understand our algorithms. Because STM must provide the illusion of multiple memory locations changing atomically, while in reality the memory locations are updated at different times using separate machine instructions, we assign a *logical value* to each memory location at any point in time, and ensure that each transaction appears to execute instantaneously at its *commit point*, reading the logical values of memory locations it reads and changing the logical values of memory locations it writes.

In our simple blocking STM, the logical value of a memory location with address a is defined to be the contents of location a, unless a is owned by a transaction that has committed and has a pair (a, v) in its writeset, in which case its logical value is v. Thus, when a transaction successfully commits, the logical values of all locations written by the transaction instantaneously change to the last values written to them by the transaction. Thus the transaction "takes effect" at its commit point, even though the values it writes are not yet stored in the target locations. After committing, the transaction copies the values from its write set to the indicated locations before releasing ownership of them.

Because a transaction maintains exclusive ownership of each location it modifies until after it copies back the new value from the writeset to it, no other transaction can modify these locations while the copyback is being performed, making it simple to determine the logical values of the locations by examining the transaction's write set. However, this also makes the simple STM blocking: a transaction that wishes to access such a location must simply wait. If a transaction requires ownership of a location that is owned by another transaction that is active (i.e., has not yet committed or been aborted), it can abort the active transaction, thus ensuring that it will not commit successfully. It can then directly acquire ownership of the location because there is no risk that the active transaction will commit. Decisions about whether to abort a conflicting transaction are made by a separate *contention manager* [13], which guarantees that any deadlock that arises is eventually resolved by aborting one or more transactions.

2.2 Details of blocking STM

2.2.1 Data structures

Figure 1 illustrates the primary data structures in our implementation, as described below. Components in **bold** are not present in the blocking implementation.

The primary data structures in our STMs are *transaction descriptors*, which are used to represent transactions, and a table of *ownership records* (orecs), which are used to represent ownership by transactions of memory locations. Each location in memory is covered by one orec in the table, determined by a simple hash on the location's address. Thus, ownership of multiple memory locations is represented by a single orec.[1]

Each transaction descriptor consists of a transaction ID (`tid`), a version number (`version`), a `status` indicator as described above, and read and write sets (`rs` and `ws`). Transaction descriptors can be reused. The `tid` and `version` fields together uniquely identify a transaction.

The read and write sets are organised into per-orec *rows*. Each read set row contains an orec identifier and a snapshot of the orec. Each write set row contains an orec identifier and an array of *entries*, each of which is an address-value pair for some address covered by the indicated orec. Each time a transaction writes to an address corresponding to an orec it has not yet acquired, it uses the next available row in its write set and creates in it an entry to record the address of the location and the value written. Subsequent writes to locations covered by the same address either update the value stored in the existing entry for the location written, or create a new entry in the row if none exists.

In our implementations, we preallocate rows for the read and write sets. If a transaction exceeds the number of preallocated rows, we abort it, allocate a larger set and retry. It is not hard to refine this implementation to allocate new rows dynamically, without aborting the transaction. If a transaction exceeds the preallocated array of write set entries for a given row, further entries are allocated dynamically and stored in a linked list.

Each orec is stored in one 64-bit word; all modifications to orecs are done using a CAS instruction or similar synchronization primitive to ensure that an orec does not change between a transaction

[1]As Ennals [5] and others point out, colocating transactional metadata in the same cache lines as the data it controls can result in significant performance improvements. This can be done in object-based STMs for garbage collected systems. However, the blocking STM on which we based our implementation is intended for use in general C/C++ programs in which no such object infrastructure exists, and the compiler lays out data according to existing standards. In this context, it is not acceptable to modify the layout of data to improve performance. If it were acceptable, we could easily do so, both with our blocking STM and the nonblocking ones based on it; this issue is orthogonal to the blocking vs. nonblocking debate, despite Ennals's suggestions that this is an optimisation that can be used with blocking STMs and not with nonblocking ones.

examining and updating the orec. Each orec consists of: `tid` and `version` fields, used to identify the transaction that most recently acquired ownership of the orec; a `mode` field indicating that the orec is `UNOWNED` or `OWNED`; and a `row` field, used to identify the write set row in which the owning transaction stores entries for locations mapping to this orec. This allows a transaction to quickly look up the write set entry for a particular memory location when it already owns the associated orec.

2.2.2 Upgrading

To ensure correct behaviour if a transaction reads a location covered by an orec and subsequently writes a location covered by the same orec, when a transaction acquires an orec, it records a copy of the orec as it was immediately before the acquisition. When validating the read set, if a transaction observes that it owns an orec it is validating, it checks to make sure that the orec snapshot recorded in the read set matches the snapshot from immediately before the acquisition; if not, the transaction is aborted. This ensures that the transaction does not commit if another transaction writes a location between the read and the subsequent orec acquisition.

2.2.3 Fast release

Because a transaction that acquires ownership of an orec stores both its `tid` and `version` in the orec, it can logically release ownership of all locations it has acquired by simply incrementing the `version` field of its transaction descriptor. This avoids the need to release orecs individually, a significant optimisation. In some cases, the profit from this *fast release* optimisation is lost because it creates additional work for a transaction that subsequently wishes to acquire a released orec: this transaction must inspect the `version` field of the previous owner's transaction descriptor to determine that it has been released. However, by removing the cost of releasing ownerships from the critical path of transactions, we enable a number of flexible alternatives, such as resetting locations to `UNOWNED` mode incrementally, in the background, or using idle processors. Furthermore, because a transaction will generally have its own transaction descriptor cached, this checking comes essentially for free when a transaction encounters an orec owned by a previous transaction that used the same descriptor. Because threads generally retain and reuse transaction descriptors, this is especially important for single-threaded applications and those that exhibit a high degree of locality.

3 The Nonblocking STM

3.1 Overview of nonblocking STM

In the nonblocking STM, a transaction can "steal" an orec from a committed transaction, rather than waiting for it to complete. Until a transaction decides to steal an orec from a committed transaction, the nonblocking STM behaves very similarly to the blocking version. Specifically, a transaction only needs to keep new values for addresses it modifies, and copy them back using ordinary store instructions, without additional synchronization.

When a transaction steals an orec and subsequently commits, it cannot simply copy its values back to locations covered by the stolen orec as usual, because the "victim" transaction from which it stole the orec might still be copying back its values, and may therefore overwrite newer values written by the "stealer". Therefore, while the victim may still be performing its copyback, the logical values of locations covered by the stolen orec must be maintained in the transaction row associated with the orec.

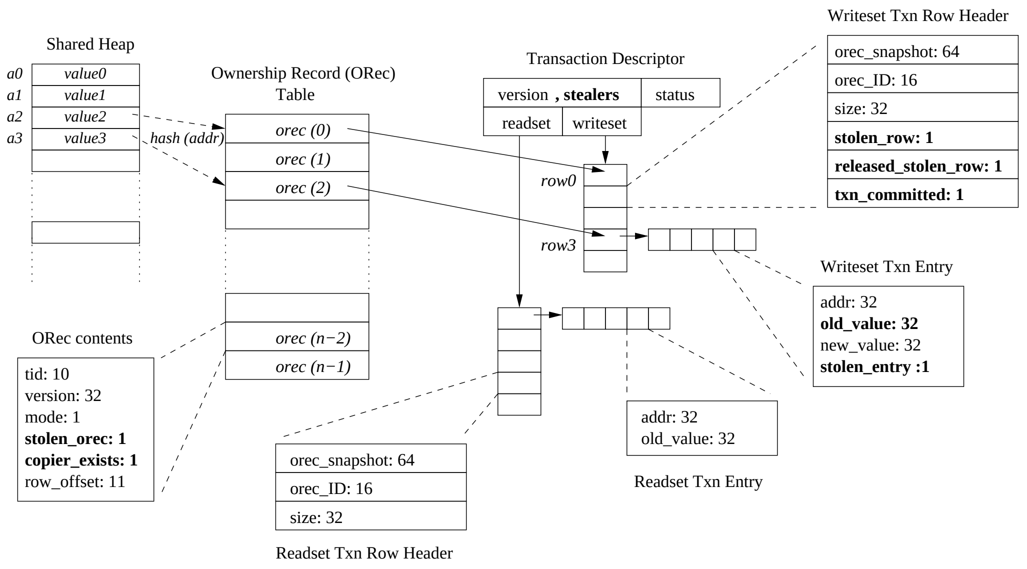


Figure 1: Illustration of main data structures in blocking and nonblocking STMs. **Bold** components only in nonblocking STM.

It is more expensive to access locations covered by stolen orecs, so it is desirable to eventually reset a stolen row to its normal, unstolen state. Before doing so, we must first ensure that the memory locations covered by the orec contain their logical values, and that no transaction is still copying back values to these locations. To keep our algorithm simple and efficient, we maintain the invariant that only one transaction at a time can be copying back values to memory for a particular orec. When an orec is first stolen, the transaction from which it was stolen is the one transaction copying back values to locations covered by that orec. After it has finished, another transaction can assume this role, copy back values stored in the transaction row associated with the orec, and then reset the orec to unstolen. We explain how this is achieved in more detail later.

3.2 Details of nonblocking STM

3.2.1 Data structures

The data structures used for the nonblocking STM are similar to those used for the blocking STM described in the previous section, with the following additions. Orecs contain two additional flags `stolen_orec` and `copier_exists`. Each write set row contains an indication `OrecID` of the orec that was most recently associated with the row, a `stolen_row` flag, a `released` flag, and a `txn_committed` flag. Each transaction entry, in addition to the `addr` and `newval` fields used in the blocking STM, has an `oldval` field and a `stolen_entry` flag. Finally, transaction descriptors have an additional `stealer_note` field, which is the head of a linked list of pointers to orecs. The `stealer_note` and `version` fields are colocated so that they can be accessed atomically using CAS.

3.2.2 Stealing, restealing, and resetting orecs

When an orec's `stolen_orec` flag is false, the logical value of a memory location covered by this orec is defined as before, and transactional execution is essentially as before. When the `stolen_orec` flag is true, the logical values of locations covered by the orec are determined by the contents of the transaction row indicated by the orec, and the status of the transaction that last acquired ownership of it. The details of the logical value definition are significantly more complicated in this case. Below we explain how orecs are stolen, restolen, and reset to unstolen, ignoring for now the logical value of memory locations covered by stolen orecs.

The `copier_exists` flag is used to ensure that the orec is not reset to unstolen while some transaction may be copying back values to locations covered by that orec. When a transaction steals an unstolen orec, it sets both the `stolen_orec` flag and the `copier_exists` flag to reflect that the victim transaction is potentially copying back values. When the victim transaction has completed copying back its values, it indicates this by clearing the `copier_exists` flag.

Careful coordination is required so that a transaction that has completed its copyback knows the set of orecs for which it should reset the `copier_exists` flag (if any). To inform the victim that it should reset the `copier_exists` flag in the stolen orec, a stealer attempts to insert a stealer note in the victim's stealer note list. The head of this list is modified using a CAS that also checks that the victim's version number has not changed. When the victim increments its version number after completing its copyback, it also takes a snapshot of the head of its stealer list, and sets it to NULL. The victim then proceeds to clear the `copier_exists` bit of each orec indicated by a note in the stealer note list. This paves the way for a subsequent stealer to reset the orec to unstolen.

A stealer either succeeds in inserting a stealer note in its victim's stealer note list, and therefore knows that the victim will get the note and clear the bit, or it fails to insert the note because the version number has changed. In the latter case, the stealer can infer that the victim has completed its copyback, so in fact there is no transaction copying back values to locations associated with this orec. In this case, we can consider that the orec was not really stolen, and the stealer can simply reset the `stolen_orec` and `copier_exists` bits in the orec and reestablish normal conditions for the orec, *provided the orec has not been restolen in the meantime*. If it *has* been restolen, then the failed stealer cannot simply reset the orec to normal conditions, because the logical values of some locations may now be stored in the transaction row of the owning transaction. Therefore, in this case, the failed stealer simply resets the `copier_exists` flag, thereby indicating that no transaction is currently copying back values to locations covered by this orec. Again, this allows a subsequent stealer to reset the orec to unstolen.

If a transaction wishes to acquire an orec that has its `stolen_orec` flag set, but its `copier_exists` bit clear, then because there is no transaction copying back values to locations covered by this orec, it can resteal the orec, setting the `copier_exists` flag. If this succeeds, then this stealer can copy back all values in the transaction row indicated by the orec, and then reset the orec to unstolen, again provided it has not since been restolen. Otherwise, if the transaction cannot reset the orec to unstolen, it proceeds as usual, but refrains from copying back values from transaction rows associated with stolen orecs upon successful commit. These values remain in the transaction row associated with the orec until some transaction successfully sets the `copier_exists` flag and copies back the values, as described above.

If an orec is repeatedly restolen before a transaction can reset it to normal, then the orec remains stolen, with an associated performance cost. For this reason, a decision to steal should not be taken lightly: a transaction should usually wait for at least a short amount of time before stealing an orec, in the hopes that the owning transaction will soon relinquish it. Similarly, a transaction should pause before restealing an already stolen orec, in the hopes that the orec will soon be reset to normal.

3.2.3 Preserving logical values when stealing orecs

When a transaction steals an unstolen orec, it must preserve the logical values of each location modified by the victim (which is committed). Thus, for each entry in the victim's write set row, the stealer creates an entry in its new transaction row for the same memory address, taking the value from the victim's `newval` field, and marking each created entry as stolen by setting its `stolen_entry` flag. The stealer copies the value to *both* the `oldval` and `newval` fields of the write set entry. This way, the stealer transaction can add new (unstolen) entries for locations it modifies after stealing the orec, and can update the `newval` fields of stolen and unstolen entries in the same way. If the stealer subsequently commits, then the `newval` fields of all entries in the row become the logical values of the specified locations; if it aborts, then the `oldval` fields of the stolen entries continue to represent the logical values of the specified locations, while the new unstolen entries become irrelevant.

Creating the new transaction row when stealing a *stolen* orec is similar, except that the stealer must also copy stolen entries in the victim's transaction row (as well as the victim's new entries if necessary, as before). Which field contains the value to be copied is determined by the owning transaction's status as explained above.

3.2.4 Managing stolen transaction rows

A transaction row cannot be reused while it is associated with a stolen orec because it contains the logical values of some locations. This restriction seemingly precludes immediate reuse of transaction descriptors and requires each transaction to allocate a new descriptor, as well as a scheme for reclaiming them.

To avoid this expense and complication, we have incorporated a mechanism that allows a descriptor to be immediately reused, even if one or more of its rows are associated with stolen orecs. This is achieved by marking transaction rows as stolen before associating a stolen orec with them, and having transactions that steal a stolen row then mark the row as having been released for reuse. In the meantime, a transaction reusing the descriptor simply skips rows that have been stolen and not yet replaced. If a pathological situation arises such that many or all of the rows for a particular descriptor are marked as stolen and cannot be reused, a thread can allocate a new transaction descriptor, but we avoid this expense in all but these uncommon scenarios.

Another issue that arises from the desire to reuse a transaction descriptor while one or more of its rows is associated with a stolen orec is determining the fate of the transaction that last stole the orec. Recall that this is necessary in order to determine which of the `oldval` and `newval` fields contains the logical value of locations covered by the row. But because the transaction descriptor may have been reused, its `status` field no longer records the status of the transaction that created the stolen row. Therefore, a transaction records its fate in the `txn_committed` flag of each stolen row in its write set *before* reusing the descriptor.

A slightly subtle mechanism is required to correctly handle races between a stealer and a transaction preparing to reuse its descriptor. Before restealing a stolen orec, the stealer examines the `status` field of the transaction descriptor indicated by the orec, and then checks the `version` field of the descriptor. If the version still matches the one in the orec, then the value read from the `status` field was accurate. Otherwise, the original transaction has already recorded its fate in the `txn_committed` flag of the row, so the new transaction uses that value.

	Contents of orec 1						Relevant write set row changes						M[3]	LV[3]
	`tid`	`ver`	`U/O`	`off`	`stln`	`cp_ex`	`tid/off`	`orec`	`stln`	`rel`	`tc`	`entries`		
a)	-	-	U	-	-	-						(`addr`,`oldval`,`newval`,`stln`)	40	40
b)	0	0	O	0	F	F	0/0	1	F	-	-	(3,-,41,F)	40	41
c)	1	0	O	0	T	T	1/0	1	T	F	-	(3,41,41,T),(1,-,21,F)	40	41
d)	1	0	O	0	T	T	1/0	1	T	F	-	(3,41,42,T),(1,-,21,F)	40	41
e)	2	0	O	0	T	T	1/0	1	T	T	-	(3,41,42,T),(1,-,21,F)		
							2/0	1	T	F	-	(3,41,43,T)	40	41
f)	3	0	O	0	T	T	2/0	1	T	T	-	(3,41,43,T)		
							3/0	1	T	F	T	(3,41,44,T)	40	44
g)	3	0	O	0	T	F							41	44
h)	0	1	O	0	T	T	0/1	1	T	F	-	(3,44,44,T)	41	44
i)	0	1	O	0	F	F	0/1	1	F	-	-	(1,-,22,F)	44	44

Figure 2: Changes to orec 1, relevant write set rows, and physical and logical values of location 0x3 during example. Abbreviations: `ver` means `version`, `off` means `row_offset`, `stln` means `stolen_orec` in orecs and `stolen_entry` in write set entries, `cp_ex` means `copier_exists`, `rel` means `released_stolen_row`, `tc` means `txn_committed`.

3.2.5 Example

The following simple example partially illustrates stealing, restealing, and resetting orecs. We assume a small memory with four locations; we use notation 0x2 to denote the location with address 2. Memory location i is covered by orec i mod 2. We use $M[i]$ to denote the contents of memory location i, and $LV[i]$ to denote the logical value of location i. Initially, all orecs are in unowned (U) mode, so all other fields in the orecs are irrelvant. Initially, $M[0] = 10$, $M[1] = 20$, $M[2] = 30$, and $M[3] = 40$. Because all orecs are unowned, $LV[i] = M[i]$ for all i.

We have the following five transactions executed by threads 0 through 3, where thread i uses transaction descriptor i. We denote transactions by thread/version number.

Transaction 0/0:	`stm_write(3,41)`
Transaction 1/0:	`stm_write(1,21)`; `stm_write(3,42)`
Transaction 2/0:	`stm_write(3,43)`
Transaction 3/0:	`stm_write(3,44)`
Transaction 0/1:	`stm_write(1,22)`

Observe that all locations written by these transactions map to orec 1. Figure 2 shows the contents of this orec, as well as the contents of relevant write set rows and the physical and logical values of location 0x3 after each step in our example execution. In this table, "-" denotes an irrelevant value, which might not have been initialised.

Initially (step a), orec 1 is unowned, so the contents of all other fields are irrelevant.

In step b), transaction 0/0 calls `stm_write(3,41)`, and therefore acquires orec 1 and adds an entry to its write set for writing 41 to location 0x3. Transaction 0/0 then commits (changing its status to `COMMITTED`) and then stops (without copying back its values).

In step c), transaction 1/0 calls `stm_write(1,21)` and therefore needs to acquire orec 1. Because the orec is owned by transaction 0/0, which has status `COMMITTED`, transaction 1/0 decides to steal the orec. It sets up a stolen row in its write set and creates a stolen entry for location 0x3 in order to preserve the logical value of this location, namely 41, from transaction 0/0's write set row. It then adds an unstolen entry for its store of 21 to location 0x1 and steals the orec, setting the `stolen_orec` and `copier_exists` flags to true.

In step d), transaction 1/0 calls `stm_write(3,42)` and therefore updates the new value for location 0x3 to 42 in its write set. At this point, if transaction 1/0 were to commit, $LV[3]$ would become 42 and $LV[1]$ would become 21. On the other hand, if it aborts, the `oldval` field of its write set entry for 0x3 preserves the value 41 previously committed by victim transaction 0/0. At that time, however, since location 0x1 was not stolen originally, the logical value of 0x1 (`LV`[1]) is the same as its physical value `M`[1].

In step e), transaction 2/0 calls `stm_write(3,43)`. Because orec 1 is owned by active transaction 1/0, transaction 2/0 first aborts it (changing its status to `ABORTED`). It then creates a stolen entry for 0x3 in its write set row, copying the `oldval` field (41) of 1/0's entry into both its `oldval` and `newval` fields. It resteals the orec, and updates the `newval` field of its write set entry for 0x3 to 43. Finally, it sets the `released` bit of 1/0's write set row, so that the row can be reused by a subsequent transaction executed by thread 1.

In step f), transaction 3/0 calls `stm_write(3,44)`. It first aborts transaction 2/0, then creates a stolen transaction row containing a stolen write set entry for 0x3, preserving the logical value 41 in both its `oldval` and `newval` fields. It then steals the orec, updates the `newval` field to 44 to reflect its own store, and then commits. Because this is a stolen row and a copier (transaction 0/0) still exists for this orec, transaction 3/0 does *not* copy back 44 to location 0x3. It then writes true to the `txn_committed` of its write set row. This is to allow subsequent transactions to determine that 44 is the correct logical value for locaton 0x3, even if transaction descriptor 3 is reused (note that the row will not be reused until its `released` field is set to true). Finally it atomically changes its version number to 1 and observes that it has no stealer notes. Transaction 3/0 is thus complete.

In step g), transaction 0/0 completes its copyback, storing 41 to location 0x3. Note that this is *not* the correct logical value of this location, because transaction 3/0 has committed since transaction 0/0 did, making 44 the logical value at location 0x3. The correct logical value is indicated by the entry in write set row 0 of transaction descriptor 3. Because the version number of transaction descriptor 3 no longer matches the version number stored in the orec, the `txn_committed` field indicates that transaction 3/0 committed successfully, so the logical value (44) is in the `newval` field. Now transaction 0/0 atomically increments its version number and takes a snapshot of the head of its stealer note list, which contains one stealer note informing transaction 0/0 that it should clear the `copier_exists` flag of orec 1. It does so, keeping the rest of the orec unchanged.

In step h), transaction 0/1 (reusing tranaction descriptor 0, with version number 1) calls `stm_write(1,22)`. It resteals the orec as before, having created a stolen write set row containing a stolen entry for location 0x3 with the logical value 44 (determined as described above) in its `oldval` and `newval` fields. However, in this case, because the `copier_exists` flag was clear before the restealing, tranaction 0/1 knows that it has the right to copy values back to locations covered by this orec. So transaction 0/1 sets the orec's `copier_exists` bit while restealing the orec.

Thereafter, in step i), transaction 0/1 stores 44 to location 0x3, and then resets the orec to unstolen. Finally, it creates an unstolen entry in its writeset row reflecting its own write of 22 to location 0x1. Now normal treatment of this orec can resume, with the simple lightweight copyback of the blocking mechanism.

3.2.6 Read sharing

Read sharing is as described previously for the blocking STM, except when a transaction reads from a location that is covered by a stolen row. In this case, as before `stm_read` takes a snapshot of the relevant orec and stores it in the read set, if it has not done so already. However, determining the logical value is more complicated when the orec is stolen, as explained above. In our implementation, the first time a transaction reads from a stolen orec, it creates a read set entry in its read set row for each stolen location in the write set row indicated by the orec, similarly to the way a stealing transaction does. Subsequent reads to locations covered by this orec search for an entry in the row for the specified address, supplying the value from that entry if one is found, and otherwise reading it from the location itself. A variety of alternatives to this strategy exist, with different tradeoffs with different expected workload characteristics. We have not explored alternatives, or adapting between them in any detail so far.

3.2.7 Undo set approach

We have presented our technique in terms of a write-set-based STM implementation, in which transactions maintain to-be-written values in their write sets, and copy these back to the memory locations upon successful commit. An alternative approach is to store new values directly in the affected memory location and maintain an "undo set", which records old values to be restored in case the transaction fails. The undo set approach has some significant advantages, because it obviates the need for reads to look up the write set (they can go straight to the affected memory location), and also it makes committing a transaction cheaper at the cost of more overhead for aborting, which is generally expected to be rarer [22].

The undo set approach also has a significant *disadvantage* when implemented in a blocking manner. In the simple blocking scheme based on write sets that we described earlier, a transaction that owns an orec while it is committing blocks other transactions. In contrast, in a blocking undo-set-based STM, transactions that require access to an orec that is owned by an *active* transaction must block. Because active transactions can be executing user code, while committing transactions are only executing STM implementation code, this leaves us much less control over the blocking, to an extent that we find unacceptable. If we can eliminate such blocking, then we can harness the significant advantages of an undo-set-based approach. It is straightforward to adapt our approach to implement a nonblocking undo-set-based STM.

4 Related work

Ensuring nonblocking progress in STMs without using specialized hardware or operating system support is a hard problem. Existing proposals for nonblocking STMs [6, 8, 13, 19] incur significant common case costs. For reasons explained earlier, we focus in this paper on word-based STMs; to our knowledge, the only previous nonblocking word-based STM is Harris and Fraser's WSTM [6, 8]. Below we describe WSTM, and explain the sources of overhead in it that we have eliminated. In the next section, we show that these improvements indeed translate to significantly improved performance.

Like the STMs described in this paper and in [2, 21], WSTM uses a hash table of ownership records (`orecs`). Each location in the shared memory hashes to one `orec`. An `orec` contains either a version number or a pointer to the transaction descriptor of the transaction that most recently acquired ownership of that `orec`. A transaction optimistically accesses locations and their corresponding `orecs`, without acquiring the `orecs`, while recording which locations are accessed (and their `orecs`). Before committing, a transaction uses CAS to acquire the `orec`s for the locations it has optimistically accessed. Then the transaction verifies its consistency and atomically commits. After committing, the transaction copies back new values from its transaction entries to the corresponding locations, and releases each `orec` individually, again using CAS. Thus, a transaction that accesses N `orec`s incurs at least the cost of $2N + 1$ `CAS`'s.

As in our STMS, a transaction in WSTM can simply abort a conflicting active transaction, but a more sophisticated mechanism is required to ensure nonblocking execution if a transaction conflicts with a committed transaction. Such conflicts are resolved using a stealing mechanism, which is superficially similar to the one we have presented, but the details are quite different.

In the WSTM stealing scheme, a stealer steals the `orec` from a victim transaction, copies the victim's transaction entries corresponding to the `orec` into its own write set, and increments a "reference count" in that `orec`. Each transaction that needs to release an `orec` does so by decrementing the `orec`'s reference count. If the reference count falls to 0, the last stealer writes back an incremented version number to the `orec`, thus releasing ownership of the `orec`. Before releasing a previously acquired `orec`, a transaction that finds that the `orec` was stolen by some other transaction follows the transaction pointer in the `orec` and *redoes* all the updates that the new stealer may have committed. This strategy ensures that if a victim transaction performs a "late" write to a location, then this value will be overwritten with the most up-to-date value before the ownership of the `orec` is released.

Our stealing scheme eliminates or reduces several sources of common-case overhead in WSTM:

- In our STM the number of `CAS`'s required for a transaction that acquires N `orec`s and does not encounter any conflicts is just $N+1$, whereas WSTM transactions always require at least $2N + 1$ `CAS`'s.

- WSTM allocates a new transaction descriptor for every transaction attempt. Allocation and reclamation of these descriptors introduces significant complication and common-case overhead. By reusing transaction descriptors, we eliminate this overhead.

- In WSTM, if a transaction $T1$ that intends to access an `orec` $O1$ notices that $O1$ is already acquired by another transaction $T2$ that has aborted, $T1$ has no means of knowing whether $T2$ stole $O1$ from some other transaction in the first place. Thus $T1$ must conservatively copy the transaction entries in $T2$'s descriptor to its own descriptor. In our STM, the `stolen_orec` bit eliminates such unnecessary copying.

- In WSTM, during heavy contention (particularly on contention hotspots), multiple transactions may engage in copybacks to the same set of memory locations repeatedly, resulting in significant bus traffic. By maintaining the invariant that at most one transaction copies back updates to locations corresponding to a given `orec` at a time, we eliminate this redundant work.

- WSTM transactions maintain old and new values of locations as well as their `orec`'s old and new version numbers. When there is no stealing, our STM stores just the old value (for locations read) or the new value (for locations updated) and does not store copies of the `orec` at all.

5 Performance evaluation

We have compared our blocking STM, our nonblocking STM, and the best previous nonblocking word-based STM, namely WSTM [6, 8], using some simple microbenchmarks on two multiprocessors:

- a 16-processor Sun Fire 6800, a cache-coherent multiprocessor with 1.2GHz UltraSPARC® III processors
- a 144-processor Sun Fire E15K, a cache-coherent multiprocessor with 1.5GHz UltraSPARC® IV+ processors (72 dual-core chips)

We observed quantitatively similar results on both machines; here we present only the results from the larger machine to demonstrate the scalability of our implementations. All code was compiled using the gcc compiler, version v3.4.4, with optimization level -O3. In each test, we vary the number of threads from 1 to 64, and measure the time taken to for all threads to perform 100,000 operations each. We report throughput as the total number of operations completed per second; each point is the average over three runs.

All of our STMs use a simple capped exponential backoff scheme to decide whether to abort conflicting active transactions. For decisions about whether to steal from a conflicting committed transaction in our nonblocking STM, we tested two configurations. The first, called `NoStealing`, is the nonblocking STM configured to always decide not to steal. This configuration evaluates the common-case cost of having the *ability* to make progress despite the preemption of a conflicting transactions, even if this option is never exercised. The second, `ImmediateStealing`, steals immediately as soon as it encounters a conflict. Experience with earlier versions of our STM showed that the best performance is achieved by more dynamic policies that wait a while to allow the conflicting transaction a chance to complete, but will steal from the transaction if there is a long delay, for example because it has been preempted. We have not yet experimented with such policies with the latest version used to produce the results presented here.

5.1 Benchmarks

We have conducted experiments using four benchmarks. The first pair (`BST` and `HashTable`) represent somewhat realistic scenarios, accessing meaningful data structures in a way that can reasonably be expected to perform well. The second pair (`Counter` and `ArrayCounter`) are intended as "torture tests", to evaluate how the STMs perform in high contention scenarios.

In the `BST` benchmark, each operation chooses to insert, delete, or lookup a random element in a concurrent set implemented using a binary search tree. Operations are chosen according to a 10%/10%/80% distribution for insert/delete/lookup, and the element is a value between 0 and 255 chosen randomly.

The `HashTable` benchmark is the same as `BST`, except that we use a hash table arranged as a fixed number of buckets, each of which contains a linked-list of of elements hashing to that bucket.

In the `Counter` benchmark, each operation increments a single counter. This benchmark evaluates the STMs in face of a heavily contended hotspot. In the `ArrayCounter` benchmark, there are 16 counters and each operation increments all 16 counters in order. This benchmark examines another scenario in which all transactions conflict with each other, but modify a larger number of locations.

5.2 Results

Figure 3 shows our results, which support the following observations.

- For `BST` (Figure 3(a),(b)), all STMs scale reasonably well, as they should in a benchmark with relatively few conflicts, up to a point. The blocking STM significantly outperforms WSTM, the previous best nonblocking word-based STM, as most people would expect. However, our nonblocking STM performs comparably with the blocking one, with the stealing policy making relatively little difference, as would be expected with few conflicts. This supports our belief that nonblocking STMs can be made to perform comparably with blocking ones in the common case.

  As the number of threads increases, all implementations begin to degrade because the number of conflicts increases. The WSTM implementation has some limitation so that we could not test it beyond 64 threads (probably not fundamental, but we have not investigated yet). While the performance of our STMs degrades with more threads, it maintains reasonable throughput up to 256 threads (Figure 3(b)). The sudden increase in throughput at 256 threads is probably due to a lucky cache mapping due to having 256 threads; we have not investigated yet.

- For `HashTable` with 16 buckets (Figure 3(c)), we see qualitatively similar results to `BST` except that the margin by which WSTM is outperformed by the others is not as dramatic. (For this and subsequent experiments, we focus only on the results up to 64 threads, as this is of most interest in comparing to WSTM. In all cases, our STMs maintain throughput up to 256 threads; see the appendix.)

  Using 256 buckets largely eliminates conflicts, allowing all STMs to perform significantly better. However, WSTM fails to scale as the others do. We believe this confirms our intuition that the overhead of allocating and reclaiming transaction descriptors limits WSTM's performance.

- For `Counter`, the blocking STM performs more than twice as well as WSTM, while the nonblocking variants perform noticably better than WSTM. The results for the nonblocking variants clearly show the cost of stealing too readily, as discussed above. There is also a noticable margin between the nonblocking STM with no stealing and the blocking STM. This probably implicates the small amount of additional per-transaction bookkeeping required by the nonblocking STM, given that the transactions are very small in this benchmark. We have some ideas for further optimisation in this regard.

- The results for `ArrayCounter` are qualitatively similar to those for `Counter`, except that the nonblocking STM variants perform closer to the blocking one, suggesting that the larger transaction size makes the per-transaction bookkeeping mentioned above less relevant. We note that the blocking STM outperforms WSTM by a factor of almost 6 in the single-threaded case, and our new nonblocking STMs very nearly match it.

5.3 Discussion

Our results unambiguously support our claim that there is ample room for improvement over the best previous nonblocking word-based STMs. We have forgone a number of optimisation opportunities and ideas in our work to date, so we believe that there is room for further improvement still.

Our results show that it is possible to significantly close the existing gap between blocking and nonblocking STMs in the common case. Thus, our results and further improvements on them contribute to an understanding of the price to be paid for choosing a nonblocking STM over a blocking one in practice, even before the question of a fundamental gap is settled.

Experimental results cannot settle the question of whether a fundamental gap exists between nonblocking and blocking STMs any more than anecdotes and intuition can; such questions can only be answered by formal proofs. However, our techniques and results can help to provide guidance for researchers interested in determining whether a fundamental gap exists, and in determining what measures are and are not promising for pursuing such a gap.

6 Concluding Remarks

We have presented a new mechanism for implementing nonblocking word-based STM without using hardware or other system support beyond what is available in today's systems. Our STM closely follows a simple blocking STM as long as transactions never decide to steal ownership of memory locations from a committed transaction (an option that is not available in the simple blocking STM). We thus demonstrate that the benefits of nonblocking execution need not come at the cost of significant overhead in the common case, as it does for previous nonblocking STMs.

In addition to the direct benefits of our contribution, it also serves to narrow the gap between the best known blocking and nonblocking word-based STMs, thus providing valuable input to researchers interested in determining whether a fundamental gap exists between blocking and non-blocking STMs. We conjecture that indeed some such gap does exist, but that it will not imply a significant performance difference in the common case. We further conjecture that the gap narrows or disappears given stronger hardware support for synchronization, such as transactional memory.

We emphasise that the advantages of being nonblocking are not only related to performance concerns. For example, the design of an interrupt handler can be severely complicated if the handler shares data with the interrupted thread via a blocking mechanism. STM has the potential to substantially simplify this task, but only if the implementation is nonblocking.

We have presented our ideas in the context of a very simple blocking STM that does not reflect the many optimizations and different design points emerging in the literature. Nonetheless, we believe that the ideas we have demonstrated can be adapted to a wide variety of blocking STMs, including recent optimized STMs, to provide an alternative to waiting for a delayed thread while not significantly impacting common-case performance. Future work includes adapting our ideas to more sophisticated STM designs, improving our stealing heuristics, and more comprehensive evaluation of our STM using more benchmarks and comparing against a wider variety of blocking STMs.

Future architectures can dramatically simplify the task of designing a nonblocking STM through simple hardware support. For example, it is easy to implement a nonblocking copyback mechanism in a system that provides hardware support for atomically confirming the version number of a transaction while copying back a value to memory from its write set.

7 Acknowledgments

We are grateful to members of the Scalable Synchronization Research Group at Sun Labs, especially Yossi Lev, for useful conversations. We also thank Guy Delamarter and Brian Whitney for their help with access to machines, and Dan Nussbaum for his help with running experiments.

References

[1] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded Transactional Memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, February 2005.

[2] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Dan Nussbaum. Hybrid Transactional Memory. In *Proceedings of the 12th Symposium on Architectural Support for Programming Languages and Operating Systems*, October 2006.

[3] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *Proc. International Symposium on Distributed Computing*, 2006.

[4] Dave Dice and Nir Shavit. What really makes transactions faster? In *Transact '06 workshop*, 2006.

[5] Robert Ennals. Software transactional memory should not be obstruction-free, 2005. http://www.cambridge.intel-research.net/ rennals/notlockfree.pdf.

[6] Keir Fraser and Tim Harris. Concurrent programming without locks, 2004. http://www.cl.cam.ac.uk/netos/papers/2004-cpwl-submission.pdf.

[7] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proc. 31st Annual International Symposium on Computer Architecture*, June 2004.

[8] Tim Harris and Keir Fraser. Language Support for Lightweight Transactions. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 388–402, 2003.

[9] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, 2005.

[10] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, page to appear, June 2006.

[11] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

[12] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, 2003.

[13] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for supporting dynamic-sized data structures. In *Proc. 22th Annual ACM Symposium on Principles of Distributed Computing*, pages 92–101, 2003.

[14] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

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

[16] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, March 2006.

[17] Sean Lie. Hardware support for unbounded transactional memory. Master's thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, May 2004.

[18] Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Adaptive software transactional memory. Technical Report TR 868, Computer Science Department, University of Rochester, May 2005.

[19] Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Adaptive Software Transactional Memory. In *Proceedings of the 19th International Symposium on Distributed Computing*, 2005.

[20] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. Lowering the overhead of nonblocking software transactional memory. In *Transact '06 workshop, to appear*, 2006.

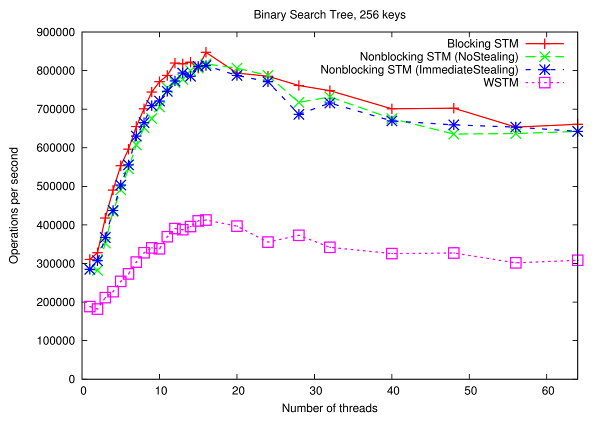
[21] Mark Moir. Hybrid Transactional Memory, July 2005. `http://research.sun.com/scalable/pubs/Moir-Hybrid-2005.pdf`.

[22] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In *Proc. 12th Annual International Symposium on High Performance Computer Architecture*, 2006.

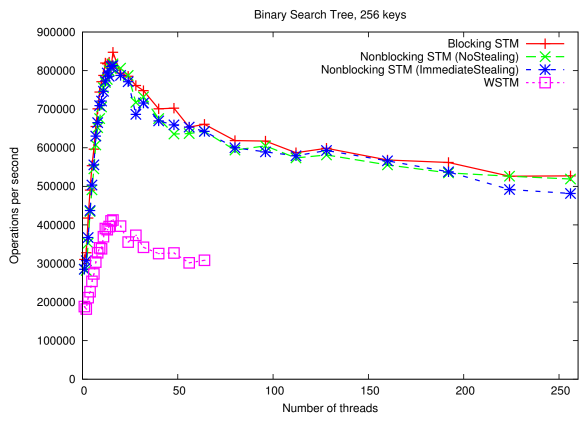
[23] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *Proc. 32nd Annual International Symposium on Computer Architecture*, pages 494–505, Washington, DC, USA, 2005.

[24] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197, 2006.

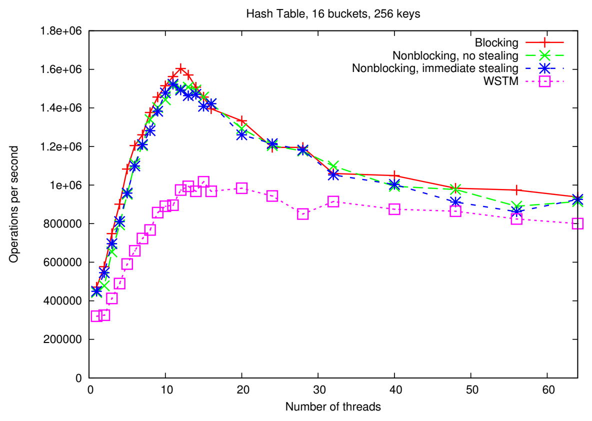
[25] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, Special Issue(10):99–116, 1997.



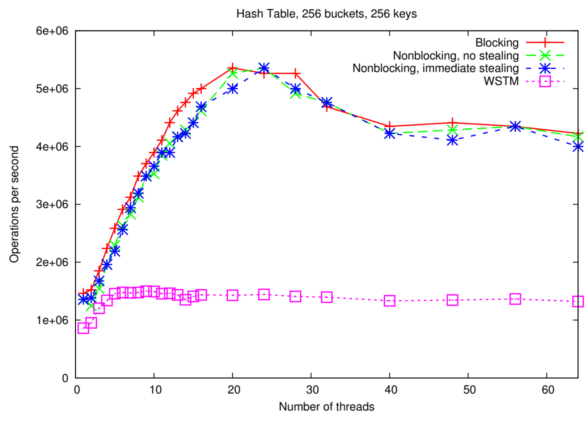
(a)



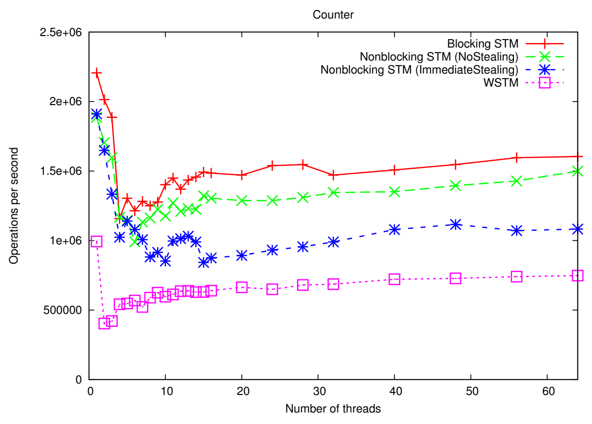
(b)



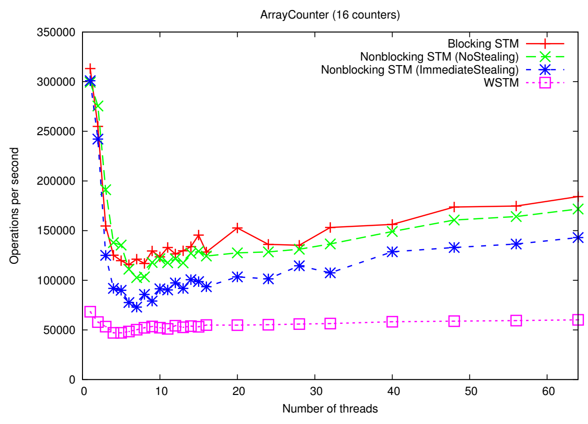
(c)



(d)



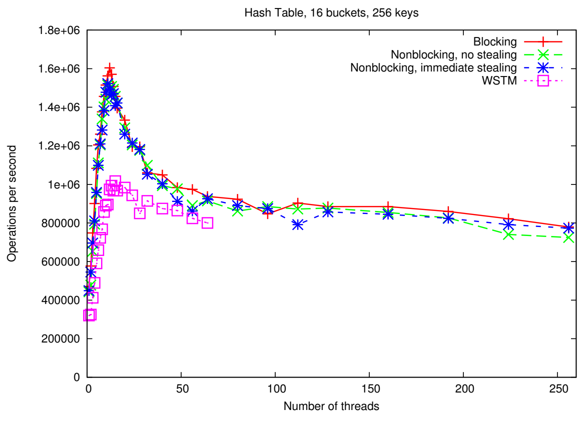
(e)



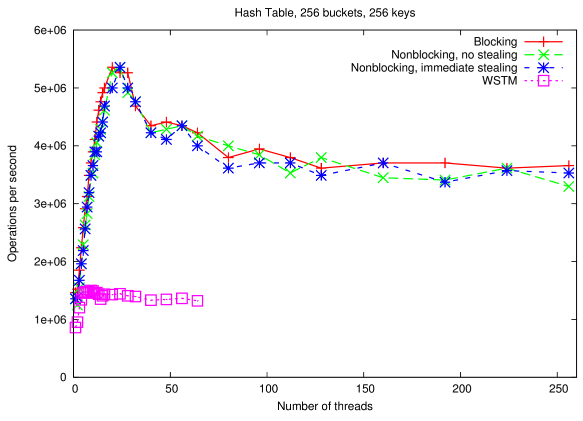
(f)

Figure 3: (a) Binary search tree (b) binary search tree up to 256 threads (c) hash table, 16 buckets (d) hash table 256 buckets (e) counters (f) array of 16 counters.

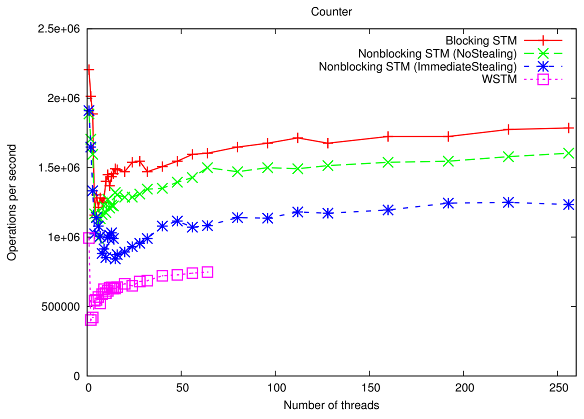
A Additional performance graphs



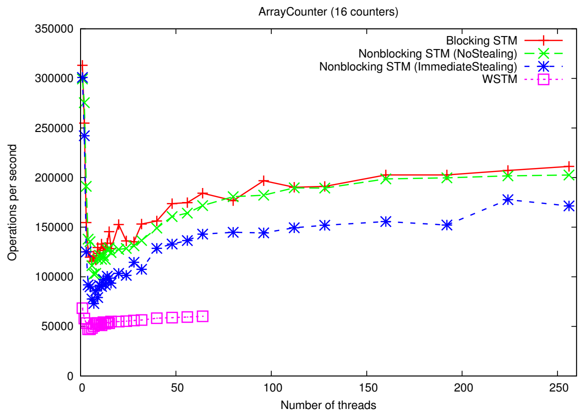
(a)



(b)



(c)



(d)

Figure 4: (a) Hash table, 16 buckets (b) hash table, 256 buckets (c) counter (d) array of 16 counters.