Toward High Performance Nonblocking Software Transactional Memory *

Virendra J. Marathe
University of Rochester
vmarathe@cs.rochester.edu

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

Abstract

Substantial advances in STM performance in recent years have mostly focused on blocking systems. We describe our work integrating the most important techniques and optimizations emerging from the recent work on blocking STMs into several variants of a nonblocking STM.

In particular, our design is based on the philosophy of keeping the common, contention free execution path as simple (consequently fast) as possible, while resorting to the more expensive data displacement and metadata management only in situations where transactions have problems making forward progress. We employ novel ownership "stealing" and metadata management techniques in our nonblocking STM to enable several recent blocking STM optimizations such as timestamp-based validation and ownership release via store instructions, all leading to a more streamlined and efficient fast path. We present an *undo log* (eager versioning) variant of our STM, as well as two *redo log* (lazy versioning) variants, the latter of which are based on the two ownership acquisition techniques (namely *eager* and *lazy*) for writes made by transactions.

Experimental results show that our efforts have improved the performance of nonblocking STMs up to the level of being competitive with the state-of-the-art blocking STMs such as TL2.

Categories and Subject Descriptors [*D.1.3 Concurrent Programming*]: Parallel Programming

General Terms Algorithms, Performance

Keywords software transactional memory, nonblocking

1. Introduction

Transactional Memory (TM) is a concurrent programming abstraction that promises to simplify the task of writing parallel programs. TM allows programmers to express *what* should be executed atomically, leaving the system to determine *how* this atomicity should be achieved. Herlihy and Moss [11] proposed hardware transactional memory (HTM) and Shavit and Touitou [24] proposed software transactional memory (STM). A recent flurry of activity in both HTM (e.g., [1, 5, 18, 19]) and STM (e.g., [3, 6, 8, 10, 16, 22]) has yielded substantial progress towards making TM practical.

Foundational work on TM 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.

Recently, many researchers have developed *blocking* STMs [3, 8, 22], recognizing that they are much easier to design and that most of 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, as pointed out by Ramadan et al. [20], in their TM enabled TxLinux kernel, it is *unacceptable* for an interrupt handler to be blocked by the thread it has interrupted.

The nonblocking STMs described in this paper are *obstruction-free* [9]: they guarantee that, if a transaction is repeatedly retried and eventually encounters no interference from other transactions, then eventually the transaction commits successfully. Obstruction-freedom does not make any progress guarantees and admits "livelocks". An out-of-band *contention manager* [10, 23] can be used in practice to eliminate such undesirable situations.

In this paper we present our algorithms in a *word-based* STM setting, where conflict detection is at the granularity of contiguous blocks of memory. This setup is important for unmanaged environments such as C and C++ since the STM cannot dictate the layout of program data, for example to colocate transactional metadata with the program data it mediates. Word-based STMs keep transactional metadata (often called *ownership records – orecs* in short) separate from program data; Tabba et al. [27] describe preliminary work on applying our initial design approach [14] to achieve efficient nonblocking *object-based* STMs.

The only other nonblocking word-based STM of which we are aware is Harris and Fraser's system (HF-STM)[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. As the results in Section 6 show, we have been able to dramatically improve performance over HF-STM and achieve performance that is competitive with state-of-the-art blocking STMs.

1.1 Our Contributions

We make the following key contributions in this paper.

- We are the first to almost entirely decouple the contention free fast path of a nonblocking STM from the complicated data displacement and metadata management required for forward progress.
- We have successfully integrated key optimizations appearing in recent state-of-the-art blocking STMs into our nonblocking STM. This includes timestamp based transaction validation and

* At the University of Rochester, this work was supported in part by NSF grants CNS-0411127, CNS-0615139, and CCF-0702505.

Copyright is held by Sun Microsystems, Inc.
PPoPP'08 February 20–23, 2008, Salt Lake City, Utah, USA.
ACM 978-1-59593-960-9/08/0002.

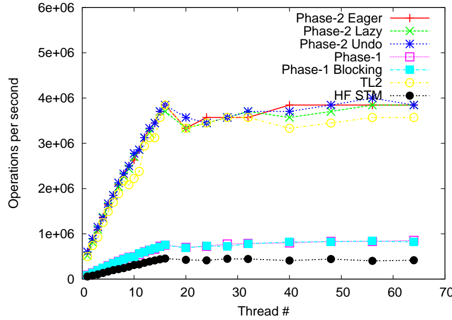

Figure 1. Binary Search Tree with 32K nodes. Notice the progressive improvement in performance of our nonblocking STMs.

simple store instruction based ownership release, both enabling a more streamlined fast path.

- We present an undo log based nonblocking STM. To our knowledge, this is the first such nonblocking STM. We also present some interesting performance tradeoffs between redo and undo logs for heavily contended workloads.

1.2 Overview of design approach

The work reported in this paper proceeded in two phases. Our philosophy for the first phase was to mimic behavior of blocking STMs as far as possible, and resort to the more expensive data displacement and metadata management only in situations where transactions have problems making forward progress. This approach effectively decouples almost all of the nonblocking progress related metadata management from the fast path, thus yielding performance comparable to the blocking STM in the common case.

In the summer of 2005, we took a simple blocking STM similar to the one described in [2], and designed an obstruction-free STM that closely tracks the blocking scheme (particularly with respect to metadata structure and cache behavior) until the contention manager decides that a transaction should not wait for another to complete. In a blocking implementation, there is no choice but to wait in such circumstances: the mechanism dictates the policy.

For our Phase-1 nonblocking STM, we introduced the ability to "steal" ownership of a memory location from another transaction, rather than waiting for the other transaction to explicitly release it. Accessing stolen locations is more complicated and expensive than accessing unstolen ones, but nonetheless stealing is worthwhile in order to avoid waiting for another transaction that is delayed for a long time, for example due to preemption. Additionally, our design focused on quickly switching the stolen locations back to the unstolen state so as to minimize the overhead of occassional stealing that happened due to high contention.

Our Phase-2 effort focused on incorporating optimization techniques, such as timestamp based transaction validation [3, 21], from state-of-the-art blocking STMs into out nonblocking STM. Figure 1 demonstrates the significant improvements rendered by our Phase-2 enhancements. Clearly, our nonblocking STM is competitive with one of the leading blocking STMs, TL2 [3]. It therefore seems premature to dimiss nonblocking STMs as fundamentally performing worse than their blocking counterparts.

1.3 Roadmap

The remainder of this paper is structured as follows. We give an overview of STM designs in Section 2. Section 3 describes our simple Phase-1 blocking STM and the modifications we made to make it nonblocking. Section 4 makes a qualitative argument about the salient features of state-of-the-art blocking STMs that primarily contribute to drastic performance improvements. In Section 5 we describe extensions to our Phase-1 design that yielded substantial improvements in performance. We also describe three variants of our Phase-2 algorithm: two variants, based on eager and lazy ownership acquisition techniques, of a redo log version; and an undo log version. In Section 6, we present experimantal results that compare all flavors of our Phase-2 nonblocking STM with the Phase-1 blocking and nonblocking STMs, the nonblocking STM by Harris and Fraser [6], and TL2, a state-of-the-art blocking STM. Our results demonstrate that our design approach has closed the performance gap between blocking and nonblocking STMs. Some discussion appears in Section 7, and we conclude in Section 8.

2. Background on STMs

A transaction speculatively reads and writes memory locations, and attempts to commit at the end, either "succeeding" as if the entire transaction executed atomically, or "failing" as if the transaction did not execute at all.

To provide the illusion that a successful transaction commits atomically, STM systems generally acquire *ownership* of all memory locations modified by the transaction in order to prevent concurrent transactions from observing partial updates. This ownership is coordinated through special *metadata* associated with program data.

Some STMs [2, 3, 6] buffer writes into a private write set during transactional execution, and copy the values to the affected memory locations upon successful commit; such STMs are sometimes referred to as *redo log* STMs, because they "redo" the writes upon commit. Other STMs [8, 22, 28] instead store speculatively written values directly into the affected memory locations, maintaining an *undo log* that preserves overwritten values so that they can be restored in case the transaction aborts. This approach is attractive because it optimizes for the hopefully common case in which most transactions commit successfully: there is no need to redo writes upon commit, and there is no need for transactional reads to search the write set for values previously written by the transaction.

Ownership acquisition can be either "eager" or "lazy"[15]. With eager acquisition, a transaction acquires ownership of a location when it first writes to the location. In contrast, with lazy acquisition a transaction acquires ownership of locations it has speculatively written only at commit time. Because undo log STMs store speculatively written data directly in the affected memory locations during transactional execution, they must acquire ownership eagerly in order to prevent other transactions from observing values written by a transaction that may subsequently abort. Redo log STMs can use either eager or lazy acquisition.

To guarantee isolation of transactions, the STM must ensure that reads done by transactions are always mutually consistent. This is achieved by verifying that all read locations have not been modified since. This transaction "validation" can be accelerated with *timestamp* based techniques [3, 21, 28]. We have incorporated timestamp based validation in all our nonblocking STMs.

After committing or aborting, a transaction releases ownership of locations it has acquired. In blocking STMs, a transaction must wait for a conflicting committed or aborted transaction to release ownership of the locations under conflict.

3. Phase-1: Decoupling Nonblocking Progress Related Work from the Fast Path

We first describe the simple Phase-1 blocking STM, which is based on the one described in [2] and includes some novel optimizations. We then explain how we modified this STM to be nonblocking,

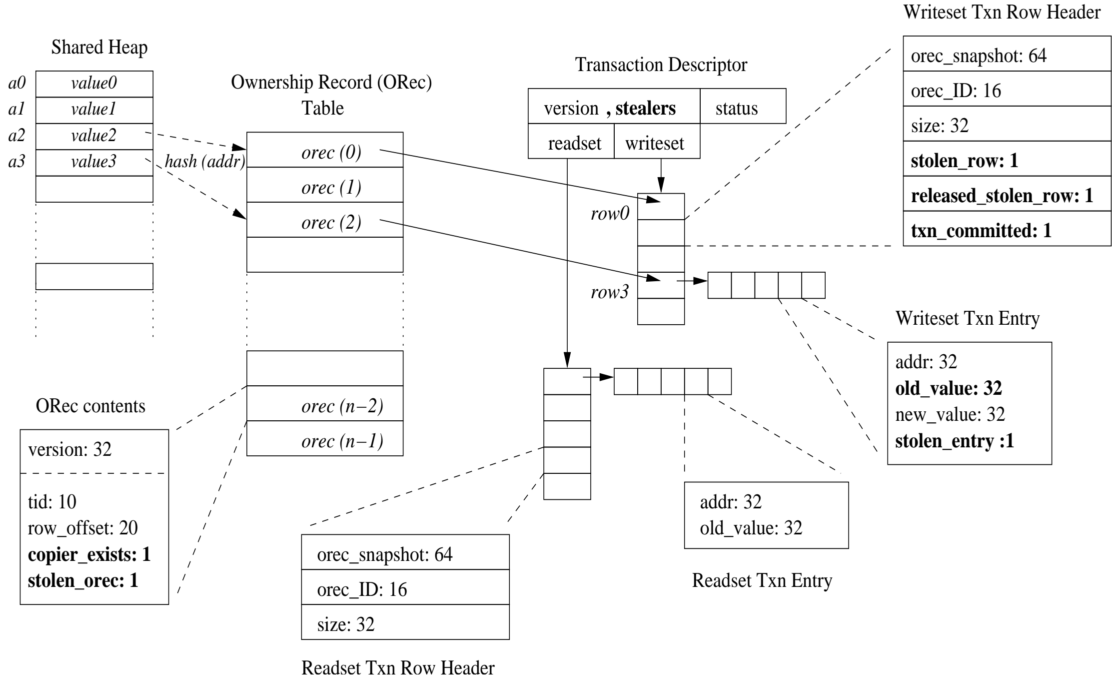

Figure 2. Data Structures of Nonblocking/Blocking STM. Fields in bold are required for the nonblocking version only.

while keeping the common case as close to the blocking STM as we could in order to achieve a similar fast path.

3.1 STM API

Out STM API includes the following calls, similar to that of other word-based STMs.

```
stm_begin(TxnDescriptor* my_txn)
stm_commit(TxnDescriptor* my_txn)
Word_t stm_read(TxnDescriptor* my_txn,
                Word_t* addr)
void stm_write(TxnDescriptor* my_txn,
               Word_t* addr,
               Word_t value)
```

3.2 Data Structures

The primary data structures in our STMs are the *transaction descriptor*, which is used to represent a transaction, and a table of *ownership records* (orecs), which are used to represent ownership by transactions of memory locations. A many-one hashing function maps memory locations into the orec table.

A transaction descriptor consists of a transaction ID (tid), a version number (version), a status (Active, Committed, or Aborted), and read and write sets. The tid and version uniquely identify a transaction. Thus, by incrementing the version number in its transaction descriptor, a thread can reuse the transaction descriptor.

The read and write sets are organized as per-orec rows, such that all entries relating to memory locations that map to the same orec are stored in the same row. Each row contains an orec identifier, a snapshot of the orec with which it is associated, and an array of *entries*, each of which is an address-value pair of some address covered by the indicated orec. On a read/write operation, if the transaction does not already contain a row used for the identified orec, it uses the next available row in its read/write set for the orec. Figure 2 depicts the structure of a transaction descriptor.

Each orec is stored in one 64-bit word, and is atomically modified using a CAS. Each orec consists of: tid and version fields, used to identify the current owner transaction; and a row field, to identify the row of the owning transaction's write set in which it stores entries for locations mapping to that orec.

Embedding the owner transaction's tid and version in the orec enables a novel *fast release* optimization, wherein the transaction may simply increment its version to implicitly release ownership of orecs it owns. This eliminates from a transaction's fast path the overhead of explicitly releasing orecs using expensive CAS instructions (as in HF-STM [6]). Note that this design decision differs from our Phase-2 design described in Section 5.

3.3 Simple Phase-1 Blocking STM

The blocking STM transaction executes in the Active state and uses the write set as a redo log. Orecs are acquired eagerly. To ensure consistency at all times, a transaction validates its entire read set every time a new orec is accessed. This step is particularly necessary for unmanaged languages such as C/C++ since temporary inconsistencies in "doomed-to-abort" transactions may cause arbitrary, irrecoverable program behavior. (Timestamp-based validation techniques [3, 21] were not invented when we designed this algorithm.) To commit, a transaction atomically switches its status from Active to Committed. Finally, the transaction copies the values from its write set (redo log) to the affected memory locations, and then fast releases the orecs it owns by incrementing its version number.

If a transaction T intends to access a location already owned by some committed transaction S, T must wait for S to release the corresponding orec. Given the use of redo-logging, T does not need to wait if S has aborted.

3.4 Making the STM Nonblocking

Our nonblocking STM allows a transaction to "steal" ownership of 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.

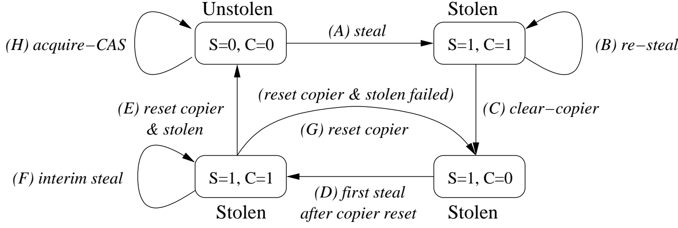

Figure 3. Transitions among the unstolen and stolen states of an orec. S and C represent the values of the stolen_orec and copier_exists flags in the orec. All transitions use an atomic CAS.

In the remainder of this section, we briefly describe the stealing mechanism that we incorporated to make our Phase-1 STM non-blocking, with reference to the state transition diagram in Figure 3. This mechanism is presented in more detail, along with a detailed example, in [14].

A transaction T steals ownership of an orec (transition A) from another transaction by CASing the orec to point to a row in T's write set (T uses this row to store its subsequent speculative updates to locations mapping into the orec). Although this seems straightforward, some more bookkeeping must be done to ensure that the *logical values* of locations mapping into the stolen orec are correctly preserved during the stealing process.

Note that orec stealing happens if a transaction (the "stealer") encounters an orec, O, owned by a committed transaction (the "victim"). This means that the victim is still in the process of copying back its speculative updates to the locations in its write set. The logical values of locations mapping into O may thus reside in the victim's write set row, R_v, that O points to before it is stolen by the stealer. To correctly preserve this view of memory, the stealer must first "merge" R_v into an available row, say R_s, in its write set. Successful stealing will make O point to R_s.

During stealing, the victim is possibly in the process of copying back updates from its redo log (more specifically from R_v) to the locations mapping into O. This "copyback" could be delayed arbitrarily. As a result, a situation may arise where the stealer, after stealing ownership of O, completes before the victim finishes its copyback phase. In this case, the logical values of locations mapping into O reside in the stealer's redo log (more precisely in R_s). If the stealer were to copyback its updates to O's locations and fast release O, there is a possibility that the victim's now "stale" updates may overwrite the stealer's more recent updates. To avoid such a race, we introduce a new stolen_orec flag, represented as the S flag in the state transition diagram, in an orec (see Figure 2). This flag indicates that orec O is in the stolen state, and logical values of locations mapping into O may reside in the write set row that it points to (R_s in our example).

As long as the stolen_orec flag for O is true, the logical values of locations mapping into O may reside in the write set row (R_s in our example) that O points to. Consequently, R_s cannot be reused by a subsequent transaction executed by the same thread using the same transaction descriptor; R_s may be reused only when O no longer points to it. We use a stolen_row flag (see Figure 2) in a write set row to indicate that a stolen orec, O, points to that row (R_s in our example). The stolen_row flag is set during the stealing process. Any subsequent transaction that "re-steals" O (transition B) must clear the stolen_row flag of R_s. Resetting of the stolen_row flag happens indirectly, in that the new stealer sets the released_stolen_row flag in R_s (Figure 2), which is then used by the victim to "reclaim" R_s for subsequent reuse.

Logical values of a location l mapping into a stolen orec O may reside in the write set row that O points to provided there exists an entry for l in that row. Otherwise the logical value is at l itself.

A stealer is usually in Active state during the stealing process. Thereafter the stealer could either commit or abort. If it commits, its redo log updates hold the new logical values of locations mapping into O. However, if it happens to abort, its redo log updates must be discarded and the "old" (stolen) values must be retained. This requirement of retaining old values if a stealer happens to abort makes maintainance of old and new values of stolen entries mandatory (Figure 2). Thus, if the stealer commits, the new values become the logical values of the locations; otherwise the old values remain the logical ones. To indicate that a transaction committed its changes to a stolen_row, we add a new flag, txn_committed to a write set row (Figure 2), which is set to true if the stealer managed to commit, and false otherwise.

Note that since the logical values of locations mapping into a stolen orec O are retained in the write set row that O points to, there is no need for stealers to copy back their updates to locations mapping into O; only the first victim is copying back its updates. (In general we maintain the invariant that at any given time, at most one transaction is copying back committed updates to locations mapping into a given orec.) After the first victim finishes its copyback, it verifies that all orecs it owns were not stolen.[1] If an orec O was stolen, the victim has the opportunity to inform the system that it has finished its copyback for O.

Access to stolen locations is expensive. Hence it is important to switch an orec back to unstolen state as quickly as possible. Since the first victim is the only transaction doing a copyback of locations mapping into O, using a single copier_exists flag in O suffices to inform the system that the lone copier of O has finished: The first stealer that steals O sets both the stolen_orec and the copier_exists flags (transition A). The first victim, after its copyback, resets the copier_exists flag indicating to the system that it has completed its copyback (transition C). This state of the orec gives the system an opportunity to safely switch the orec back to the unstolen state as follows: At this point there exists no transaction copying back updates to locations mapping into O. As a result, the next stealer of O can safely assume the responsibility of doing the copyback (by setting the copier_exists flag during the stealing CAS, transition D). It copies back the most recent logical values of locations mapping into O in its write set row, and subsequently resets both the stolen_orec and copier_exists flags using a CAS (transition E). This switches O back to the unstolen state.

Note that the stealer's CAS to clear O's stolen_orec and copier_exists flags may fail if another transaction steals O in the interim (transition F). In such a situation, the stealer resets O's copier_exists flag (transition G), thereby handing over the copyback responsibility to a future stealer.

Read Sharing Our implementation permits read sharing, even for stolen orecs. A reader must ensure that the stolen orec is not concurrently modified by a stealer, and retrieve the logical values of the target locations from the write set row associated with the stolen orec. The reader simply maintains a snapshot of the orec, and read validation is achieved by ensuring that all of the orecs from which the transaction has read still match the previous snapshots. We ensure that the reader maintains exactly one copy of each orec read in its read set.

[1] This is done with a stealers list in the transaction descriptor (Figure 2) that is atomically accessed with its version field. The stealer atomically adds itself in the stealers list to inform the victim about theft of an orec. Our technical report [14] details all these design aspects.

4. What Makes Blocking STMs Fast?

At the heart of the significantly better performance of recent blocking STMs (see curves for our Phase-1 STMs and TL2 in Figure 1) is their simplicity. This simplicity, combined with a number of optimization techniques [3, 8, 22, 28] has yeilded significant improvements in STM performance. In this section we briefly discuss what we believe to be the key features that make blocking STMs so fast.

(a) Streamlined Fast Path: Recent state-of-the-art blocking STMs have been carefully engineered to make the fast path for speculative reads and writes as efficient as possible. Among several important design decisions are: (i) simple metadata structure, making common case conflict detection more efficient; (ii) simple ownership acquisition and release operations consisting of a compare-and-swap (CAS) and a store instruction, respectively; and (iii) more streamlined read and write set implementations.

(b) Timestamp-based Validation: Some recent breakthroughs in guaranteeing transaction consistency at a low common case cost (such as timestamp-based transactiona validation [3, 21]) have contributed significantly to performance improvement in blocking STMs. To our knowledge, our work is the first to integrate timestamp-based validation in a nonblocking STM.

(c) Undo logging Capability: Undo log based implementations [8, 22] result in inexpensive reads of locations that have already been modified by the same transaction.

5. Phase-2: Toward Fast Nonblocking STMs

As suggested in Section 4, we believe that incorporating the key optimizations of blocking STMs is sufficient to make nonblocking STMs comparably efficient in the common case, thereby guaranteeing nonblocking progress for almost no extra cost.

5.1 Integrating Recent Blocking STM Optimizations

Our first phase yeilded significant performance improvements over the prior best word-based nonblocking STM, the HF-STM [6] (see Figure 1). However, there were some key optimizations missing from our implementation. Firstly, recent state-of-the-art blocking STMs employ the timestamp-based validation scheme, which significantly mitigates the overhead of transaction validation.

Secondly, although our fast release optimization eliminates the need for explicit ownership release operations (implemented using CASes in HF-STM), it introduces an extra level of indirection, where a transaction T_1 reading an orec O must identify if the transaction O points to, say T_2, is no longer Active. Such an approach not only requires extra instructions on the fast path, but may also lead to expensive bus transactions due to cache misses. A *lazy cleanup* strategy (as in ASTM [16]), wherein the first reader of orec O CASes it to an Unowned state, seems like an attractive alternative. However, lazy cleanup simply moves the ownership release CAS from the owner's commit/abort cleanup time (as in HF-STM) to a later time, and does not seem like a real improvement.

Our second phase addressed both optimizations with some novel, and surprisingly simple extensions to the Phase-1 STM. Specifically, we integrated, in our Phase-1 STM, an inexpensive orec release operation that employs a single word store instruction per acquired orec. This ownership release operation mimics the behavior of current high performance blocking STMs more accurately than fast release. Furthermore, our new ownership release operation facilitates incorporation of the recent timestamp-based transaction validation schemes [3, 21] in our STMs. Specifically, we were able to superimpose a timestamp with the orec version number. With some more simple modifications we were able to build an undo log version of our Phase-2 nonblocking STM.

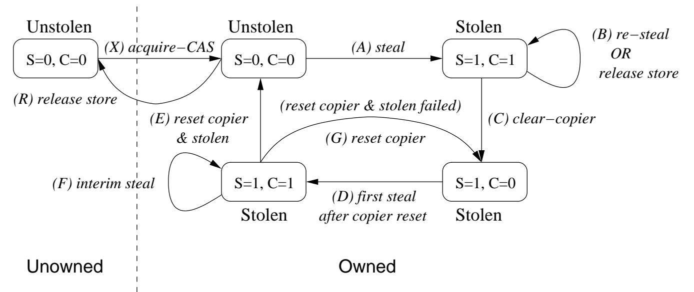

Figure 4. Transitions of the orec state in the Phase-2 nonblocking STM. S and C represent the values of the stolen_orec and copier_exists flags in the orec. Unowned represents the state when the orec's stolen_orec flag is false and the orec's version field contains a timestamp. Owned state represents the orec's state wherein it is either owned by an Active transaction, a Committed transaction (where the transaction is in its copyback phase and is yet to release the orec), or is in the stolen state (although it may not be owned by an Active transaction at some give time). All transitions, except the *release store* transitions, use an atomic CAS.

5.1.1 Timestamp-based Validation

Timestamp-based validation employs a globally shared "clock". When a transaction begins execution it reads the global clock and stores the value in a transaction local begin_timestamp field. This value is used, during the transaction's execution, to determine if the locations accessed by the transaction are mutually consistent. Each orec contains a timestamp field which approximates the "logical time" at which the orec was last modified by a transaction. A transaction is guaranteed to view a consistent version of an orec O if O's timestamp is less than or equal to the transaction's begin_timestamp. This is the simple step of validating a transaction at each shared memory access. The whole read set of the transaction may be revalidated at commit time. At commit time, a writer reads the global time[2] and stores it in the timestamp field of all acquired orecs, thereby releasing them.

5.1.2 Efficient Ownership Release

Our new ownership release operation is based on the observation that we can superimpose the timestamp value and the transaction version number on an orec's version field. Initially an orec's version contains a timestamp. During an acquire CAS, the writer transaction swaps its current version (differentiated from a timestamp by the least significant bit) into the orec's version field. The new ownership release operation of a transaction T is simple — use an ordinary store instruction to overwrite the version in an acquired orec O with the most recent timestamp value in the global clock (accessed by T at commit time). Our key insight is that permitting this behavior, and the already existing representation of the stolen state of an orec, lets us define an unowned state for an orec in a different way. We say that an orec is unowned when it is not stolen (the stolen_orec flag is false) and it contains a timestamp.

Although it may sound straightforward, the new orec release operation has several subtleties that arise in stealing and transaction abort scenarios. As a result, we explain the details of the orec release operation on a case-by-case basis. In our description, we refer to the new state transition diagram in Figure 4.

[2] Several alternatives for scalable clocks in the context of STMs have been proposed recently [3, 21]. We use the implementation available in the TL2 [3] library, where the global clock is a counter that is atomically incremented by a committing transaction.

The Uncontended Case Note that in all cases, except for the orec release operation, the entire orec (two adjacent 32-bit words) is accessed by transactions atomically. An unowned orec is simply acquired by the CAS shown in transition X. Consider an example where transaction T_1 owns orec O; i.e. O contains T_1's ID, the write set row number that contains the speculative updates made by T_1 to locations mapping into O, and the current version of T_1 (both the stolen_orec and copier_exists flags of O are false). If T_1 commits, it first copies back all speculative updates to locations mapping into O and thereafter releases O by storing a new timestamp in O's version field (transition R). Since O's stolen_orec flag is false, the release essentially switches O back to the unowned state.

The Stealing Case Now consider transaction T_2 that intends to acquire O. If T_1 has already released O (via transition R), O is already in the unowned state, and T_2 does not need to steal O. However, if T_1 has not yet released O, T_2 steals O as per our stealing algorithm discussed earlier (transition A). Stealing requires a CAS over the entire orec, which also sets the stolen_orec and copier_exists flags in O. This switches O into the stolen state.

Meanwhile T_1 may finish copyback of its speculative updates and attempt to release O by storing its release timestamp in O's version field, which now contains T_2's version (transition B). Although this release overwrites T_2's version, O still persists in its stolen state since its stolen_orec flag (set by T_2) is still true. (Note that the algorithm ignores the orec's version field contents when the stolen_orec flag is true.) At this point T_1 can verify that O is stolen by another transaction (T_2 in our example).

Since we continue to adhere to our invariant that there can exist at most one transaction doing the copyback for an orec at any given time, T_1 is the only transaction doing the copyback for O. Following our stealing protocol, T_1 can inform the system that it has finished the copyback by resetting O's copier_exists flag (transition C). A subsequent transaction, say T_3, that intends to acquire O, may switch O back to unstolen (albeit owned by T_3) state, as per the state transition diagram in Figure 4 (transitions D followed by E). If these transitions are successful, T_3 can release O with a simple release store (transition R). This portion of the algorithm is identical to the Phase-1 algorithm from Section 3.

There exist two subtleties in ordering of events which may lead to unpredictable behavior in our new stealing algorithm. The first concern is that we cannot guarantee that the store-based orec-release followed by verification that the orec is not stolen both happen atomically. As a result, a situation may arise where transaction T_1 releases unstolen orec O and is delayed arbitrarily before it rereads O to verify that it was not stolen. In the interim, O may be modified several times by concurrent transactions and eventually switch to the stolen state. If T_1 reads O at this point, assumes that it is the victim, and clears the copier_exists flag (which was set for another transaction, say T_2, to clear) the runtime may end up with more than one copier for an orec, a violation of our invariant.

To ensure that the right transaction (T_2 in our example) clears the copier_exists flag of a stolen orec (O in our example), we added a new copier_ID field in the write set row data structure (from Figure 2). The first stealer of an orec sets the copier_ID field in its write set row to the value of the victim's ID. Only when the possible victim's ID is the same as the copier_ID of the write set row that the stolen orec points to, should the victim clear the orec's copier_exists flag. In our example, only T_2 can clear O's copier_exists flag. When a transaction, say T_3, assumes the responsibility of doing a copyback (via transition D), it sets the corresponding write set row's copier_ID to its own ID.

The second source of concern is related to the fact that now an orec's version field contains either a transaction version or a timestamp. Recall that in the Phase-1 STMs the version number in the orec's version field was used to determine if the currently executing version of the transaction descriptor that the orec points to is the one that acquired the orec. In our new algorithm, the stolen orec's version field may be overwritten by a timestamp of the first victim, causing loss of information about the transaction version that actually stole the orec. Our solution was to add another field to the write set row data structure, which we call the stealing_time_version. The stealer transaction, during the stealing process, stores its current version into its target write set row's stealing_time_version field. Any new transactions that access the orec in stolen state use the stealing_time_version field to determine whether the current (or a past) version of the owner owns (or owned) the orec.

The Abort Case Since the Phase-1 STM used redo log and implicit fast release techniques, the operation of acquiring an orec that points to an aborted transaction was not considered as orec stealing. In our new algorithm, we continue to adhere to the policy that acquiring an orec that points to an aborted transaction is not stealing (this policy changes in case of our undo log version, discussed later). However, now an aborted transaction must also explicitly release an acquired orec. The simple store instruction based release operation on an orec O, by an aborted transaction, say T_1, may overwrite the version field of O, which contains the version of O's current owner, say T_2, with a timestamp. Since O's stolen_orec flag is not true, T_1's store illegitimately switches O's state to unowned.

We have a simple solution for this problem – an orec-wide CAS based release. The CAS ensures that the version field of the orec will not be overwritten by an aborted transaction if the orec was already acquired by another concurrent transaction. Since it is reasonable to assume that aborts will be rare, we believe that our solution does not entail significant overhead.

5.1.3 Read and Write Set Implementations

As may be clear from our description, the write set implementation in our new algorithm (except for the newly added fields – copier_ID and stealing_time_version – in the write set row data structure) is more or less the same as our Phase-1 STM.

Realizing that the read set implementation need not be the same as the write set implementation, we modified the read set data structure to a more streamlined version consisting of a linear linked-list of entries superimposed on a list of contiguous blocks of 256 entries each. This structure is very similar to that used in recent blocking STMs [3, 8, 22]. To make the fast path for reads more streamlined, for each new read operation the transaction appends an entry in the read set. Each entry consists of an orec address, which is used at commit time for read set validation. Thus, redundant entries for the same location may exist in the read set. As shown by Harris et al. [8], for long running transactions, these redundancies may be reduced by periodically *filtering* the read set. If the target location's orec is already owned by the transaction itself, the transaction's corresponding write set row is searched for a possibly updated value (by the same transaction) of the target location. This searching is inherent in redo log based implementations, but not in undo log based implementations.

5.2 Lazy Ownership Acquisition

The algorithm presented thusfar uses an *eager* ownership acquisition policy for writes. Modifying the algorithm to support *lazy* ownership acquisition [15] (where the the writer acquires ownership of locations written to at commit time) was quite straightforward. We introduced a linear write list data structure, which was very similar to the streamlined read set data structure (the only difference being that the write list entries contain address-value pairs), to retain the redo log of the transaction. At commit time, the transaction traverses the write list and acquires corresponding orecs. In the process, the transaction also builds up the 2-dimensional write

set (as is in the eager acquire version of the STM). This extra step of replicating the redo log made the lazy acquire STM version easily compatible with our nonblocking stealing algorithm with little extra overhead (shown in Section 6).

5.3 The Undo Log Algorithm

In a redo log based STM, a transaction intending to read an already modified location must refer to its write set for the most recent logical value of the location. This lookup for a *read-after-write* operation is potentially a significant source of overhead in redo log based STMs. In undo log based STMs a writer transaction stores the *old* value of the target location in its write set, and makes a *direct* update to the location. This avoids the lookup required in a subsequent read-after-write operation by the transaction. On the flip side, in redo log based blocking STMs, a transaction that encounters a conflict with an already aborted transaction may safely acquire ownership of the location without waiting for the aborted transaction's release. However, for correctness, a transaction in an undo log based blocking STM must wait for such an aborted transaction to release ownership of the location, which may possibly take an arbitrary amount of time, particularly because the aborted transaction may not notice that it is aborted for a long time. With nonblocking progress guarantees in undo log based STMs one can get the best of both worlds – elimination of read-after-write lookup overhead and of arbitrary waiting during transaction aborts.

The undo log variant of our algorithm was surprisingly simple. In redo log based STMs, stealing is necessary in case a conflicting transaction has already committed since the logical values of all its updates may reside in its write set. The same reasoning applies to conflicting aborted transactions in the undo log variant of our nonblocking STM. Thus, in our STM's undo log variant stealing pre-dominantly happens when transactions abort.

Contrary to our initial impression, in our undo log based nonblocking STM, stealing is necessary even when a transaction encounters an already committed transaction. This is an artifact of our store instruction based orec release policy, which we maintained in our undo log STM. The orec release operation races with the same orec's acquisition by another transaction. Note however, that stealing is needed in this case only to maintain a consistent view of the orec's state. The logical values of the locations modified by the victim already reside in the respective locations. Thus we do not need to merge the victim's write set row into the stealer's write set. All other parts of the nonblocking stealing algorithm remain the same in the undo log STM.

5.4 Fast Path Behavior

As our empirical evaluation (Section 6) suggests, our nonblocking STMs are competitive with state-of-the-art blocking STMs. Most of the credit goes to the fast path behavior of our STMs.

5.4.1 Read Fast Path

In the fast path, a speculative (transactional) read in our nonblocking STM is very similar to that of state-or-the-art blocking STMs – a transaction computes the target location's orec address, reads the entire orec atomically in registers, makes two ownership tests (one to check if the orec contains a version and the other to check if the orec is stolen, either of which indicates that the orec is owned), verifies that the orec's timestamp is consistent, and finally logs the orec in the read set. Figure 5 depicts the fast path in C-like pseudocode form. Of these instructions, our algorithm's fast path differs from recent blocking STMs in the orec size, and an extra test for the stolen orec case, which amounts to slight increase in register pressure (for one extra orec word), and bitwise operations with a comparison (for testing stolen state) which typically succeeds.

```
Word_t stm_read(Word_t* addr) {
  ORec_t* p_orec = get_orec(addr);
  ORec_t orec = *p_orec;

  if (is_stolen(orec) || is_version(orec.version))
    goto slowpath;

  if (orec.version > my_read_timestamp)
    // assume inconsistency
    self_abort();

  return log_read(addr);

slowpath:
  // acquired myself, contention, or stolen orec case
  ...
  ...
}
```

Figure 5. Fast path for speculative (transactional) reads.

Our empirical results indicate that this overhead is not significant enough to make a noticable difference in performance.

5.4.2 Write Fast Path

```
void stm_write(Word_t* addr, Word_t value) {
  ORec_t* p_orec = get_orec(addr);
  ORec_t orec = *p_orec;

  if (is_stolen(orec) || is_version(orec.version))
    goto slowpath;

  if (orec.version > my_read_timestamp)
    // assume inconsistency
    self_abort();

  ORec_t new_orec = [my_ID | row | my_version];

  if (CAS(p_orec, orec, new_orec)) {
    log_write(addr, value);
    return;
  }

slowpath:
  // acquired myself, contention, or stolen orec case
  ...
  ...
}
```

Figure 6. Fast path for speculative (transactional) writes.

In the fast path for speculative (transactional) writes a transaction: computes the target location's orec address, reads the entire orec atomically in registers, makes the two ownership tests, builds the new orec contents in two 32-bit registers, CASes the new orec in the target orec, and logs the new value in the write set. Figure 6 depicts the fast path in C-like pseudocode form. Effectively, as compared to recent blocking STMs, the fast path requires two extra 32-bit registers (one each for the old and new orec values), some computation to determine the correct value of row, and memory accesses, which are usually cache hits, for the fields (my_ID, row, and my_version) of the writer's descriptor. However, the subsequent CAS acts as a memory barrier (our experimental machine contains UltraSPARC processors where the CAS instruction results in a memory barrier) and tends to hide the latency of many earlier operations. Our experiments indicate that the resulting fastpath overhead for writes is quite low.

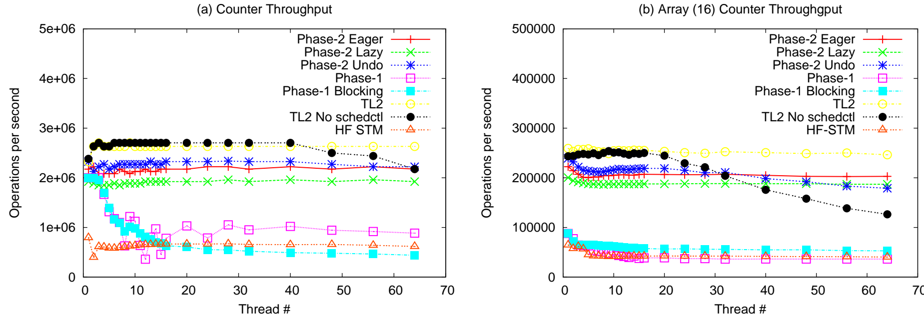

Figure 7. Throughput on the shared Counter (cntr) and Array of 16 Counters (array_cntr) benchmarks.

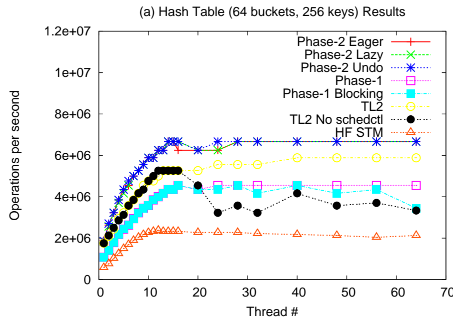

Figure 8. Throughput on a Hash Table (hashtable) consisting of 256 keys and 64 buckets.

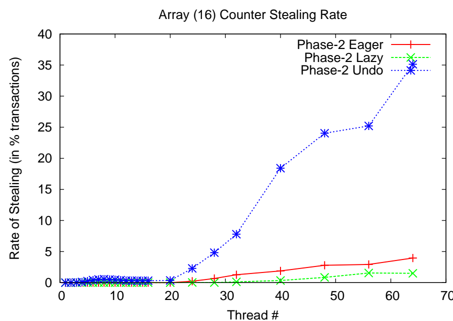

Figure 10. Stealing rate in Phase-2 STMs in array_cntr.

6. Performance Evaluation

6.1 Methodology

All our STMs were implemented as C language libraries. We implemented the Phase-1 blocking and nonblocking STMs (both employing eager orec acquisition policy), the redo log based eager and lazy variants, and the undo log based variant of our Phase-2 design. For comparison purposes we used the publically available HF-STM [6] and the TL2 library [3], both of which employ redo logs and lazy orec acquisition policy. Both these STM libraries are also implemented in C. In all cases we used the *Polite* contention manager, which employs exponential backoff during a conflict.

TL2 has been carefully engineered to minimize possibility of preemption when a transaction is in the process of releasing acquired locks. This was done by adding schedctl calls at the beginning of the lock-release phase. In our benchmarks, this strategy has been very effective for TL2 – in the absence of such schedctl calls, TL2's performance rapidly deteriorates with increasing preemption rate. Even though these strategies may be effective in general (as they are in our benchmarks), they rely on external system specific functionality and may not be easily portable.

We used four microbenchmarks to study performance of all these STMs.

Counter (cntr): is a simple shared counter that all threads continuously try to increment via transactions. This benchmark reflects behavior of high contention workloads consisting of extremely short transactions.

Array Counter (array_cntr): is a shared array of 16 counters, where all counters are incremented (starting from the smallest index) by transactions. It represents highly contended workloads with more realistic write set sizes.

Hash Table (hashtable): is a concurrent hash table consisting of 64 buckets and 256 keys. Each bucket contains an overflow linked list. A transaction either does an insert, a delete, or a lookup of a given key; we present results with 10/10/80 percent distribution of the three operation types respectively.

Binary Search Tree (bst): is a simple binary search tree. The set of operations and their distribution on bst were the same as that of hashtable. We tested the bst with two different sizes: 256 keys and 32K keys. The small size increases contention, whereas the larger size depicts performance in more realistic low-contention scenarios.

Note that our choice of microbenchmarks was deliberate. We believe such workloads amplify performance tradeoffs among blocking and nonblocking STM design choices. Due to their tendency of aggravating contention, they are more like "stress tests" for these systems. For low thread counts however, they also illustrate the behaviour of these STMs under the common contention free case. As a result, our microbenchmarks serve to study the behaviour of our STMs across a wide range of scenarios.

We conducted experiments on two machines: (i) a 16-processor Sun Fire 6800, a cache-coherent multiprocessor with 1.2 GHz UltraSPARC III processors; (ii) a Sun Fire T1000 UltraSPARC T1[13]-based single-chip multiprocessor (*T1000*) with 8 cores and 4 multiplexed threads per core. Results on the two machines were qualitatively similar. Due to space restrictions, we present results from the 16-processor system. All the libraries were compiled using *GCC v4.2.0* at *-O3* optimization level.

Memory Management. STM implementations impose some interesting restrictions on memory management systems [4, 12, 17]. In this paper, we do not address these issues, and use pre-allocated

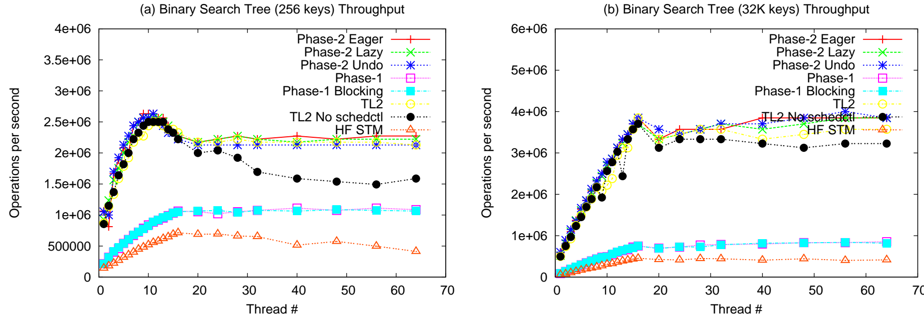

Figure 9. Throughput on Binary Search Tree (bst) with 256 keys and 32K keys.

data structures for our experiments (specifically for nodes in bst and hashtable). Although primitive, we believe that our approach isolates the effect of memory allocation performance/scalability on our results to a great extent, allowing us to more directly compare different STM designs.

6.2 Performance Results

Figures 7 through 9 show throughput in transactions per second with concurrent thread count ranging from 1 to 64 in all the microbenchmarks. Each thread executes 10^5 transactions in each test run. Throughput was averaged over 3 test runs. As may be clear from the performance curves, our Phase 2 extensions have boosted performance of our nonblocking STMs to the level of being competitive with state-of-the-art blocking STMs (TL2 in our results). We believe these results indicate the success of our efforts in significantly reducing the performance gap between blocking and nonblocking STMs, contrary to the commonly perceived notion that nonblocking STMs are "inherently" much slower than the blocking ones. Moreover, without schedctl support, TL2's throughput degrades significantly in several benchmarks, indicating the potential performance hazards of blocking STM implementations without scheduler support [22].

Specifically, Figure 7 compares performance of all STMs under very high contention workloads. Both cntr and array_cntr are write intensive benchmarks. These benchmarks show the still existing fastpath overhead in the write operation in our STMs. We believe these to be examples of "worst case" behavior for our STMs. In more realistic benchmarks the ratio of reads to writes is expected to be higher thus reducing this performance gap. Notice that our undo log STM performs slightly better than the redo log, eager acquire based nonblocking STM. This is because of the extra memory operations required in the latter STM to redo its speculative updates from its write set to the target locations. The lazy acquire version incurs slightly more overhead of replicating the write set at commit time, which manifests in the throughput curves.

Notice that the throughput of our undo log STM degrades slightly after hitting preemption in the array_cntr experiments. Figure 10 gives an insight into the reason behind the degradation. Recall that transaction aborts may trigger stealing in our undo log STM. The abort rate in array_cntr peaks at about 25% for all STMs by the time we reach 16 threads, and remains there with increasing thread count. This relatively high abort rate, coupled with arbitrary delays due to preemption are responsible for a drastic increase in the orec stealing rate in our undo log STM, to up to 35%. In comparison, the stealing frequency is not as dramatically affected in the redo log STMs as shown in Figure 10.

The impact on throughput of array_cntr, although not really significant, is noticable in Figure 7. Since stealing was the primary reason for performance degradation of our undo log STM in array_cntr, we experimented with alternate means of delaying the stealing process, e.g. a transaction exponentially backs off before stealing an orec. Although the stealing frequency dropped, the overall throughput did not change noticably. Interestingly, though cntr does not permit any scaling, we observed that the abort rate is not high (lower than 2%). The transactions are short enough that the window for aborting conflicting transactions is too small.

Our observation brings out an interesting new tradeoff between undo and redo log based STMs from the perspective of nonblocking progress guarantees: too much contention may result in worse performance degradation in undo log STMs.

The hash table and the binary search tree are scalable workloads that exhibit low contention. As is clear from all three graphs, our nonblocking STMs are competitive with TL2. In fact, our STMs scale better than TL2 on the hashtable benchmark.

7. Discussion

Nonblocking progress conditions have some interesting interactions with some aspects of transactional memory runtimes such as privatization [26] and condition synchronization [7].

Privatization. To allow memory to be accessed both transactionally and non-transactionally, and to support dynamic memory allocation [3], STMs should generally support *privatization* [25, 26, 28]. Most privatization solutions known to date are blocking, and it seems likely that practical solutions will be blocking. Nonetheless, some purposes of privatization can be supported in a nonblocking manner, for example, dynamic memory allocation can be supported by deferring freeing of blocks of memory until they have been privatized, without blocking the thread that frees memory. Furthermore, some applications may not require privatization. Therefore, although some restrictions may be needed, the privatization problem does not prevent the construction of useful nonblocking STMs.

Conditional Waiting. Harris et al. [7] introduced the retry construct for condition synchronization in memory transactions. Although conditional waiting may seem in direct conflict with nonblocking STMs, we argue that there is no real conflict here – conditional waiting is a part of application semantics where a thread intentionally waits for an event, and nonblocking STMs guarantee forward progress in the absence of such intentions.

8. Conclusion

Although there may be some fundamental design tradeoffs between blocking and nonblocking STMs, we claim that their common case performance may not be inherently different. To support that claim, we showed how to decouple fast path transactional code from the

infrastructure that enables nonblocking progress in our STMs. This decoupling enables more recent optimizations of blocking STMs in our nonblocking STMs with surprisingly simple extensions. Our work has shown that most key optimizations applied to state-of-the-art blocking STMs can also be applied to nonblocking STMs. Moreover, nonblocking STMs are naturally tolerant of pathologies such as preemption, priority inversion, and thread failures. To our knowledge, our work is the first to show that one can build an undo log based nonblocking STM. Empirical results show that all variants of our nonblocking STM are competitive with state-of-the-art blocking STMs. Future directions include more rigorous experimentation with larger workloads and migrating our ideas to object-based nonblocking STM designs.

9. Acknowdegments

We are grateful to Yossi Lev for initial discussions, and to Dave Dice for providing us with a copy of TL2. We are also grateful to the anonymous reviewers, Michael Scott, and Micheal Spear for useful comments on the paper.

References

[1] C. S. Ananian, K. A. amd Bradley C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded Transactional Memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, 2005.

[2] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid Transactional Memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 336–346, 2006.

[3] D. Dice, N. Shavit, and O. Shalev. Transactional Locking II. In *Proceedings of the 20th International Symposium on Distributed Computing*, pages 194–208, 2006.

[4] K. Fraser and T. Harris. Practical Lock-Freedom. Technical Report UCAM-CL-TR-579, Cambridge University Computer Laboratory, 2004.

[5] L. Hammond, V. Wong, M. Chen, B. Hertzberg, B. D. Carlstrom, J. D. Davis, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, 2004.

[6] T. Harris and K. 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.

[7] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–60, 2005.

[8] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing Memory Transactions. In *ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, page to appear, June 2006.

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

[10] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, pages 92–101, 2003.

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

[12] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. Hertzberg. McRT-Malloc: A Scalable Transactional Memory Allocator. In *Proceedings of the 5th International Symposium on Memory Management*, pages 74–83, 2006.

[13] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way Multithreaded Sparc Processor. *IEEE Micro*, 25(2):21–29, 2005.

[14] V. J. Marathe and M. Moir. Efficient Nonblocking Software Transactional Memory. Technical report, Forthcoming Technical Report, Sun Microsystems Laboratories.

[15] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Design Tradeoffs in Modern Software Transactional Memory Systems. In *Proceedings of the 7th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems*, October 2004.

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

[17] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. S. III, and M. L. Scott. Lowering the overhead of nonblocking software transactional memory. In *the 1st ACM SIGPLAN Workshop on Transactional Computing*, 2006.

[18] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based Transactional Memory. In *Proceedings of the 12th Annual International Symposium on High Performance Computer Architecture*, pages 258–269, 2006.

[19] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 494–505, 2005.

[20] H. E. Ramadan, C. J. Rossbach, D. E. Porter, O. S. Hofmann, A. Bhandari, and E. Witchel. MetaTM/TxLunix: Transactional Memory for an Operating System. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 92–103, 2007.

[21] T. Riegel, C. Fetzer, and T. Hohnstein. Time-based Transactional Memory with Scalable Time Bases. In *Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 221–228, 2007.

[22] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A High Performance Software Transactional Memory System for a Multi-Core Runtime. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 187–197, 2006.

[23] W. N. Scherer III and M. L. Scott. Advanced Contention Management in Dynamic Software Transactional Memory. In *Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing*, pages 240–248, 2005.

[24] N. Shavit and D. Touitou. Software Transactional Memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[25] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing Isolation and Ordering in STM. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 78–88, 2007.

[26] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization Techniques for Software Transactional Memory. Technical Report TR 915, Department of Computer Science, University of Rochester, 2007.

[27] F. Tabba, C. Wang, J. R. Goodman, and M. Moir. NZTM: Nonblocking Zero-Indirection Transactional Memory. In *the 2nd ACM SIGPLAN Workshop on Transactional Computing*, 2007.

[28] C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and A.-R. Adl-Tabatabai. Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language. In *Proceedings of the 5th International Symposium on Code Generation and Optimization*, pages 34–48, 2007.